Nexus: An Interoperability Layer for Parallel and Distributed Computer Systems

Ian Foster, Carl Kesselman
Robert Olson, Steven Tuecke

CRPC-TR94456
May, 1994

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

This work was supported in part by the Office of Scientific Computing, U.S. Department of Energy, and the CRPC.

Nexus: An Interoperability Layer for Parallel and Distributed Computer Systems*

Ian Foster
Carl Kesselman
Robert Olson
Steven Tuecke

version 1.10
May 6, 1994

Abstract

Nexus is a set of services that can be used to implement various task-parallel languages, data-parallel languages, and message-passing libraries. Nexus is designed to permit the efficient, portable implementation of individual parallel programming systems and the interoperability of programs developed with different tools. Nexus supports light-weight threading and active message technology, allowing integration of message passing and threads.

1 Introduction

Nexus is a set of services that can be used to implement many different parallel programming tools, including task-parallel languages such as Fortran M and CC++; data-parallel languages such as pC++ and High Performance Fortran (HPF); and message-passing libraries such as Message Passing Interface (MPI), p4, and PVM. Nexus is intended as a compiler target or as a basis for a higher-level library, not for direct use by an end-user programmer.

Nexus services provide direct support for light-weight threading, address space management, communication, and synchronization. A computation consists of a set of *threads*, each executing in an address space called a *context*. An individual thread executes a sequential

*This work was supported in part by the Office of Scientific Computing, U.S. Department of Energy, under Contract W-31-109-Eng-38, and in part by the National Science Foundation's Center for Research in Parallel Computation under Contract CCR-8809615.

program, which may read and write data shared with other threads executing in the same context. It can also generate asynchronous *remote service requests*, which invoke procedures in other contexts. In a heterogeneous system, arguments to a remote service request are automatically translated to a machine-independent format.

Nexus is currently being used as a compiler target for Fortran M and CC++ compilers. These compilers generate an initial program and a set of *handler routines*, which can be invoked by using remote service requests. The main program and handler routines are compiled and linked with the Nexus library to produce an executable program. HPF compilers and message-passing libraries can also be modified to use Nexus services. Hence, Nexus makes it possible to combine, in a single application, programs developed with different tools.

2 Basic Abstractions

Nexus supports five basic abstractions: the *node*, *context*, *thread*, *global pointer*, and *remote service request*.

Node. A *node* represents a physical processing resource. It is distinguished by its location (machine name and, in a multicomputer, processor number). A node may variously correspond to a physical processor, a shared-memory multiprocessor, or a Unix process.

Context. A *context* is an address space plus an executable program. A context is located within a node; more than one context can be allocated to a node. The address space consists of one or more data segments. An initial data segment is created when a context is allocated, and additional segments can be added as needed. A data segment can be part of only one context. The program associated with a context defines a `NexusBoot()` routine, which is invoked when the context is created; a `NexusExit()` routine, which is invoked when the context is destroyed; and any handlers that may be invoked by using remote service requests (see below).

Thread. A *thread* is a thread of control. A thread is located within a context; more than one thread can be allocated to a context. Hence, the mapping of computation to physical processors is determined by both the mapping of threads to contexts and the mapping of contexts to nodes. The relationship between nodes, contexts, and threads is illustrated in Figure 1.

Global Pointer. A *global pointer* is a {Node,Context,Address} triple. A thread can access data within its context as local data. To access data within a different context (on the same or different node), it must be provided with a global pointer.

Remote Service Request. A thread can request that an action be performed on a remote node by issuing a *remote service request*. This takes a handler identifier, a global pointer, and a message buffer as arguments and causes the specified handler to be executed on the

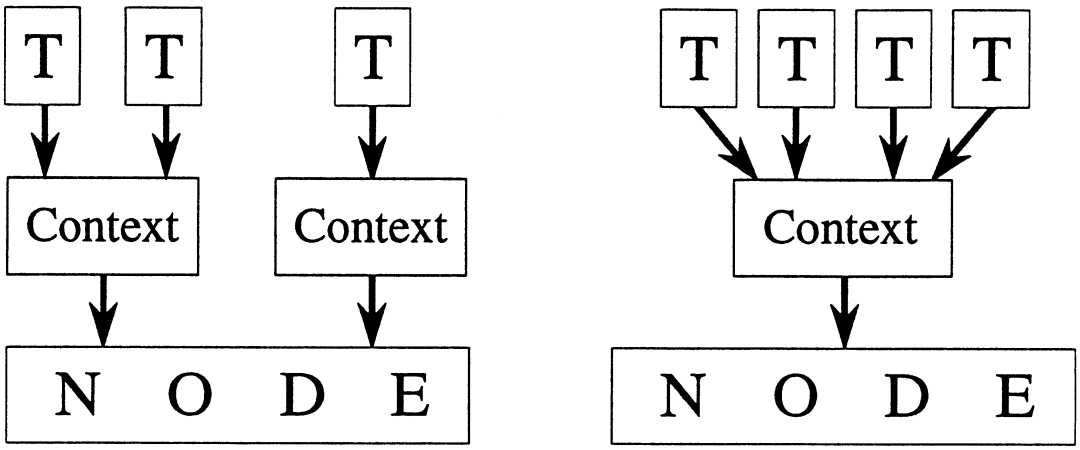

Figure 1: Nodes, Contexts, and Threads

node and within the context of the global pointer. The handler is passed the local address component of the global pointer and the message buffer as arguments.

These abstractions can be used to implement a variety of programming language concepts. For example, a Fortran M *process* corresponds to a context (used to hold process common) plus a thread (the thread of control); a Fortran M send statement may be translated into a remote service request that places a message in a remote message queue in the appropriate process's context. A CC++ processor object corresponds to a context; CC++ statements called in parallel blocks execute as threads (if local) or as remote service requests (if remote).

3 Nexus Runtime Library

The Nexus runtime library provides primitive functions for node management, context management, thread management, and communication. In the following sections, threads and global pointers are represented by the C-language typedefs `nexus_thread_t` and `nexus_global_pointer` respectively.

Any program that uses Nexus functions must include "`nexus.h`".

3.1 Initialization and Argument Handling

Nexus is intended to be both portable and usable by diverse programming packages. Therefore, initialization and argument handling in Nexus must be able to deal with a wide variety of situations.

Definitions.

- *package*: The system using Nexus, such as Fortran M or CC++.
- *application*: The end-user program that uses a package which is implemented upon Nexus.
- *package designator*: The string (i.e., "`-fm`") used to separate command line arguments destined for the application from those destined for Nexus and the package. Arguments before the first package designator are for the application, and those after it are for Nexus and the package.
- *process*: An address space and set of associated resources that will be used by Nexus to implement its node and context abstractions. Normally this corresponds to a Unix process.
- *master node*: One of the Nexus nodes is designated the master node. This node has the special task of initiating execution of the application program.

Initialization Model. Nexus initialization is designed to handle various combinations of the following situations:

- A single process is created, which starts additional processes during Nexus initialization. This is a common approach for starting a parallel program on a network of workstations. A variety of methods may be used to start the other processes, including `rsh`, special startup daemons, etc. Each process represents a separate node. The first process is designated as the master node.
- All processes are started simultaneously by operating system tools. This is a common approach on massively parallel processing machines. Again, each process represents a separate node. One process is designated the master node.
- Processes are added to the parallel program dynamically through either context creation (§ 3.3.1) or node creation (§ 3.2.1).

In each case, the application is assumed to start execution as a single thread of control on the master node. It can then create additional threads of control by making remote service requests, create new processes using `nexus_acquire_nodes()`, etc.

Initialization is performed in two phases: `nexus_init()` begins initialization and `nexus_start()` completes it. In Fortran M, the main routine simply calls `nexus_init()` followed immediately by `nexus_start()`. In contrast, CC++ requires that Nexus be initialized before the first global constructor is called, which is long before `main()` is called. The split initialization allows `nexus_init()` to be called from the first global constructor, followed by the execution of the remaining constructors, followed by a call to `nexus_start()` from within `main()`.

System Configuration. The system can be configured at runtime using arguments provided using one or more of three mechanisms: an environment variable, command line arguments (if they are accessible), and a parameters string. The third mechanism is used when a process is created by a pre-existing Nexus node. Redundant arguments in the parameters string override arguments in the environment variable, and arguments in the command line override both.

Because `nexus_init()` may be called prior to `main()`, the command line arguments cannot simply be extracted from the argument list (`argc/argv`) that is normally passed to `main()`. (Of course, Fortran does not have `argc/argv`, which is another problem with that approach.) Instead, Nexus must find the command line arguments through some other method, such as the `environ` variable on most Unix machines. If Nexus succeeds, `nexus_init()` will

1. retrieve the command line arguments (from the `environ` variable),
2. combine the command line arguments with the environment variable arguments and the parameters string,
3. make the application destined arguments available through `nexus_user_iargc()`, `nexus_user_ge` and `nexus_get_argc_and_argv()`, and
4. parse the Nexus and package arguments. The package is allowed access to any arguments following the package designator that Nexus does not recognize through function callbacks provided by the package in the `nexus_init()` call.

If Nexus cannot retrieve the command line arguments (or if a particular startup method does not allow the use of command line arguments), then only the environment variable and parameters string can be used.

3.1.1 `nexus_init()`

```
void nexus_init(char *args_env_variable,
                char *package_designator,
                void (*package_args_init_func)(),
                int  (*package_args_func)(int,int),
                void (*usage_message_func)(),
                int  (*new_process_params_func)(char *,int)
                nexus_global_pointer_t **node_gps,
                int *n_node_gps)
```

Initialize a process for Nexus. The `args_env_variable` argument holds the name of the environment variable to check for configuration information. The `package_designator` argument holds the package designator string. The

`package_args_init_func`, `package_args_func`, `usage_message_func`, and `new_process_params_func` arguments are pointers to package callback functions as described below, or `NULL` pointers to designate no callback.

This routine should be the first one called on any new process that will be used within a Nexus computation. Depending on the particular Nexus implementation, a new process is created when a node and/or when a context is created; arguments automatically passed to this process by the Nexus run-time system will allow `nexus_init()` to distinguish these cases.

`nexus_init()` first parses the environment variable, the parameters string, and command line arguments. This process consists of the following steps:

1. The command line arguments are split (using the passed package designator) and combined with the environment variable and parameters string, as described above.
2. The `package_args_init_func` package callback function is called. The package can use this to initialize variables that will be used to hold configuration information extracted from the arguments.
3. Nexus invokes the `package_args_func` package callback function once for each argument that is not recognized by Nexus, passing the current argument number and the total number of arguments. Using these in conjunction with `nexus_package_getarg()`, it can decide whether a particular argument is meant for the package. If so, it should extract all relevant information into its own variables. It should return a new current argument that is greater than or equal to the current argument it was passed.
4. If an error occurred during argument parsing (an argument was recognized neither by Nexus nor by the package), the `usage_message_func` package callback is called. It should print a usage message for its arguments to `stdout`.

Nexus is now initialized. A subsequent call to `nexus_start()` should now be made to complete the initialization. Package initialization can be performed between the `nexus_init()` and `nexus_start()` calls.

On machines which start a set of processes simultaneously, such as on many parallel computers, `nexus_init()` will be called simultaneously in each of these processes. One of these processes will be designated the master node. On other machines, such as workstations, command line arguments to the master node process may cause `nexus_init()` to create other nodes, perhaps on other workstations.

All nodes, including the master, will automatically have a default context created on them. On the master node, `nexus_init()` sets its `node_gps` argument to an array of global pointers to these contexts, and `n_node_gps` to the number of global pointers in this array. The master node's global pointer is always the first in this array. Subsequent context creations and

remote service requests may be used to run threads on those nodes or on additional nodes created by `nexus_acquire_nodes()` (see § 3.2.1). On processes other than the master node, `nexus_init()` sets `node_gps` to `NULL` and `n_node_gps` to zero.

The array of global pointers returned in `nodes_gps` by this function is allocated using `nexus_malloc()`. Therefore, when it is no longer needed each global pointer should be destroyed using `nexus_destroy_global_pointer()`, and the array freed using `nexus_free()`.

The `new_process_params_func` package callback function is saved by Nexus for future use. Whenever a new process is created for a node and/or context, this routine is called with a character buffer and the buffer size. This callback function should fill in the buffer with any (space separated) package arguments that need to be passed to the new process; it should return the number of characters that it wrote into the buffer, up to the passed buffer size.

3.1.2 nexus_start()

```
void nexus_start()
```

This must be called after `nexus_init()`. On the master node, `nexus_start()` returns. In all other cases, `nexus_start()` does not return; program execution on these nodes is invoked by subsequent context creations and remote service requests to these nodes.

3.1.3 nexus_user_iargc()

```
int nexus_user_iargc()
```

Return the number of user arguments (number of arguments up to, but not including, the package designator argument).

3.1.4 nexus_user_getarg()

```
char *nexus_user_getarg(int i)
```

Return a pointer to the string that represents the `i`'th user argument. This string should not be modified.

3.1.5 nexus_get_argc_and_argv()

```
void nexus_get_argc_and_argv(int *argc,
                             char ***argv)
```

Store at the locations referenced by `argc` and `argv` the number and location of the user arguments.

3.1.6 nexus_package_iargc()

```
int nexus_package_iargc()
```

Return the number of package arguments (number of arguments after the package designator argument).

3.1.7 nexus_package_getarg()

```
char *nexus_package_getarg(int i)
```

Return a pointer to the string that represents the i'th package argument. This string should not be modified.

3.2 Node Management

Three primitive functions are provided for manipulating the set of nodes in a Nexus computation:

- nexus_acquire_nodes: bring additional nodes into the computation
- nexus_release_nodes: release nodes from the computation
- nexus_current_node: obtain the current node's node descriptor (see §3.7)

3.2.1 nexus_acquire_nodes()

```
void nexus_acquire_nodes_on_host(
                char *host_name,
                nexus_path_name *path_name,
                int count,
                nexus_global_pointer_t **node_gps,
                int *n_node_gps)


void nexus_acquire_nodes_of_type(
                nexus_arch_type_t type,
                nexus_path_name *path_name,
                int count,
                nexus_global_pointer_t **node_gps,
                int *n_node_gps)
```

Both of these functions introduce count new nodes into the computation. The first call obtains those nodes from the machine specified by host_name. The second version of the call allocates nodes of a specific architecture type.

The path_name argument specifies the path of the Nexus node server executable (i.e., an application compiled with Nexus, or a generic Nexus node server). If path_name is NULL, then the path to the current executable on this node is used. If host_name is NEXUS_MY_HOST, count nodes are allocated on the same host as the current node. If type is NEXUS_TYPE_ANY, count nodes of any type are allocated. Each routine returns an array of global pointers in node_gps, one for each allocated node, and the size of this array in n_node_gps. If not node can be acquired, node_gps is set to NULL and n_node_gps is set to 0.

The array of global pointers returned in nodes_gps by these functions is allocated using nexus_malloc(). Therefore, when it is no longer needed each global pointer should be destroyed using nexus_destroy_global_pointer(), and the array freed using nexus_free().

3.2.2 nexus_release_nodes()

```
int nexus_release_nodes(nexus_global_pointer_t *node_gps,
                        int n_node_gps)
```

Remove a set of nodes from the computation. The nodes to be released are specified by the global pointer array node_gps, which has n_node_gps elements. All contexts and threads that are active on the specified nodes will be destroyed. Returns 0 if the nodes are successfully released, or -1 otherwise.

3.3 Context Management

Primitive functions are provided for context operations:

- nexus_init_create_context_handle: initialize context creation handle
- nexus_create_context: create a context
- nexus_create_context_wait: wait for context creation(s) to complete
- nexus_destroy_current_context: destroy a context

A context, when created, consists of an *initial data segment* (a block of memory in the context which is requested during context creation) and an executable program.

Data segments can be manipulated with the operations:

- `nexus_context_initial_segment`: retrieve the initial data segment of the current context
- `nexus_malloc`: allocate a data segment in the current context
- `nexus_free`: free a data segment from the current context

The executable program is loaded from a user-specified location. This program may contain the following functions:

- `NexusBoot`: a function that is invoked upon creation of the context (required)
- `NexusExit`: a handler that is invoked upon termination of the context (optional)
- `NexusUnknownHandler`: a handler that is invoked if a remote service request is made to this context with an unknown handler (optional)

See § 4 for more information on these functions.

3.3.1 `nexus_create_context()`

```
void nexus_create_context(nexus_global_pointer_t *node_gp,
                          char *executable_path,
                          int size,
                          nexus_global_pointer_t *new_context_gp,
                          int *return_code,
                          nexus_create_context_handle_t *contexts)
```

Create a context on the same node as the context pointed to by the `node_gp` argument. The context is allocated an *initial data segment* of `size` bytes and loads the executable specified by `executable_path`.

A global pointer to the new context's initial data segment is placed in `new_context_gp`, and a return code is placed in `return_code`, once the context has been initialized. A non-zero `return_code` indicates that the context creation failed. `NexusBoot()` (§ 4.1) is automatically invoked in the new context when it is created, and that context does not complete initialization until `NexusBoot()` has returned. If `NexusBoot()` returns non-zero in this context, then the context creation fails and that return value is placed in `return_code`.

If `contexts` is `NULL`, then this call waits for the new context to be initialized, and `new_context_gp` is valid immediately upon return of this call. If `contexts` is a valid `nexus_create_context_handle_t` that was initialized by `nexus_init_create_context_handle()`, then this call will return before the new context is initialized, and `new_context_gp` is not valid until after the subsequent `nexus_create_context_wait()` call using this `contexts` handle.

3.3.2 nexus_init_create_context_handle()

```
void nexus_init_create_context_handle(
              nexus_create_context_handle_t *contexts,
              int n_contexts)
```

Initialize the context handle, `contexts`, to be used by `n_contexts` subsequent calls to `nexus_create_context`.

Use of a context handle when creating multiple contexts allows the context creations to overlap with each other, and to overlap with work by the package.

3.3.3 nexus_create_context_wait()

```
void nexus_create_context_wait(nexus_create_context_handle_t *contexts)
```

Wait for the contexts created using the `contexts` handle to be initialized.

3.3.4 nexus_destroy_current_context()

```
void nexus_destroy_current_context()
```

Deallocate the current context by freeing all data segments acquired during its execution. The thread that calls this function is terminated. The behavior is undefined if other threads executing on the current context have not terminated.

Before Nexus services have been deallocated, a user-defined function `NexusExit()` is called. This can be used to free package data structures, etc. `NexusExit()` returns a void and takes no arguments.

3.3.5 nexus_context_initial_segment()

```
void *nexus_context_initial_segment()
```

Return a pointer to the initial data segment of the context of the calling thread.

3.3.6 nexus_malloc()

```
void *nexus_malloc(size_t size)
```

Allocate a data segment with `size` bytes and add it to the current context. Return a pointer to this newly allocated data segment, or `NULL` if it cannot be allocated.

3.3.7 nexus_free()

```
void nexus_free(void *data_segment)
```

Free the data segment pointed to by data_segment from the current context. This operation may be applied to the context's initial data segment.

3.4 Thread Management

Nexus threads are modeled after a subset of POSIX threads (IEEE standard P1003.4a).

Primitive functions are provided for basic thread operations:

- nexus_thread_create: create a thread
- nexus_thread_exit: terminate the current thread
- nexus_thread_yield: yield the processor to another thread
- nexus_thread_self: return the thread ID of the calling thread
- nexus_thread_equal: compare two thread IDs
- nexus_thread_once: for dynamic module initialization
- nexus_thread_key_create: create a thread specific data key
- nexus_thread_setspecific: associate a value with a thread specific data key
- nexus_thread_getspecific: retrieve the value associated with a thread specific data key

Mutual exclusion and synchronization between threads is provided by the operations:

- nexus_mutex_init: initialize a mutual exclusion lock
- nexus_mutex_destroy: destroy a lock
- nexus_mutex_lock: obtain a mutually exclusive access to lock
- nexus_mutex_unlock: release a lock
- nexus_cond_init: initialize a condition variable
- nexus_cond_destroy: destroy a condition variable

- `nexus_cond_wait`: wait for a condition
- `nexus_cond_signal`: signal a condition
- `nexus_cond_broadcast`: signal to all waiting for a condition

3.4.1 `nexus_thread_create()`

```
typedef void *(*nexus_thread_func_t)(void *user_arg);

int nexus_thread_create(nexus_thread_t *thread,
                        nexus_thread_attr_t *attr,
                        nexus_thread_func_t func,
                        void *user_arg)
```

Create a new thread in the current context which invokes the supplied function `func` with one argument `user_arg`. The thread ID for the newly created thread is placed in `thread`. Return zero if the thread is successfully created, or non-zero otherwise.

The `attr` argument for specification of attributes for the thread. However, this is not yet implemented.

Note: There are no equivalents to `pthread_detach()` and `pthread_join()` in Nexus. All Nexus threads are automatically detached when they are created.

3.4.2 `nexus_thread_exit()`

```
void nexus_thread_exit(void *status)
```

Terminate the calling thread. Returning from the user thread function will implicitly terminate the thread.

The `status` argument is currently not used.

3.4.3 `nexus_thread_yield()`

```
void nexus_thread_yield()
```

Yield the processor to another thread.

3.4.4 `nexus_thread_self()`

```
nexus_thread_t nexus_thread_self()
```

Return the thread ID of the calling thread.

3.4.5 nexus_thread_equal()

```
int nexus_thread_equal(nexus_thread_t t1,
                       nexus_thread_t t2)
```

Compare the two thread IDs, t1 and t2. If they are the same then this returns non-zero, otherwise it returns zero.

3.4.6 nexus_thread_once()

```
nexus_thread_once_t once_control = NEXUS_THREAD_ONCE_INIT;

int nexus_thread_once(nexus_thread_once_t *once_control,
                      void (*init_routine)() )
```

The first call to nexus_thread_once() by any thread in a context, with a given once_control, will call the init_routine() with no arguments. Subsequent calls of nexus_thread_once() will not call the init_routine(). On return of nexus_thread_once() it is guaranteed that init_routine() has completed. The once_control parameter is used to determine whether the associated initialization routine has been called.

This returns 0 upon successful completion, otherwise -1.

3.4.7 nexus_thread_key_create()

```
typedef void (*nexus_thread_key_destructor_func_t)(void *value);

int nexus_thread_key_create(
                nexus_thread_key_t *key,
                nexus_thread_key_destructor_func_t func)
```

Create a thread specific data key that is visible to all threads in the context, and place that key in the key argument. Although the same key value may be used by different threads, the values bound to the key by nexus_thread_setspecific() are maintained on a per-thread basis.

The value associated with a new key is NULL in all active threads, and will be initialized to NULL in all threads that are subsequently created.

If func is not NULL, then upon termination of the thread if the value for this key is not NULL the function pointed to by func is called with the current value for the key as its argument.

This function returns zero upon successful completion, or non-zero otherwise. A -1 return indicates that the key name space is exhaused.

3.4.8 nexus_thread_setspecific()

```
void nexus_thread_setspecific(nexus_thread_key_t key,
                              void *value)
```

Set the value associated with the thread specific data `key` to `value`. Different threads may bind different values to the same key.

3.4.9 nexus_thread_getspecific()

```
void nexus_thread_getspecific(nexus_thread_key_t key,
                              void **value)
```

Get the thread specific data value associated with `key`, and return it in the `value` argument.

3.4.10 nexus_mutex_init()

```
void nexus_mutex_init(nexus_mutex_t *mutex,
                      nexus_mutexattr_t *attr)
```

Initialize the mutual exclusion lock, `mutex`. The attributes for the mutex are specified by `attr`. Default attributes will be used if `attr` is `NULL`.

The result of calling `nexus_mutex_lock()` or `nexus_mutex_unlock()` on a mutex that has not been initialized is undefined.

3.4.11 nexus_mutex_destroy()

```
void nexus_mutex_destroy(nexus_mutex_t *mutex)
```

Destroy the `mutex` that was initialized with `nexus_mutex_init()`. The result of calling `nexus_mutex_lock()` or `nexus_mutex_unlock()` on a mutex that has been destroyed is undefined.

3.4.12 nexus_mutex_lock()

```
void nexus_mutex_lock(nexus_mutex_t *mutex)
```

Block until the mutual exclusion lock, `mutex`, is acquired. This may or may not be implemented as a spin lock (i.e., busy wait).

3.4.13 nexus_mutex_unlock()

```
void nexus_mutex_unlock(nexus_mutex_t *mutex)
```

Unlock the mutual exclusion lock, `mutex`, enabling another thread to acquire the mutex. Fairness in locking is not guaranteed; that is, a thread is not guaranteed to acquire a lock if other threads are also attempting to acquire the same lock.

3.4.14 nexus_cond_init()

```
void nexus_cond_init(nexus_cond_t *cond,
                     nexus_condattr_t *attr)
```

Initialize the condition variable, `cond`. The attributes for the condition are specified by `attr`. Default attributes will be used if `attr` is `NULL`.

The result of calling any other `nexus_cond_*()` function on a condition that has not been initialized is undefined.

3.4.15 nexus_cond_destroy()

```
void nexus_cond_destroy(nexus_cond_t *cond)
```

Destroy the specified condition. The result of calling any other `nexus_cond_*()` function on a condition that has been destroyed is undefined.

3.4.16 nexus_cond_wait()

```
void nexus_cond_wait(nexus_cond_t *cond,
                     nexus_mutex_t *mutex)
```

Atomically release `mutex` and wait on `cond`. When the function returns, `mutex` has been reacquired.

If the thread executing the function has not acquired `mutex`, the result is undefined.

3.4.17 nexus_cond_signal()

```
void nexus_cond_signal(nexus_cond_t *cond)
```

Signal the specified condition, waking up one thread that is suspended on this condition. If no threads are suspended on this condition, this call will have no effect.

3.4.18 nexus_cond_broadcast()

```
void nexus_cond_broadcast(nexus_cond_t *cond)
```

Unsuspend all threads suspended on the specified condition.

3.5 Communication

Functions are provided for issuing remote service requests:

- `nexus_init_remote_service_request`: initiate a remote service request
- `nexus_sizeof_TYPE`: determine the amount of message buffer space needed to send data values of a special TYPE
- `nexus_put_TYPE`: place a data value of a special TYPE in a message buffer
- `nexus_check_buffer_size`: check the message buffer for overflow, and, if necessary, resize the buffer
- `nexus_send_remote_service_request`: issue a remote service request

When a remote service request is executed by a handler, primitive functions are provided for handling it:

- `nexus_get_TYPE`: extract a data value of a special TYPE from a message buffer
- `nexus_stash_buffer`: stash the buffer so that it is accessible outside of the message handler
- `nexus_get_stashed_TYPE`: extract a data value of a special TYPE from a stashed message buffer
- `nexus_free_stashed_buffer`: free a stashed buffer

Other handler related primitives include the following:

- `nexus_register_handlers`: register handlers
- `nexus_substitute_handler`: change a handler registration
- `nexus_poll`: handle any outstanding messages

Handlers can be either threaded or non-threaded. A threaded handler executes in a thread created specifically for it, is passed a stashed buffer of type `nexus_stashed_buffer_t`, and has no restrictions on what it may do. A non-threaded handler executes in an existing thread, is passed a buffer of type `nexus_buffer_t`, and has some restrictions on what it may do. The following are noteworthy items regarding handler design and behavior:

- Remote service requests between any two contexts using non-threaded handlers are performed in sequence. No other constraint is placed on the order in which remote service requests are executed; in particular, they can be executed concurrently with other non-threaded handlers requested from other contexts, and with other threaded handlers from any context. There is no ordering of threaded handlers.

- The buffer that is passed to a non-threaded handler is automatically freed upon exit from the handler. If it is required that the buffer persist after the completion of the handler, then it must be *stashed* using `nexus_stash_buffer()` and later freed by `nexus_free_stashed_buffer()`.

- The stashed buffer that is passed to a threaded handler must be explicitly freed by a call to `nexus_free_stashed_buffer()`.

- The `nexus_get_TYPE()` routines can be used to operate only on `nexus_buffer_t` buffers, and only within non-threaded handlers. The `nexus_get_stashed_TYPE(` routines must be used to access stashed buffers, either from threaded handlers or from other threads.

- A non-threaded handler must not suspend indefinitely. In some implementations of Nexus, all non-threaded handlers may be called in sequence from within a single thread. Therefore, if a non-threaded handler suspends, it will indefinitely postpone the execution of other handlers. If the behavior of a handler requires suspension, that handler must either create a thread using `nexus_create_thread()` in which to implement that behavior, or must be a threaded handler.

- Any data that is put in a buffer (using `nexus_put_TYPE()` must not change between the `nexus_init_remote_service_request()` and the `nexus_send_remote_service_request()`. This allows some implementations of Nexus to optimize away a data copy when sending remote service requests.

- Some care must be taken to avoid deadlock between the thread sending a remote service request and the non-threaded handler for that remote service request. For example, when sending a remote service request from a context to the same context, the non-threaded handler may be invoked from the same thread that sent the remote service request. Thus, the non-threaded handler must not try to lock a resource which the sending thread already has locked and will not release until after the send has completed.

3.5.1 nexus_init_remote_service_request()

```
void nexus_init_remote_service_request(
                nexus_buffer_t *buffer,
                nexus_global_pointer_t *gp,
                char *handler_name,
                int handler_id,
                int force_translation)
```

Allocate and initialize a message buffer to be sent to the node specified by gp. gp is used to determine what sort of data translation is needed, if any. If the force_translation flag is set to a nonzero value, data placed in the buffer will always be translated to a machine-independent format.

The handler invoked to service the request is specified by the value of the handler_id and handler_name arguments. Handler names and identifiers are local to a context. The handler will be invoked in the context specified by gp with two arguments: the local address corresponding to gp, and a local pointer to a buffer with the same contents as buffer.

Return a buffer in buffer that can be packed using nexus_put_TYPE calls, or NULL if a buffer cannot be successfully initialized.

Invoking init_remote_service_request() on a NULL global pointer results in a fatal run-time error.

3.5.2 nexus_sizeof_TYPE()

```
size_t nexus_sizeof_TYPE(nexus_buffer_t *buffer,
                         int count)
```

Return the size in bytes required to encode count items of data type TYPE into buffer. This can be used to calculate the precise size of the message buffer needed for a given message.

3.5.3 nexus_sizeof_global_pointer()

```
int nexus_sizeof_global_pointer(nexus_buffer_t *buffer,
                                nexus_global_pointer_t *gp,
                                int count,
                                int *n_elements)
```

Return the size in bytes required to encode the first count elements of the global pointer array, gp. Set n_elements to the number of nexus_put_TYPE() operations that will be used by nexus_put_global_pointer() to put these global pointers into the buffer; this is for use with nexus_set_buffer_size().

The size of a global pointer varies, depending upon the context to which it points; thus the extra gp argument. Global pointers that point to the same context have the same size.

3.5.4 nexus_set_buffer_size()

```
void nexus_set_buffer_size(nexus_buffer_t *buffer,
                           int size,
                           int n_elements)
```

Sets `buffer` to be able to hold `size` bytes of data and `n_elements` of data. `n_elements` should be the number of `nexus_put_TYPE()` operations that will subsequently be invoked on this buffer, or -1 if that number is not known.

This procedure does not have to be called. If it is not called, the number of `nexus_put_TYPE()` operations is assumed to be unknown, and `nexus_put_TYPE()` and `nexus_check_buffer_size()` will be used to fill in the buffer.

3.5.5 nexus_put_TYPE()

```
void nexus_put_TYPE(nexus_buffer_t *buffer,
                    TYPE *data,
                    int count)
```

Copy `count` data elements from address `data` to the message buffer referenced by `buffer`. Convert data to a machine-independent form if necessary. These operations do not check for buffer overflow; the compiler must either allocate a buffer of adequate size using `nexus_set_buffer_size` or generate calls to `nexus_check_buffer_size`.

Valid TYPE values include float, double, short, u_short, int, u_int, long, u_long, char, u_char, and global_pointer.

Notes It is not clear how 64-bit integers will be handled. Are these longs? Are they long longs?

3.5.6 nexus_check_buffer_size()

```
int nexus_check_buffer_size(nexus_buffer_t *buffer,
                            size_t slack,
                            int increment)
```

Check that the message `buffer` has at least `slack` bytes remaining. If no resizing is necessary, leave `buffer` unchanged and return non-zero. If resizing is necessary, increase the size by

increment byte increments until the buffer has at least slack bytes remaining. If resizing is successful, modify buffer to a new, larger buffer and return non-zero. Otherwise, if increment is equal to zero and slack bytes are not available in the buffer, then leave buffer unchanged and return zero. Note that buffer must be a buffer structure previously allocated by a call to nexus_init_remote_service_request.

The slack argument should be calculated by using nexus_sizeof_TYPE().

3.5.7 nexus_send_remote_service_request()

```
void nexus_send_remote_service_request(
                    nexus_buffer_t *buffer)

void nexus_send_urgent_remote_service_request(
                    nexus_buffer_t *buffer)
```

Generate a remote service request message to the node and context referenced by the global pointer gp used in the nexus_init_remote_service_request() call used to create the buffer. After the message is generated, the buffer is freed.

Only two constraints are placed on when remote service requests are executed. First, an urgent remote service request executes within a bounded time (i.e. no indefinite postponement). Second, two remote service requests sent from one context to another are executed in order and in sequence. (Therefore, if two threads each send a remote service request to the same context, they can guarantee the order in which those requests will be handled by guaranteeing the order in which they are sent.)

3.5.8 nexus_get_TYPE()

```
void nexus_get_TYPE(nexus_buffer_t *buffer,
                    TYPE *dest,
                    int count)
```

Copy count elements from the message buffer to location dest. Convert data from a machine-independent form if necessary.

These routines can be used only within a handler. To save and access a buffer outside a handler, nexus_stash_buffer(), nexus_get_stashed_TYPE(), and nexus_free_stash_buffer() must be used.

See nexus_put_TYPE() for a list of the valid TYPE values.

3.5.9 nexus_stash_buffer()

```
void nexus_stash_buffer(nexus_buffer_t *buffer,
                        nexus_stashed_buffer_t *stashed_buffer)
```

When a remote service request is handled, the buffer that is passed to the handler is valid only for the duration of that handler and is automatically freed upon completion of the handler. It cannot, for example, be copied into a user data structure, to be accessed later from outside of the handler.

`nexus_stash_buffer()` saves the passed `buffer` so that it can be accessed later via the `nexus_get_stashed_TYPE()` calls, and then freed by `nexus_free_stashed_buffer()`. This stashed buffer is placed into `stashed_buffer`.

It is legal to stash a buffer from which some data has already be retrieved using `nexus_get_TYPE()` calls. Subsequent `nexus_get_stashed_TYPE()` calls will continue retrieving data where the `nexus_get_TYPE()` calls left off.

This mechanism means that an implementation of Nexus need not buffer data received in a remote service request, if that data can be used immediately. Yet it also allows selective buffering where it is advantageous to do so (for example, to enqueue a buffer for later processing). If a Nexus implementation is not buffering messages, this operation must allocate a buffer and stash the message in that buffer. But if a Nexus implementation is already buffering messages, this procedure can efficiently convert the buffer to a stashed buffer without doing an extra copy of the buffer contents.

3.5.10 nexus_get_stashed_TYPE()

```
void nexus_get_stashed_TYPE(nexus_stashed_buffer_t *stashed_buffer,
                            TYPE *dest,
                            int count)
```

Same as nexus_get_TYPE but obtains values from a stashed buffer.

3.5.11 nexus_free_stashed_buffer()

```
void nexus_free_stashed_buffer(nexus_stashed_buffer_t *stashed_buffer)
```

Free the `stashed_buffer` that was created by `nexus_stash_buffer()`.

3.5.12 nexus_register_handlers()

```
typedef void (*nexus_non_threaded_handler_func_t)(
```

```
                                    void *address,
                                    nexus_buffer_t *buffer);
typedef void (*nexus_threaded_handler_func_t)(
                                    void *address,
                                    nexus_stashed_buffer_t *buffer);
typedef enum _nexus_handler_type_t
{
    NEXUS_HANDLER_TYPE_THREADED,
    NEXUS_HANDLER_TYPE_NON_THREADED
} nexus_handler_type_t;

typedef struct _nexus_handler_t {
    char *                  name;
    int                     id;
    nexus_handler_type_t    type;
    nexus_handler_func_t    func; /* nexus_non_threaded_handler_func_t */
                                  /* or nexus_threaded_handler_func_t  */
} nexus_handler_t;

void nexus_register_handlers(nexus_handler_t *handlers)
```

A handler is a function that is executed in response to a remote service request. A threaded handler executes in a thread created specificly for it, while a non-threaded handler executes in an existing thread. The arguments to a handler are the address portion of a global pointer and a pointer to a Nexus buffer. A handler must be registered by a `nexus_register_handlers()` call before it can be invoked. Handlers are local to a context.

The `handlers` variable is an array of `nexus_handler_t` structures, terminated by an element with a `NULL` `func` field. This call associates a handler `name` and `id` with a function, `func`. The `type` specifies whether the handler is threaded or non-threaded. A handler name must be unique within a context. The handler id and function names need not be unique.

A handler is run in the context specified by the remote service request and has as arguments the local address specified by the global pointer and a buffer. A non-threaded handler takes a `nexus_buffer_t` buffer, which is automatically freed on exit from the handler. A threaded handler takes a `nexus_stashed_buffer_t` buffer, which must be freed explicitly by calling `nexus_free_stashed_buffer()`.

In the current implementation, the handler id must be the hash value for the handler name, as returned by `nexus_handler_hash()` (or the equivalent value generated at compile time). The problem addressed here is that of a global name space. Some sort of global name space is needed for handlers: one must know the "name" of a handler in order to invoke it by a remote service request. The string name of the handler function is a natural choice since this information is already available to the compiler. Unfortunately, this means that every

remote service request handler invocation would require a string to function pointer lookup, which would be fairly expensive. A more efficient global name for a handler would a unique integer; however, this is much more difficult to implement without a special link step. In the current implementation, our compromise is to use the string name of the handler as the global name, but also to pass the hash value for that name along with it. And as long as the hash function is well known, the hash value for a name can be generated at compile time. This should greatly speed the name to pointer lookup, with only a small increase in communication cost and little additional complexity beyond using just the string.

3.5.13 nexus_substitute_handler()

```
void nexus_substitute_handler(
                char *name,
                int id,
                nexus_handler_type_t new_type,
                nexus_handler_func_t new_func,
                nexus_handler_type_t *old_type,
                nexus_handler_func_t *old_func)
```

Replace the handler function for the handler designated by the `name` and `id` with `new_func` which is of the specified `new_type`. Return a pointer to the current handler function in `old_func`, and its type in `old_type`. If there is no handler is registered for the designated `name` and `id`, then register this new handler and return `NULL` in `old_handler_func`.

In the current implementation, `id` should be the hash value for `name`, as returned by `nexus_handler_hash()` (or the equivalent value generated at compile time).

3.5.14 nexus_handler_hash()

```
int nexus_handler_hash(char *name)
```

Return the hash value for the passed handler `name`.

The hash value is the sum of the ASCII values for the characters in `name`, modulo 1021.

3.5.15 nexus_poll()

```
void nexus_poll()
```

Handle any outstanding remote service requests to the context from which this function is called. If none are outstanding, then return immediately.

This function is useful only in a Nexus implementation that does not automatically handle remote service requests asynchronously as they arrive. Two such cases are a single threaded implementation (§ 3.9.2), and a non-preemptive thread implementation. In an implementation that does handle remote service requests asynchronously (a preemptive multi-threaded implementation), `nexus_poll()` does nothing.

3.6 Global Pointer Manipulation

3.6.1 `nexus_global_pointer()`

```
void nexus_global_pointer(nexus_global_pointer_t *new_gp,
                          void *address)
```

Place a global pointer that references the supplied `address` into `new_gp`.

3.6.2 `nexus_convert_global_pointer_address()`

```
void *nexus_convert_global_pointer_address(nexus_global_pointer_t *gp)
```

Return the local address for the specified global pointer. This operation assumes that the node and context of the global pointer are the same as the current node and context. If not, the results are unpredictable.

3.6.3 `nexus_destroy_global_pointer()`

```
void nexus_destroy_global_pointer(nexus_global_pointer_t *gp)
```

Destroy the specified global pointer, freeing its associated resources. The result of calling any Nexus function with a global pointer that has been destroyed (or with a copy of a destroyed global pointer) is undefined.

Any global pointers that are not explicitly destroyed using this function will be destroyed when the context in which they reside is destroyed.

3.6.4 `nexus_null_global_pointer()`

```
void nexus_null_global_pointer(nexus_global_pointer_t *gp)
```

Place a `NULL` global pointer into `gp`.

3.6.5 nexus_is_null_global_pointer()

```
int nexus_is_null_global_pointer(nexus_global_pointer_t *gp)
```

Return non-zero if gp is a NULL global pointer, otherwise return zero.

3.7 Inquiry Functions

3.7.1 nexus_node_type()

```
nexus_arch_type_t nexus_node_type(nexus_global_pointer_t *gp)
```

Return the architecture identifier for node pointed to by the global pointer. Each different type of machine has a unique architecture identifier.

3.7.2 nexus_node_class()

```
nexus_class_type_t nexus_node_class(nexus_global_pointer_t *gp)
```

Return the class identifier for the node pointed to by the global pointer. Each node type falls within a node class. All nodes within a node class can communicate with each other without performing any data conversion (i.e., they have the same byte ordering, word length, and floating point representation).

3.7.3 nexus_same_context()

```
int nexus_same_context(nexus_global_pointer_t *gp1,
                       nexus_global_pointer_t *gp2)
```

Return non-zero if the two global pointers point to the same context on the same node, or zero otherwise.

3.7.4 nexus_same_global_pointer()

```
int nexus_same_global_pointer(nexus_global_pointer_t *gp1,
                              nexus_global_pointer_t *gp2)
```

Return non-zero if the two global pointers point to the same address in the same context on the same node, or zero otherwise.

3.7.5 nexus_global_pointer_to_current_context()

```
int nexus_global_pointer_to_current_context(
                                nexus_global_pointer_t *gp)
```

Return non-zero if the global pointer points to an address in the context of the calling thread.

3.8 Miscellaneous

3.8.1 nexus_master_node()

```
int nexus_master_node()
```

Return non-zero if the current node is the master node, or zero otherwise.

3.8.2 nexus_exit()

```
void nexus_exit(int rc,
                int shutdown)
```

Terminate the computation with a return code of `rc`. All threads, nodes, and contexts remaining in the computation are terminated. It is guaranteed that `NexusExit()` will be called on all contexts.

If `shutdown` is not 0, then `nexus_shutdown()` is called by `nexus_exit()`. If `shutdown` is 0, then `nexus_shutdown()` is not called by `nexus_exit()`. Instead, it must be called from the user program after the exit.

3.8.3 nexus_shutdown()

```
void nexus_shutdown()
```

Shut down Nexus. This function is called automatically by `nexus_exit()` if the `shutdown` argument to `nexus_exit()` is not 0. In other situations, the user must call `nexus_shutdown()` explicitly after calling `nexus_exit()`. For example, the last global destructor in CC++, which executes after `nexus_exit()`, calls `nexus_shutdown()`.

3.8.4 nexus_abort()

```
void nexus_abort()
```

Terminate the computation. All threads, nodes, and contexts remaining in the computation are terminated. `NexusExit()` may not be called on all contexts.

3.9 Defined Symbols

Several symbols are beneficial to an application that uses Nexus.

3.9.1 NEXUS_NON_PREEMPTIVE_THREADS

NEXUS_NON_PREEMPTIVE_THREADS is defined in `nexus.h` if the thread module being used by Nexus is non-preemptive.

On such a system, `nexus_poll()` or any of the other Nexus communication routines must be called sufficiently often to handler outstanding remote service requests.

3.9.2 NEXUS_SINGLE_THREADED

NEXUS_SINGLE_THREADED is defined in `nexus.h` if the thread module being used by Nexus does not support multiple threads.

Much to our dismay, some machines do not support multithreading. In order to support these machines, Nexus has been defined in such a way that a subset of the full Nexus functionality can still work in this single-threaded environment:

- `nexus_poll()` will handle any outstanding remote service requests. It is assumed that the application will call `nexus_poll()` sufficiently often for this to be effective.
- Only a single thread can exist at a time. When a new context is created, there are no threads executing in that context. A new thread can be created by using `nexus_create_thread()` from a handler in the context. If an additional call is made to `nexus_create_thread()` before the previous thread terminates (`nexus_terminate_current_thre` this will generate a fatal error and cause the computation to abort (`nexus_abort()`).

NEXUS_SINGLE_THREADED allows the application using Nexus to adapt its behavior at compile time for this special environment.

3.9.3 NEXUS_USE_MACROS

By default, all Nexus calls are function calls. This simplifies debugging and minimizes code size. Many Nexus routines, however, may map trivially to underlying system routines. (For example, the Nexus thread routines map to Posix thread routines.) In this case, it may be more efficient to implement some Nexus functions as C macros which invoke the underlying system routines.

If NEXUS_USE_MACROS is defined before `nexus.h` is included in an application, then C macros are used where appropriate to avoid function call overheads for some Nexus routines.

4 Package-Supplied Functions

A package using Nexus must provide a small set of functions for use by Nexus:

- `NexusBoot`: a function that is invoked upon creation of the context (required)
- `NexusExit`: a function that is invoked upon termination of the context (optional)
- `NexusUnknownHandler`: a function that is invoked if a remote service request is made to this context with an unknown handler (optional)

4.1 NexusBoot()

```
int NexusBoot()
```

`NexusBoot()` is automatically invoked when a context is created (see § 3.3.1). If may be used, for example, to register handlers required by the context.

If `NexusBoot()` returns a non-zero value, then the creation of this context will fail (see § 3.3.1), and `nexus_create_context()` will use this value for its return code. If `NexusBoot()` returns zero, then context initialization will complete normally. It is recommended that `NexusBoot()` return a positive value to indicate failure, since negative values are returned by `nexus_create_context()` when context creation fails due to resource limitations.

4.2 NexusExit()

```
void NexusExit()
```

If there is a handler registered using the name `NexusExit`, that function will be called immediately before a context is terminated (see § 3.3.4). This must be a non-threaded handler.

4.3 NexusUnknownHandler()

```
void NexusUnknownHandler(void *address,
                         nexus_buffer_t *buffer,
                         char *handler_name,
                         int handler_id)
```

If there is a handler registered using the name `NexusUnknownHandler`, that function will be invoked in a context when a remote service request is made to the context using an unknown

handler. The `address` and `buffer` are the same as those of a normal handler: the global pointer's local address and the buffer used in the remote service request. The `handler_name` and `handler_id` arguments are those of the unknown handler specified in the remote service request. This must be a non-threaded handler.