QNial Extensions

John Gibbons

01 August 2015

Extension to Core Nial

This section describes a number of extensions to the standard version of Q'Nial. The extensions consist of C-coded primitives to provide the basic capabilities together with Q'Nial libraries containing wrapper functions and transformers to provide patterns of usage of the primitives.

These features can be switched on or off using the package manager if desired.

The features consists of the following

1. Associative Arrays or Hash Tables
2. Child Process creation and management
3. Byte Streams for Inter-Process communication and data encoding/decoding
4. Support for Shared Memory
5. Dynamic loading of primitives

Each of these is described in the following sections along with examples of their use.

Associative Arrays

Associative arrays are an extension to Q'Nial implemented as hash tables. The keys can be either Q'Nial phrases, character arrays or integers. They are implemented as pure Q'Nial data structures but the primitives are coded primarily in C for efficiency.

Because the tables are pure Q'Nial data structures they can be saved in a workspace or serialised and sent between processes. However, it is not advisable to write or display them on the console as they are not intended to be human readable. This will be changed in a later version of Q'Nial.

For historical reasons there are two sets of functions, one that treats them as hash tables directly and one that views them as associative arrays with functions similar to the standard array manipulation functions of Q'Nial.

Implementation

As implemented, a hash table is an array of arrays. At the top level there are 5 entries

1. a standard phrase to identify a hash table
2. an array of keys
3. an array of values (of the same size as keys)
4. some counters (number of entries, probe count, deleted count)
5. a metadata slot for programmer use

The code uses linear hashing with rehashing to handle collisions. The size of a table is always a power of two. Rehashing uses a table of large primes to avoid collisions as much as possible.

A table is automatically expanded if it becomes more than 70% full or the deleted count exceeds a nominated percentage.

Two sets of routines are provided, a set of primitives and a set of Nial coded routines that mimic the routines of normal Q'Nial arrays.

Primitives

The primitives are for the most part implemented in C with some implemented in Q'Nial where performance is not an issue.

Note that in all of these routines an invalid key (one that is not a phrase, character array or integer) will cause an invalid key fault to be returned.

Basic Routines

These routines are coded in C and will validate the table and where necessary the type of the *key* supplied to the primitive. As stated above a key must be either a phrase, and integer or a character array.

__tcreate count Create a new table with initial size *count* and return it. If count is not a power of 2 the table size will be the smallest power of two greater than count (with a minimum size of 32).

__tset table [key, value] Add a *key/value* pair to the table or update an existing *key*. If the resulting table would be more than 70% full the table will be rehashed. This routine also sets the number of probes (hashes and rehashes) used to find a suitable slot to add the entry. This is mainly used for peformance analysis.

table __tget key, {default} Retrieve a value from the table using the supplied key. If no entry has this *key* and a *default-value* is supplied then it will be returned. Otherwise a Q'Nial fault will be thrown.

istable table Test for a table, returning either *True* or *False*. This test uses the standard phrase as a way of identifying tables.

__tsetm table meta-data Set the *meta-data* value for the table. This can be any Q'Nial value and the *meta-data* field can be used for any purpose. The idea is taken from *Lua* where it is used to implement prototype-instance inheritance.

__tgetm table Get the metadata value of the table.

__tdel table key Remove the key/value pair corresponding to *key* from table. The code remembers the deleted key to allow the empty slot to be re-used by the same key at a later stage. The function returns 1 if the key was found or a fault (*?tdel_args*) if not found.

__getkeys table Return the collection of keys as an array.

Nial Coded Routines

The following routines are coded in Nial and extend the functionality of the primitives.

tCount T Return the number of entries in the table

tsize T Return the current capacity of the hash table. Note that tables will automatically resize so this does not indicate an upper limit.

tnew KeyValuePairs Create a new hash table from a list of key/value pairs

Associative Arrays

These routines are implemented on top of the basic primitives described above and are designed to mimic the standard Q'Nial array functions. They will not return a table to be printed.

aupdate AA [Key, Val] Add the key value pair to the associative array

aupdateall AA KeyValPairs Add a list of key value pairs to the associative array

apick Key AA Return the value asociated with the supplied key

achoose Keys AA Return the array of values associated with the supplied array of keys

atell AA Return the array of keys of the associative array

apickall AA Return the array of all key/value pairs of the associative array

aremove IS OP AA Key Remove an entry from the associative array, returning a boolean value to indicate whether or not the key was found in the array.

atally AA Return the size of the array

acapacity AA Return the current capacity of the associative array. This is not an upper limit as associative arrays will automatically resize to accomodate new entries.

acreate Name KeyValpairs Create a new associative array from the list of key/value pairs and assign it to the named variable

aequal AA BB Determine if two associative array have the same sets of key/value pairs

Process Creation and Management

These primitives provide the capability for a Nial process to create, manage and remove child processes.

> **NOTE** The file *sprocess.ndf* in **niallib** contains a number of supporting definitions and the *Examples* folder of the distribution contains some examples of usage.

Termination of a child processes is handled by the SIGCHLD handler and a child can either be managed or unmanaged. By default a process is managed.

A managed child is one where the parent process wishes to be notified upon termination so that the parent code can perform some post processing. A typical example would be a child spawned to perform a computation in parallel.

An unmanaged child is one where the parent process has no interest in being notified of the termination of the child and the childs resources can be automatically reused. In this case the re-use happens when creating new processes or closing existing processes.

A simple example of an unmanaged process would be to create a plot while the parent continues its processing.

Child processes can also be clones of the parent inheriting its workspace with all data and functions, or run a completely separate program.

When combined with *sockets* and *byte streams* it is possible to create a cluster of processes that spread a Q'Nial computation across either a single multi-core system, multiple machines or a combination of both using Nial arrays as messages.

Two basic approaches are supported:

1. Loose coupling of processes with communication via pipes or sockets in which the processes can be on the same machine or spread across a cloud
2. Tight coupling of processes with communication via shared memory

The approaches can also be mixed in a system implementation.

Internally within the library a table of child processes is maintained. Each table entry maintains information about the child.

spawn_child flags Fork a child process with communication over pipes as the standard input and output of the child process. The child is a *non-interactive* clone of the parent. The return value on success in an internal index of the child process, otherwise a fault is returned. The *flags* value is 1 for an unmanaged child and 0 otherwise.

spawn_shell flags This primitive creates a child process running a bash shell and a pseudo-terminal as its input and output. Any external process can be run in this environment. The parent can initiate execution of a command by writing to the input stream associated with the process. Similarly it can directly read the output of the process. The returned value and flags are identical to *spawn_child*.

spawn_cmd progname args flags This function creates a child process running the nominated command. The input and output streams of the child are pipes so some commands that need an interactive terminal will not run in this environment. On the other hand, pseudo terminals are a limited resource.

child_writer child This returns a stream for writing to the standard input of the child

child_reader child This returns a stream for reading form the standard output of the child

interrupt_child child signal Send a signal to the child process. The signal value can either be 0 for a kill signal or non-zero for an interrupt.

child_status child The *child* index here can either be -1, to match any managed terminated child or the index of an existing child. The function returns a triple of the child id, its current status and its termination code. The current status of the child is 1 for an active child and 2 for a terminated child.

sys_exit code Terminate this process and return the *code* value as the exit code.

nano_time This function provides a high precision timer for determining the performance of code. It returns a real value with nanosecond precision. It cannot be used as a wall clock.

nano_sleep seconds nano-seconds High precision sleep function for the process.

Byte Streams

Nial streams are an extensible byte buffering mechanism that can be used for reading and writing of data to files, sockets, pipes etc as well as internal encoding and decoding of Q'Nial data structures.

> **NOTE** The file *nstreams.ndf* in **niallib** contains supporting definitions and the *Examples* directory of the distribution has some example code.

A stream is referenced in the application by a integer index into a table of stream data structures. This index is supplied when opening a stream.

The module provides a number of primitives for opening and closing streams, connecting them to file descriptors, writing and reading and for encoding and decoding Q'Nial arrays.

Streams provide a boundary between application logic and the needs of networking and file systems. On input the data from a file or network connection is transferred to the streams buffers and then read by the application into Nial arrays. On output data is transferred from Nial data structures into the buffers and then subsequently written.

This approach allows streams to be used in either a blocking or non-blocking style depending on the applications requirements. This is accomplished by using a polling approach in a number of routines with a supplied timeout. This timeout value can either be a positive integer (>= 0) indicating microseconds or a value of -1 indicating an indefinite timeout.

Streams can also be used as a purely internal, byte based, data structure disconected from file descriptors.

The primitives can be divided into 5 categories

1. Creating/Deleting Streams
2. Basic I/O
3. Serialising/Deserialising Arrays
4. Polling I/O
5. Opening and closing, pipes, socket pairs etc.

Creating Streams

The following routines are used to allocate and free internal data structures and buffers. They do not involve any data transfer.

nio_open fd mode Create a stream and assign it to a file descriptor fd (-1 if no descriptor) with the nominated mode. At the moment the mode value is unused in the runtime

nio_close stream Close any associated file descriptor and release all memory buffers associated with the stream

Basic Operations

The basic operations are concerned with moving data backwards and forwards under programmer control or controlling the relationship between streams and file descriptors.

They are as follows:

nio_count stream Returns the number of characters buffered on the stream at this moment. On an input stream this will not attempt to transfer any data from the network or file system.

nio_read_stream stream climit timeout This routine will attempt to ensure that the internal buffers contain at least *climit* bytes. The call will repeatedly poll for input from the associated descriptor if needed with a timeout value of *timeout* until either there are *climit* bytes buffered or an end of file condition is reached. The function returns the number of bytes buffered.

nio_read stream num-bytes This function will attempt to read *num-bytes* from the stream. If the stream already has *num-bytes* buffered it will return those without reference to any associated file descriptor. If there are less than *num-bytes* buffered and the stream has an open file descriptor it will attempt a single read from the descriptor without waiting for the descriptor to become ready. It will then return up to *num-bytes* or *Null* if the stream is empty.

nio_readln stream Read a properly terminated line of characters from the stream. including the line terminators. If no line can be found then it returns null. The call polls for additional input before searching for a line.

nio_write_stream stream timeout While the associated file descriptor is writeable, write as many bytes as possible. This will return the count written or a fault if an error occured (e.g closed descriptor).

nio_write stream char-array Add the bytes in the supplied character array to the stream and return the count written. This involves no transfer to an external file descriptor.

nio_writeln stream char-array Add the character array to the stream and then add a line terminator sequence. Returns the number of characters written.

nio_flush stream Using a blocking model, write the contents of the stream's buffers to the associated file descriptor and return the number written. This will only fail to write all bytes if the descriptor is closed or the process is interrupted.

Serialising Arrays

A Q'Nial array that is not self-referential can be serialised to a stream

nio_block_array stream array Encode an array on the nominated stream and return the number of characters used in the encoding. No transfer of data to an associated file descriptor is performed.

nio_unblock_array stream Decode and return an array from the nominated stream. At the moment this is a blocking operation. If an error occurs, *Null* is returned.

Polling Streams

These primitives provide a link to the underlying OS polling mechanisms using the *select* system call (the lowest common denominator on Linux and OSX). This approach has a couple of disadvantages based on limitations of the *select* system call.

Firstly the call is limited to around 1024 file descriptors although for most Nial programs this would not be a problem. Secondly if any descriptor in a call has a problem the entire call fails without identifying the source of the problem.

Two approaches are provided, check if a single stream is readable/writeable or check for I/O on an array of streams. The single stream approach can emulate

the multiple stream approach using a Nial transformer (*EACH* etc) although the approach will be slower as the timeout becomes cumulative.

The calls use a *timeout* value to determine how long the call should wait for availability of the descriptor. This can be zero for no wait, a positive integer value in microseconds or -1 for an indefinite blocking wait.

The single stream primitives are:

nio_is_readable stream timeout Check to see if the *stream* is readable waiting for the timeout value. If an invalid stream is supplied the call will return a fault (*?invalid_stream*) otherwise it will return 1 if the stream has data, 0 if the stream has no data or -1 if the stream is at end of file or closed or has an error.

nio_is_writeable stream timeout Check to see if the *stream* is writeable, waiting for the timeout value. If an invalid stream is supplied the call will return a fault (*?invalid_stream*) otherwise it will return 1 if the stream can accept data, 0 if the stream will not accept data or -1 if the stream is at end of file, or closed or has an error.

And the multiple stream primitive is :

nio_poll timeout read-list write-list exception-list This directly maps to the *select* system call. Each of the lists is a list of streams and the function checks to see if any of their descriptors are readable, writeable or has an exception.This returns three boolean arrays corresponding to the streams which indicates if the associated stream is readable, writable or has an exception. The timeout is -1 for indefinite blocking or a number of microseconds. If an error occurs the call returns *Null*.

Socket Pairs and Pipes

nio_socketpair flags Create a pair of UNIX Stream sockets sockets that are endpoints of a single internal connection. The flag is an integer value that controls whether or not the sockets will be blocking (non-zero) or non-blocking (zero). The call returns a pair of file descriptors or the fault *?syserr*. The file descriptors can then be associated with a stream.

nio_newpipe flags Create a pipe and return a pair of the file descriptors. The flag controls blocking or non-blocking behaviour.

Shared Memory

In its current form Q'Nial is not multi-threaded and the shared memory primitives are an addition to provide similar capabilities along with the child process approach described earlier.

> **NOTE** The file *memspaces.ndf* in **niallib** contains supporting definitions and the *Examples* directory of the distribution has some example code.

The primitives are divided into

1. Creating and mapping shared memory and files
2. Coordinating processes interacting through shared memory
3. Copying data in/out of shared memory
4. Support routines

The primitives are intended to provide the base layer for higher layer experimentation (e.g. Software Transactional Memory).

A Nial process can access multiple shared memory segments with each segment identified by an integer index into an internal table. The segments can correspond to named shared memory, locally allocated shared memory or to memory mapped files.

Data is copied to/from shared memory into Nial data types and only a limited number of types are supported, boolean, integer, character and real lists. Each type is identified by an integer code. More complex types can be shared by first serialising them and then sharing the serialised form.

When copying in from shared memory the primitive generates the Nial array to avoid memory corruption.

The supported types are identified by an integer code as show in the following table and also defined in the **niallib** file **memspaces.ndf**.

Type	Code	Name
Integers	1	MSP_INTTYPE
Booleans	2	MSP_BOOLTYPE
Characters	3	MSP_CHARTYPE
Reals	4	MSP_REALTYPE

Creating and Mapping Shared Memory

These primitives provide for creatng and mapping shared memory segments as well as memory mapping files.

msp_shm_open name open-code permissions Create a POSIX named shared memory space and return a file descriptor which can then be used to memory map the space. The *open-code* is a bit mask value which maps to the *shm_open* values *O_RDONLY* (1), *O_CREAT* (2) and *O_EXCL* (4). If *O_RDONLY* is not specified then the default *O_RDWR* will be used. The *permissions* value is the standard open mode.

msp_shm_unlink name Unlink a shared memory space.

msp_map_local size flags Create a local memory space. If the *flags* value is non zero the created space is private, otherwise shared amongst the children of this process. Returns the index of the space.

msp_map_fd size flags fd Memory map a file based on the descriptor. The map can be a part of the file ($size > 0$) or the whole file ($size = 0$). The flags are a bit mask defining the access mode, private (1), read only (2). The default is shared with read/write.

Process Coordination

These primitives provide low level synchronisation using atomic operations.

msp_cas mem-space offset compare-value replacement-value This provides an atomic compare and swap of integer values. The offset must be aligned on a word boundary (32 bit or 64 bit depending on the configured version). The primitive returns true if the operation succeeded, otherwise false.

Copy In/Copy Out

A small number of routines are provided to move data between shared memory and the Nial process.

Note these are not atomic operations.

msg_get_raw mem-space nial-type memory-offset count Copy data from a shared memory segment (*mem-space*) into a newly created Nial array of dimension 1 (this can be reshaped within Nial). The memory offset is from the start of the memory segment and the *count* is the number of entries of the nominated type to copy.

msp_put_raw mem-space nial-data memory-offset count Copy data from a Nial array into a shared memory segment. The *count* is the number of entries from the Nial array.

Support Routines

These are simple utility routines.

Q'Nial organises memory on word boundaries (32 bits or 64 bits depending on the configured choice) and provides alignment. The support routines provide a way to determine the amount of shared memory needed to support one of the supported Nial types.

msp_msize nial-type count Returns the number of bytes associated with a given type and count. Both arguments are integers with the type being one of *MSP_BOOLTYPE*, *MSP_INTTYPE*, *MSP_REALTYPE* or *MSP_CHARTYPE*.

msp_sysconf code Return some information about the processor on which Nial is running. If *code* is 0 then return the number of configured cores, if *code* if 1 then return the number of available cores.

Dynamic Loading

Dynamic loading provides an alternative to reconfiguring Q'Nial when one wishes to add additional primitives.

> **NOTE** A number of examples are provided in the *Modules* directory of the distribution which demonstrate how to code the primitives as well as how to build the shared object. A CMake example configuration is provided as well as example *NDF* files to load the primitives and simplify the calls using currying.

Dynamically loaded routines can not be saved in a workspace and loaded at a later time. The shared libraries will not be restored as part of a workspace load.

The routines described here are intended to form the base layer. As such they are direct wrappers of system library functions.

The dynamically loaded functions are standard Nial coded functions conforming to Nial coding conventions.

The external routines in each dynamic library are not merged when the library is loaded. It is possible for two different libraries to have primitives with the same name.

ndl_load file-name Load the shared object library contained in the nominated file. The externals of the library are not added to the set of Nial globals. At the moment the *file-name* parameter must be either an absolute file name of a relative path to the library. This returns an internal data structure which should not be changed.

ndl_close lib Close a shared library. After this call no function from the library will be usable.

ndl_getsym lib fun-name Return a function pointer data structure that can be used to call the function identified by *fun-name* from the shared object *lib*.

ndl_call fun-ptr args Call the function identified by *fun-ptr* with the arguments *args*. This returns the results of the function call.