Programming in the ATHOMUX Environment

Thomas Schöbel-Theuer

Version 0.31, 10 December 2004

Contents

1 Foreword

To understand this document, you should have read some basic papers on the architecture of ATHOMUX, and you should be familiar with C programming. Currently the description is very brief; you can help yourself by reading example code. A lot of stuff is missing; this document may soon be outdated.

Before reading this document, first read the Preprocessor User's Guide.

2 Basic Data Types

For implementation of ordinary bricks, please don't `#include` anything in the header section! If you have to share some *internal* data structures (and only if those structures are really truly *internal!*) with other brick implementations, do so by including a common header in the *implementation* preamble.

Please use only the following portable basic data types, as defined in `common.h`:

`addr_t` An address in the logical address space of a nest. It's probably larger than the size of a pointer of the target architecture, i.e. currently 64Bits for both i386 and x68_64. However keep in mind that when ATHOMUX is ported to a PDP-11 or to another 16Bit platform (e.g. in the area of control applications), this type may be redefined to 32Bits or even 16Bits. Don't rely on a particular size of this type, but keep your code universally generic ;-)

`len_t` A length or unsigned (positive) offset in the logical address space. It is guaranteed to have the same size as `addr_t`, so you can typecast between them.

`off_t` A signed offset in the logical address space. Guaranteed to have the same size as `addr_t` and `len_t`.

`paddr_t` An address in the physical address space. It has the same size as a pointer. Nevertheless, when you have to access physical memory blocks, *always* use the macro `MAKE_PTR()` to convert `paddr_t` to an ordinary `void*`. In some physical models, offsets may be transparently added by that macro. Thus never type-cast by hand!

`plen_t` A length or unsigned (positive) offset in the physical address space. It is guaranteed to have the same size as `paddr_t`, so you can typecast between them.

`poff_t` A signed offset in the physical address space. Guaranteed to have the same size as `paddr_t` and `plen_t`.

`index_t` A signed integer capable of indexing small arrays; don't use this in portable data structures.

`mand_t` Holds a mandate. A mandate designates a partificiant at locking. In generral, a brick instance may act on behalf of many mandates.

`bool` An enum with values `FALSE` and `TRUE`, with obvious meaning.

`success_t` Currently an alias for `bool`, for consistent usage as return status of operations. Later, this may be replaced by another definition with further values than `TRUE` and `FALSE`, e.g. for discriminating fatal versus non-fatal errors. Thus declare your own status variables always with `success_t`, never with `bool`!

`direction_t` Enum for the `$trans` operation, has values `direct_read`, `direct_write`, and `direct_stop`.

`prio_t` Enum for the `$trans` operation, has values `prio_background`, `prio_normal`, `prio_urgent`.

`version_t` Enum for telling the actuality status of a memory region. Has values `vers_undef`, `vers_old`, and `vers_newest`.

`lock_t` Enum determining the type of `lock` or `unlock` operations. Has values `lock_none`, `lock_read`, `lock_write`.

`action_t` Enum determining the behaviour of `wait`. Has values `action_ask`, `action_try`, `action_wait`.

`name_t` Character string, currently of length 24, to hold names of bricks and inputs/outputs. TODO: change this to length 16 (i.e. 2*sizeof(addr_t)) and shorten some already established brick names. Shorter strings may be kept in `sname_t`, longer ones in `lname_t`.

In device drivers, you may use the following types for specifying hardware-dependent data layouts. *Never* use them in ordinary ATHOMUX bricks, because these will not scale with the architectural model!

`uns1` and `int1`: one unsigned or signed byte.

`uns2` and `int2`: an unsigned or signed 16bit-halfword.

`uns4` and `int4`: an unsigned or signed 32bit-word.

`uns8` and `int8`: an unsigned or signed 64bit-longword.

Further advice: be careful when using standard types `int`, `unsigned`, `short`, `long`, `long long` and so on. Their size is not constant, but may vary with the used compiler. It may even vary independently from the ATHOMUX architecture model. Use them only in system-specific bricks, but *never* in ordinary bricks. In particular, *never* use them for data formats which could be read by other brick instance on a *heterogenous* cluster in a network! These data types are non-portable, so please avoid them as much as possible. TODO: check their usage by the preprocessor and issue warnings.

TODO: discriminate between LSB and MSB types, and add macros for their conversion. Ensure that *migratable* data formats (in difference to *internal* data formats) are solely built upon exact specs of their byte sex! Integrate with meta-nest data descriptions, in order to allow automated sex conversions by adaptor bricks.

3 The Nest Interface

The syntax is described by an enrichment of the preprocessor syntax with data type specs. When an argument is followed by `:= something`, you may omit that parameter at a call, and `something` is automatically used as the default value for that parameter.

All operations have implicit parameters `@args` of type `struct args *` and `@param` of type `char *`. All further standard parameters are all fields of `struct args` (see definition in `common.h`). Writing `@param_name` is equivalent to `@args->param_name` (see preprocessor guide). Via `@param`, a generic NULL-terminated string can be passed which must not contain linefeeds. Its interpretation is up to the implementation. Currently it is used for passing of filenames and pathnames, but this is provisionary. DISCUSS: introduce a universally generic, but nevertheless easy parsable URL syntax.

All operations have at least one return value `success_t success`. When the operation is called, `@success` is guaranteed to be `FALSE`. When your operation fails (due to some reason), you may just simply `return` from it without changing `@success` at all. Conversely, if your operation was successful, do not forget to set `@success = TRUE`. When you reuse the `@args` parameter by low-level passing via short-form syntax (see preprocessor guide), you have to ensure that the callee will find `@success == FALSE`, i.e. don't forget to clear it before such an efficient shortcut call whenever it *could* have been set in the meantime (e.g. by a prior shortcut call).

3.1 Operations Defined on Outputs

Normally, the operation control flow goes from inputs to outputs. Thus operations are implemented at outputs. Only in some rare cases (such as `$retract`, see section 3.2), the operation control flow goes in the opposite direction.

3.1.1 Initialization and Termination / Pseudo-Statelessness

Besides the ordinary initialization of a brick instance, an additional operation has been added which is called by the responsible `control_*` each time a dynamic output is instantiated, or when a pseudo-stateless instance is switched to a stateless state.

```
$output_init (bool destr, bool constr, bool clear := FALSE)
=> (success_t success)
```

When `destr==TRUE`, flush any state to the inputs and de-instantiate the output instance it is called on. When `constr==TRUE`, construct the output instance. When both parameters are set, you may omit the destruction followed by construction, but you have to flush any state (if some exists; otherwise you may decide to do nothing at all). When both are unset, just do nothing (but possibly some consistency checks).

When `clear==TRUE`, the *contents* of output nest should be initialized / cleared in some defined way, leading to a defined state.

NOTICE: sometimes it makes sense to `@=outputcall` this operation over wires (e.g. for recursively switching to a stateless state), but in most cases it will be called by the `control_*` level, in particular by the `$instconn` operation.

ATTENTION! prior to 30 August 2004, there was only a single `$init` instead of `$output_init`! Please update your sourcecode!

3.1.2 Physical IO

```
$trans (addr_t log_addr, len_t log_len, paddr_t phys_addr,
direction_t direction, prio_t prio := prio_normal) =>
(success_t success, plen_t phys_len)
```

Start or submit an IO request (`direct_read` or `direct_write`) or try to stop already started/submitted ones (`direct_stop`) if they exist. The logical address, length, and physical address are in `log_addr`, `log_len`, and `phys_addr` respectively. The `prio` tells whether the IO request can wait arbitrarily (`prio_background`), or has normal priority (`prio_normal`) or is extremely urgent (`prio_urgent`). *Inside* each of these priority classes, starvation has to be avoided (e.g. by a disk scheduler), but *between* these classes starvation may occur, e.g. when the IO channel is overloaded.

Return value: upon `success==TRUE`, the actual length of the submitted request is returned in `phys_len`. It may be smaller than `log_len`, depending on the capabilities of the driver. The caller must check and handle that case (e.g. transfer the rest separately).

```
$wait (addr_t log_addr, len_t log_len, prio_t prio :=
prio_normal, action_t action := action_wait) => (success_t
success)
```

Test whether an uncompleted IO request of priority `prio` or higher exists in the logical address range specified by `log_addr` and `log_len`.

When `action == action_wait`, wait until at least all IO requests which were present at the start of the operation (i.e. have not arrived during the wait) have completed or are in an error state (e.g. IO error), and return error-freeness in `success`. Note that when no IO requests were present at all, no delay will occur. It is thus possible to issue `$wait` even on memory regions where never a `$trans` has been started. It need no be used pairwise with `$trans`. It just ensures that everything is in sync.

When `action == action_try`, return immediately telling whether no error has occured. As a side effect, ensure that new `$trans` requests arriving after that will never be serviced before the old ones have completed (transaction barrier).

When `action == action_ask`, return immediately telling whether everything is currently in sync (status poll).

3.1.3 Logical IO

```
$get (addr_t log_addr, len_t log_len, bool forwrite :=
FALSE) => (success_t success, paddr_t phys_addr, plen_t
phys_len, version_t version)
```

Try to allocate a physicial buffer for the logical memory area specified by `log_addr` and `log_len`. When `forwrite == FALSE`, the buffer must not be modified afterwards; in particular it may be concurrently used by many readers without locking. Otherwise the data may be modified.

Upon successful return, the physical address and length is returned in `phys_addr` and `phys_len`. When `version == vers_newest`, you need not issue a following `$trans` to update the buffer. Otherwise, you cannot be sure that the buffer contents is up to date. However, if you will overwrite the buffer anyway, you need not (and in fact should not) issue an unnecessary IO operation for updating it.

```
$put (addr_t log_addr, len_t log_len, prio_t prio :=
prio_none) => (success_t success)
```

This operation *must* always be used pairwise with $get, because internal reference counting may be used to determine which buffers are currently in use and which not. You *must* supply the same `log_addr` and `log_len` as at the paired `$get`.

The `prio` parameter tells whether you have *actually* modified the data. If you have modified it, you *must* set it to at least `prio_background` or higher (it may be used in a later asynchronous writeback by a buffer cache implementation). Otherwise, you must set it to `prio_none`, even if you have *intended* a write at the former `$get`. This tells in effect whether you have dirtyfied the buffer or not.

Try to avoid unnecessary dirtifying of buffers wherever possible! This is crucial for performance!

3.1.4 Locking

```
$lock [mandate] (addr_t log_addr, len_t log_len, lock_t
data_lock := lock_write, lock_t addr_lock := lock_read,
addr_t try_addr := log_addr, len_t try_len := log_len,
action_t action := action_wait) => (success_t success,
addr_t try_addr, len_t try_len)
```

Issue a lock request und the mandate `mandate` with an obligatory part specified by `log_addr` and `log_len`, and an optional part specified by `try_addr` and `try_len`. The optional part must always be a superset of the obligatory part. When omitting the optional locking parameters, classical locking is requested, but optional locking may be transparently added behind the scenes by a local lock manager; see my papers on optional locking for understanding these concepts in more detail. The parameters `data_lock` and `addr_lock` tell whether and how to lock the data area resp. the address association of the nest (no lock / readlock / writelock). Locking the address association must be used to protect against concurrent `$move`, `$create` and `$delete` operations. Locking of the data area has the usual semantics (protection against concurrent data modifications). The parameter `action` requests either to wait until the lock can be granted or a deadlock has occurred, or to try to get the lock immediately without waiting, or just to ask whether locking would have been possible (but need not at the next attempt due to possible races).

When you try to lock some area once again under same `mandate` (regardless of the instance issuing that lock), you will not block yourself, but instead *enlarge* the locked area held under that mandate. This is called *merging* of locked regions. Thus you won't have to unlock your regions properly nested (i.e. the locks are *not recursive*, but rather *accumulating*).

When you specify a different `data_lock` or `addr_lock` type for an already held area, you will try to *upgrade* or *downgrade* the lock type. Note that upgrading will not work in general, since deadlocks may occur.

When the operation was successful, you can see the size of an optionally granted area by the results in `try_addr` and `try_len`. These may be smaller than originally requested, but never smaller than the obligatory part (see my papers on optional locking). Note that truly optional parts may be concurrently retracted at any time (i.e. racing arainst you), so please don't use that areas for critical data. However you may speculate that further obligatory locks in that area will *very likely* not block and thus show extremely good performance in a distributed system.

```
$unlock [mandate] (addr_t log_addr, len_t log_len,
addr_t try_addr := log_addr, len_t try_len := log_len) =>
(success_t success, addr_t try_addr, len_t try_len)
```

Ensure that the specified areas are no longer locked under the specified `mandate`. The parameters have the same meaning as above. You need not unlock in the same granularity as you have locked. For example, you may release all your locks in a single atomic action by specifying the whole logical address space (aka 2-phase-locking). You may also release only *parts* of a larger lock area, leading to *splits* of your locked regions. Exact pairing of `$lock` with `$unlock` is not necessary.

Notes on mandates: Always be sure to supply the right `mandate` parameter. Omitting that parameter in braces will insert the default value `@#_mand`, but that may be the wrong semantics. For example, when you forward a lock operation at a `dir_*` or `union` brick from some output to some input, you will normally also have to forward the mandate of the original requestor. Otherwise you may produce incorrect behaviour and even deadlocks, because the underlying lock manager will believe that all locks were originally requested from the same instance (namely your instance), which is wrong in logical sense. However, when your `dir_*` brick deals with directory status information for which it is solely *responsible*, you should issue lock requests for those areas under your own default mandate `@#_mand`. Otherwise you could allow too much incorrect parallelism.

3.1.5 Allocation in the Logical Address Space

`$gadr (len_t log_len, bool where := FALSE, bool exclu := TRUE, action_t action := action_wait, len_t try_len := log_len) => (success_t success, addr_t log_addr, len_t log_len)`

Atomically reserve space of minimum size `log_len` and maximum size `try_len`. When `where == FALSE`, reserve the space in a formerly undefined region (default for memory allocators and for writers on a pipe), otherwise reserve at defined regions (used for readers on a pipe). When `exclu == FALSE`, you get a valid area, but it is not atomically reserved for your exclusive use (concurrent lookahead without reservation). With `action != action_wait`, you can specify that failure of immediately obtaining the area will not hurt you existentially (e.g. useful for speculative memory preallocation strategies).

`$padr (addr_t log_addr, len_t log_len, bool where := FALSE) => (success_t success)`

Unreserve the space of a former `$gadr`; note that `where` must have the same value as in the former `$gadr`, regardless of an intermediate `$create` or `$delete`.

Note: `$gadr` is no longer a *pre*-reservation as in a former version of the nest interface, but it executes full reservations completely *independent* from `$create` / `$delete`. The reason is granularity: you may reserve a very huge part of the address space by `$gadr` and then `$create` only sparse parts of it (e.g. like in hash tables).

3.1.6 Dynamic Nests

`$create (addr_t log_addr, len_t log_len, bool clear := FALSE, bool melt := TRUE) => (success_t success)`

Create a defined area in the logical address space. When `clear` is set and you try to read data in that area afterwards, NULL blocks will be delivered; otherwise the contents may contain uninitialized garbage. When `melt == FALSE`, packet borders will show up in the adjoint nest (NYI).

`$delete (addr_t log_addr, len_t log_len, bool melt := TRUE) => (success_t success)`

Create a hole in the logical address space (and throw away data). Reading or writing in defined parts of a sparse nest instance will result in errors (`success == FALSE`).

`$move (addr_t log_addr, len_t log_len, offs_t offset, offs_t offset_max := offset) => (success_t success, offs_t offset)`

The famous move operation. Used for space management in logical address spaces.

In extension to the description from the papers, two offsets `offset` and `offset_max` may be specified. The implementation is free to choose any distance in between these two values, whichever is more convenient or efficient. The actually chosen `offset` is returned again.

3.1.7 Combined Operations

The elementary operations are meant to be orthogonal to each other, i.e. no operation can be simulated by a combination of others. However, in many cases typical usage patterns require always the same sequences of operation calls. For example, reading from a pipe involves `$gadr`, `$get`, `$trans`, and `$wait`. After consuming the data, `$put`, `$delete` and `$padr` is called. To save the overhead of issuing each call separately, combinations are defined which execute such sequences in a single call.

Implementation of combined operations is optional (at least in theory). If you don't implement one, a default version as defined in `common.h` is automatically used. If you implement one, it *must* deliver the same semantics as the default version! Please study the default implementations. All you can (and should) do is to implement *better performance* for the combined version. Combined operations are in particular necessary for distributed systems, because the *latencies* for issuing separate calls would *add up* when you would call all elementary operations sequentially.

TODO: add further combinations. The current set is incomplete.

DISCUSS: can we generate all combinations *automatically* from the prototype specs of the elementary operations? Can we do that for the full power set of all possible combinations? Can we create a generic calling syntax for that? Can we even generate specialized high-performance versions from the sourecode of the elementary operations automatically???

In general, the individual members of a combined operation share a single `struct args` instance by simply reusing the same field names many times, even when some of them are produced by a prior call. This saves a lot of unnecessary copying of parameters. In general, the sum of all input parameters consumed by all elementary operations and not produced by any other one must be supplied from the outside. However, parameters which make sense only at a fixed specific value are *not* supplied from the outside, but bound to that fixed value by the combined operation. For example, executing `$trans` directly after `$get` makes only sense when reading data, thus `direction` is automatically set to `direct_read`. In some cases, the *order* of arguments is slightly changed to become better suited for omission of default values.

Combined Transfer Operations

```
$transwait (addr_t log_addr, len_t log_len, paddr_t
phys_addr, direction_t direction, prio_t prio :=
prio_normal) => (success_t success, plen_t phys_len)
```

Execute `$trans` followed by `$wait`.

```
$gettranswait (addr_t log_addr, len_t log_len, prio_t prio
:= prio_normal) => (success_t success, paddr_t phys_addr,
plen_t phys_len)
```

Execute `$get` followed by `$transwait` for reading the data.

```
$transwaitput (addr_t log_addr, len_t log_len, paddr_t
phys_addr, prio_t prio := prio_normal) => (success_t
success)
```

Execute `$transwait` for writing followed by `$put`. THIS WILL VANISH! NO LONGER USE IT! Passing around physical addresses when doing LOGICAL IO is a BAD IDEA! Use the next operation instead:

```
$putwait (addr_t log_addr, len_t log_len, prio_t prio :=
prio_normal) => (success_t success)
```

Execute `$put` followed by `$wait`. The implementation should force an implicit `$trans` internally in the buffer cache if it has not already done so (but most implementations will have done before anyway).

Memory Creation at known address

```
$createget (addr_t log_addr, len_t log_len, bool clear :=
FALSE, bool melt := TRUE) => (success_t success, paddr_t
phys_addr, plen_t phys_len)
```

Execute `$create` followed by `$get` (note that a further `$transwait` for reading would be senseless, because the data has been freshly created. However a `$transwait` for writing zeroed blocks could be feasible, but is there anyone needing this?).

```
$putdelete (addr_t log_addr, len_t log_len) => (success_t
success)
```

Execute `$put` followed by `$delete` with `prio = prio_none`; this is the antagonist of `$createget`. It can also be used to free an invalid buffer without writing back anymore; the internal implementation should choose to cancel any pending IO request for it.

Memory Allocation at unknown address

`$gadrcreateget (len_t log_len, bool clear := FALSE, bool exclu := TRUE, action_t action := action_wait, bool melt := TRUE, len_t try_len := log_len) => (success_t success, addr_t log_addr, len_t log_len, paddr_t phys_addr, plen_t phys_len)`

Execute `$gadr`, then `$create`, and finally `$get`. In other words, allocate some new data block and deliver it, similar to classical `malloc()`.

`$putdeletepadr (addr_t log_addr, len_t log_len) => (success_t success)`

Combination of `$putdelete` followed by `$padr`. This is in essence the opposite of `$gadrcreateget` and similar to classical `free()`.

Logical IO on Pipes

Both the reader and writer must each call two operations, one for preparing the transfer, the other for finalizing. The preparing operation will lead to a pre-reservation, i.e. an *intermdiate state* between allocation or freeness. The finalzing operation will then either free or finally allocate the region.

Reader

`$gadrgettranswait (len_t log_len, bool exclu := TRUE, action_t action := action_wait, forwrite := FALSE, prio := prio_normal, len_t try_len := log_len) => (success_t success, addr_t log_addr, len_t log_len, paddr_t phys_addr, plen_t phys_len)`

This allocates space in the defined region of the nest, allocates a physical buffer and fills it with the data. After that, you may access the data. After accessing the data, you *must* call the following for recycling the buffer space (for usage by the writer):

`$putdeletepadr (addr_t log_addr, len_t log_len) => (success_t success)`

This is the same operation as in memory allocation, but here the *purpose* is slightly different at the *semantic* level only. It frees both the physical buffer and the logical address space.

Writer

`$gadrcreateget (len_t log_len, bool clear := FALSE, bool exclu := TRUE, action_t action := action_wait, bool melt := TRUE, len_t try_len := log_len) => (success_t success, addr_t log_addr, len_t log_len, paddr_t phys_addr, plen_t phys_len)`

This pre-allocates space in the undefined part of the logical address space and a physical buffer (containing unitialized data). After that, you *must* fill your own data in. The buffer must then be made available to the reader by the following operation:

`$putpadr (addr_t log_addr, len_t log_len, prio := prio_background) => (success_t success)`

Physical IO on Pipes

Reader `$gadrtranswaitdeletepadr (paddr_t phys_addr, plen_t phys_len, action_t action := action_wait) => (success_t success, len_t log_len)`

This corresponds to conventional Unix `read()` with copy semantics.

Writer `$gadrcreatetranswaitpadr (paddr_t phys_addr, plen_t phys_len, action_t action := action_wait, bool melt := TRUE) => (success_t success, len_t log_len)`

This corresponds to conventional Unix `write()` with copy semantics.

Preparations of Both Kinds of IO on Pipes The following are straightforward combinations which can be used to prepare both logiocal and physical IO on a pipe:
`$gadrcreatet (len_t log_len, bool clear := FALSE, bool exclu := TRUE, action_t action := action_wait, bool melt := TRUE, len_t try_len := log_len) => (success_t success, addr_t log_addr, len_t log_len)`
`$deletepadr (addr_t log_addr, len_t log_len) => (success_t success)`

3.2 Operations Defined on Inputs

These operations are reserved for exceptional cases, such as signals or critical operation states. In particular, `$retract` may be used for *partially* giving back some resources when there is a shortage. This is required at some places, e.g. simulation of the well-known clock algorithm by `$retract` between a `buffer` cache and `mmu` instances. Another application is retraction of *optional locks* in client-server configurations.

Input operations are "travelling" in the *reverse* direction of wires. This can countercare the hierarchical structure of brick networks and result in various problems.

When you want to call these operations, you *must* use the syntax `@=inputcall`. When executing `@=inputcall` on an input, the operation will be directly called at the specified input. When executing `@=inputcall` on an output (which is the common case), the operation will be forwarded to and executed on *all* inputs to which the specified output is currently connected!

WARNING! This means, a *bunch* of calls may actually be performed. Beware of exponentional explosion in the number of operation calls, and beware of ping-pong endless recursion between "normal" `@=outputcall` and `@=inputcall`!

The `success` status of an `@=inputcall` will yield `FALSE` if any of the actual operation calls yields `FALSE`, otherwise (or when no connection existed at all such that no call will be executed at all) it will yield TRUE.

3.2.1 Initialization / Termination

Analogously to $output_init, the local state of inputs may also be initialized (but try to avoid this! local state is a BAD THING!). Another usage is at (dynamic) arrays of inputs.
`$input_init (bool destr, bool constr, bool clear := FALSE) => (success_t success)`

The sematics is the same as with `$output_init`. The only difference is that the operation is assigned to an input instance, and local input state variables can be directly accessed via `@<fieldname`.

NOTICE: sometimes it makes sense to `@=inputcall` this operation over wires (e.g. for recursively switching to a stateless state), but in most cases it will be called by the `control_*` level, in particular by the `$instconn` operation.

3.2.2 Retraction of Resources

`$retract (prio_t prio, addr_t log_addr := 0, len_t log_len := (len_t)-1, addr_t try_addr := log_addr, len_t try_len := log_len) => (success_t success)`

When this operation is called, you *should* or *must* give back any state (whether physical memory achieved by `$get` or locks achieved by `$lock`, PC information, or whatever) to the former owner. The `prio` is telling you the urgency:

When `prio == prio_urgent`, the resources may be "confiscated" if you don't *immediately* return the resources from `log_addr` to `log_addr+log_len`. In the area specified by `try_addr` and `try_len`, you are free to return additional resources. Confiscation means that resources may become *invalid* without notice; this may result in failure of your brick instance. This case is much similar to a termination signal.

When `prio == prio_normal`, you must also return the resources *immediately*, but confiscation will not occur.

When `prio == prio_background`, you may delay the return of the resouces for some time, e.g. if you are currently using them for some impartant task.

When `prio == prio_none`, you are free to return some of the resources or not. This may be used as a reminder to return currently unused resources.

NOTE: you may implement *optional locking* by this operation.

3.3 Operations Defined on Bricks

NYI

3.4 The General Operation `$op`

You may implement a pseudo-operation[1] called `$op` in any section (or for all sections by specifying `(:ALL:)`[2]). Whenever you omit the implementation of another elementary operation, `$op` will be used by default instead. In addition to elementary operations, combined operations which don't invoke any already implemented operation are also handled by `$op`. This means, you can *ensure* that *any* unimplemented operation not conflicting with an implemented operation is automatically redirected to your `$op` implementation. If you only implement `$op` and nothing else, any operation will be redirected to it.

Note: the redirection is done at preprocessing time.

Inside `$op`, you can find out the originally called operation via `@op_code`, which is of `enum` type `op_t`. You can also find out the section number via `@sect_code`. Usually you will have to program some rather large `switch` statement by hand in order to handle argument polymorphism correctly. Please note that in `$op`, you can use any valid `@name` without automatic checking for consistency with `@op_code`, so you are yourself *fully responsible* for correctness of argument polymorphism!

You can call the pseudo-operation `$op` by hand, but only via shortform syntax. Longform syntax is unavailable due to unresolvable polymorphism. Thus you have to ensure by hand that `@op_code` and `@sect_code` have correct values *before* calling `$op`, as well as that `@success=FALSE`. A useful application is polymorphic forwarding of calls to avoid the code bloat produced by `$OP_NAME`. The generated code will automatically dispatch to the right operation by examining `@op_code`; note that `$op` itself does not really exist in the nest interface.

[1]The pseudo-operation `$op` is translated to a single C function.

[2]See `section(:ALL:)` as described in the preprocessor guide.

3.5 The Generic Strategy Interface

The generic strategy interface is explained in `strat.h` / `strat.c`. It is just a special data format for the interpretation of the contents of a nest. Each brick instance of a network is represented by one data block. Currently its size is limited by `DEFAULT_TRANSFER`, but this may be changed at a later revision. Each brick instance is represented by a `NULL`-terminated ASCII string with a C-like syntax. Here is an example:

```
brick=buffer_dummy_linux {
  param buffer_dummy_linux=init_param
  input=dev {
    param dev=other_param
    connect dev==11:out
  }
  output=out {
    param out=out_param
    connect out=13:in
  }
}
```

When calling `$transwait` for reading, you will get a string with `LF`-terminated lines. When you write such a string, actions may be performed by the strategy level (e.g. `control_*`). However, actions will only be performed by those lines where a `:=` or `/=` is used in place of `==`. When you simply write back the original string containing only `==`, nothing will happen. When something cannot be changed at all (*immutability*), a single `=` indicates this in place of `==`. You cannot replace `=` by `:=` or `/=`. There is only one single exeption: the *type* of a brick instance (its name) is immutable at runtime (since changing it would affect the existence of inputs and outputs), but you can de-instantiate it by saying `brick /= buffer_dummy_linux\n`.

As another example, when you submit a string `connect dev/=11:out\n` the wire to the output `out` of the `device_dummy_linux` instance at logical address 11 will be de-instantiated. With `:=` you can afterwards create a new wire. Whether a parameter or an attribute of a brick (or of an input or of an output) can be changed at all is indicated by == versus =.

You can use regular expressions for searching and replacing strings in a generic strategy nest. The intention behind this approach is to ease the development of *universally generic* `strategy_*` bricks which operate on strategy nests in a rule-based fashion, similar to sed and awk scripts operating on ordinary text files. For example, you can write a universally generic interpreter or compiler performing *transformations* at the strategy level, which are specified by (enhanced) graph grammars, by tree automata, by L-systems, or by some expert system, or by many other methodologies developed in theoretical computer science or artificial intelligence.

3.6 Additional Strategy Operations

The manipulation of small ASCII strings representing simple bricks with a low number of inputs and outputs may be even faster than inquiring and manipulating each input/output and their attributes separately and sequentially (e.g. consider the problem of dynamic space allocation at the interface level for variable-length string parameters). Most simple bricks like `adaptor_*` will fit in this category. In addition, a `$trans`fer of a single string embodying many `:=` actions can save network latencies when compared to issuing each `:=` action separately. However, for very large local bricks with many inputs and outputs, such as `fs_*`, the string representation may induce serious performance problems.

In order to address such performance issues, I have decided to add further operations for performing manipulations inside strategy nests. These operations are *redundant*, i.e. they just perform the same semantics as directly manipulating the corresponding ASCII string representation. Moreover, there are ways for *automatically translating* between a strategy operation and its corresponding ASCII representation, and vice versa.

Thus you may decide to implement only one of both interfaces. For example, for rapid prototyping the ASCII string interface is usually more convenient. However, when performance at a local site is crucial, implement your functionality by strategy operations and let them automatically be called when somebody submits his requests as ASCII strings.

The following is experimental and may be changed in later revisions.

`$instbrick (addr_t log_addr, name_t name, bool constr := FALSE, bool destr := FALSE) => (success_t success)`

When no brick instance already exists at address `log_addr` of the strategy nest, instantiate a new one with type `name`. Otherwise and/or when either `@constr` or `@destr` is `TRUE`, call `$brick_init` with those parameters.

NOTE: at the first creation of a new brick instance, you will leave `@constr` and `@destr` as `FALSE` in most cases, because the inputs have not yet been wired (and thus `$brick_init` cannot yet be called). However, in many cases you will call `$instbrick` again at the same address as soon as you have connected all relevant inputs.

`$deinstbrick (addr_t log_addr, bool destr := TRUE) => (success_t success)`

When `destr==TRUE`, first call `$instbrick`. Afterwards, de-instantiate the brick at address `log_addr`.

NOTE: by setting `destr` to `FALSE`, you can avoid calling `$instbrick`, e.g. if you already know that it has been called previously, or if you have already called `$deinstconn` for all inputs and outputs by hand.

```
struct conn_info {
  addr_t conn_addr;
  index_t conn_index;
  sname_t conn_name;
};
```

This structure is used for describing a connector: the start address of the brick representation in the strategy nest, the connector name, and the index (only used for arrays of outputs).

`$instconn (struct conn_info * conn1, bool clear := FALSE, bool constr := TRUE, bool destr := FALSE) => (success_t success)`

When `conn1` denotes as *dynamic* array member which does not already exist, instantiate it first. Afterwards, its `$input_init` or `$output_init` operation is called with parameters `clear`, `constr` and `destr`.

NOTE: when you call `$instbrick` which finally results in a call to `$brick_init` with appropriate parameter, you don't need to call this fopr any static input or output (even for static arrays). Otherwise, don't forget to call this for *any* connector, otherwise it will not be in operating phase.

ATTENTION for dynamic arrays: when `conn1->conn_index == -1`, instantiate a new dynamic input or output at an unused array index, and return that index in `conn1->conn_index`. For non-arrays, please initialize `conn_index` always to 0.

`$deinstconn (struct conn_info * conn1, bool destr := TRUE) => (success_t success)`

When `destr == TRUE`, call `$input_init` or `$output_init` for destruction. Afterwards, de-instantiate the connector.

NOTE: normally, this is only needed for *dynamic* array members; static ones are usually automatically handled by `$deinstbrick`.

`$connect (struct conn_info * conn1, struct conn_info * conn2) => (success_t success)`

If the own input `conn1` or the output `conn2` of another brick instance has not yet been initialized by `$instconn`, it will be done now (also called *lazy* initialization). In any case, create a new wire by connecting the own input `conn1` to the foreign output `conn2`.

`$disconnect (struct conn_info * conn1) => (success_t success)`

Disconnect the input `conn1` from its partner by deleting the wire leading to it. Note: `$deinstconn` is *not* performed on neither input or output.

`$getconn (struct conn_info * conn1, struct conn_info * res_conn, index_t conn_len) => (success_t success, index_t conn_len)`

Inquire the connector `conn1` (may be input or output) about its partner and return results in the array `res_conn` whose maximum length (in bytes) is given in `conn_len`. Upon success, the actual number of bytes written to `res_conn` is returned in `conn_len`. Note that input can be connected to at most 1 partner, but outputs to any number.

`$findconn (struct conn_info * conn1, struct conn_info * res_conn := NULL, index_t conn_len := 0) => (success_t success, index_t conn_len)`

Upon calling, only `conn1->conn_addr` need to be initialized. Find the connector with a param value as specified by the generic operation parameter `@param` and return its name in `conn1->conn_name` and index in `conn1->conn_index`. When `res_conn` is set, the currently connected partners will be delivered in the same way as above. Currently used for efficient lookup of filenames and pathnames.

3.7 The Main Directory

In each strategy nest, a special virtual brick instance exists at address 0. It has the name `ATHOMUX_MAINDIR` and contains only outputs. New outputs with new output names may be instantiated at any time, and connections to other bricks may be added at any time. It is intended to hold the "anchors" of the strategy nest, e.g. the root of the filesystem, the main `control` instance, and others. `ATHOMUX_MAINDIR` is a true dummy brick which cannot execute any operations. By convention, its outputs should be only connected to inputs of other bricks named `hook`, in order to indicate that its topological use.

3.8 Regex Library Functions

In `strat.h`, you will find some strings denoting commonly used regular expressions for searching in strategy nests. You will also find some parsing routines based on such strings. These are only for your convenience; you may use other methodologies for searching and replacing in strategy nests as you like.

4 Error Handling

Any operation in an operating system may potentially fail. Thus ATHOMUX adopts the philosophy to check the return code of *any* operation. Always. With almost[3] no exeption.

As already mentioned, the parameter `success` is consistently used for determining success. It is guaranteed to be `FALSE` at the start of an operation invocation. Thus you simply can `return` from the operation to indicate some failure.

The basic application logic of ATHOMUX should be built on this extremely simple method solely. However, there are cases where more complicated error analysis has to be performed. For example, POSIX compatibility requires to distinguish between different error codes. How to implement that in ATHOMUX?

The idea is rather simple: don't implement POSIX error codes in ATHOMUX bricks, but possibly only in adaptor bricks. Within native ATHOMUX, use one of the following error handling macros as defined in `common.ath`:

`@.err (text...)`

Unconditionally return from the current operation (or function) via `return`, without altering `@success` (which is normally left `FALSE`). Before return, the `text` (and further arguments in the style of `printf()`) will be passed to the caller, who can analyze it or pass it to a human. Currently, the text is simply printed to a debug output, but this will soon

[3]Well, there may be some exeptions in *very* special cases, e.g. when trying to fix some problem and the result of the fixing operations has no direct influence on further measures.

be changed; the text will be pushed on a "reverse stack" (which is pushed upon an error return and popped upon a fresh operation call).

`@.check (condition, text...)`

When the `condition` is true, do a `@.err`.

`@.abort (text...)`

Unconditionally terminate. This corresponds to a "kernel panic". Use *only* in absolutely messy situations when there is really no more chance to recover from the error in any way.

`@.fatal (condition, text...)`

When the `condition` is true, do a `@.abort`.

These macros will automatically add further information for the caller, in particular the name of the brick type and the brick instance. Since ATHOMUX is composed of many fine-grained and lightweight brick instances, this carries a lot of information.

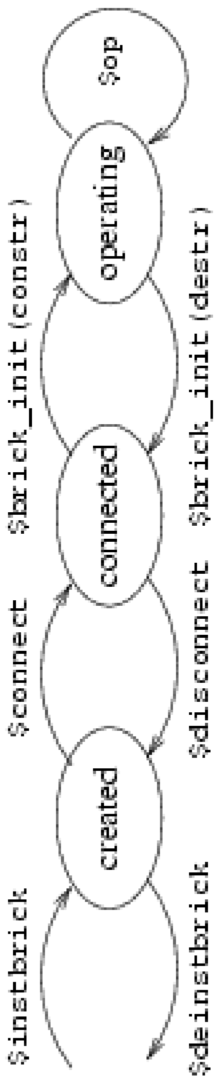
In particular, when you implement POSIX error handling, implement each different POSIX error check in another brick type! For example, use one brick type to check for `EPERM`. This way, you can uniquely determine the error code from the brick type. Using trivial bricks for such checks will cost no performance penalty once the preprocessor is smart enough to employ sophisticated macro expansion at the brick level with almost no runtime overhead. With this philosophy, error handling can be highly modularized and becomes highly configurable.

`@.warn (condition, text...)`

When the `condition` is true, print a warning message without returning.

5 The Lifecycle of Brick Instances

Usually a brick instance goes through many phases during its life:


$instbrick
created
$connect
connected
$brick_init(constr)
operating
$op
$brick_init(destr)
$disconnect
$deinstbrick

According to the figure, operations must be called in a certain order for both construction and destruction of brick instances.

5.1 Created Phase

After a brick has come into existence by `$instbrick`, it is called *created*.

At created phase, no inputs have been connected yet. Thus no operation can be called at all, not even `$brick_init`. Before calling any operation, at least the inputs must have been connected.

Special case: if there are no inputs at all, the created phase is equivalent to the connected phase; this may be the case for some device drivers.

During creation, the `init{}` routine for the brick is called. Since there are no connections yet, you *must not call* any operations from `init{}`; not even despite the fact that the `@=call` syntax does not work to prohibit you from trying that.

Symmetrically, `exit{}` is called upon `$deinstbrick`. Notice that `$deinstbrick` is only when all inputs have been disconnected.

5.2 Connected Phase

As soon as all inputs have been connected via `$connect`, the brick is called *connected*.

IMPORTANT: prior to connected phase, you **must not call** any operations, not even `$brick_init`!

Special case: for (dynamic) arrays of inputs, not all of them must be connected, since the number of connections can vary dynamically at runtime. However, you are at your own risk for ensuring / checking that no bad operation call at an unconnected array member can occur.

In order to return to created phase, you must call `$disconnect` for any connected input.

5.3 Operating Phase

Before any operation can be called at any input or output, the corresponding input / output must have been initialized by `$input_init` or `$output_init` with `@constr=TRUE`. Then the input / output is called *operating*.

IMPORTANT: It is an error to call any operation on a non-operating input or output; it is no longer possible to implement outputs in such a way that non-operating mode can be tolerated. When you forget to initialize an input or output before calling any operation on it, a predefined stub implementation will be called instead, always returning `@success=FALSE`; additionally, in debug mode it may terminate your executable (depending on the Athomux environment where you run it).

Instead of calling `$input_init` and `$output_init` for all inputs and outputs individually, you may call `$brick_init` instead which by default calls `$output_init` for all inputs and outputs except *dynamic* arrays.

Turning back to connected phase is possible by calling `$*_init` with parameter `@destr=TRUE`. IMPORTANT: before returning from such a destruction, be sure that any logical state (if any) has been returned; this may be achieved by `$retract` from all outputs.

5.4 Macros for Inquiry of the Phase

The following macros (from `common.ath`) may be used to determine the current phase of a brick, input or output instance:

`@.is_connected(`*input_spec*`)` Determine whether the input is currently connected. The input belonging to the current operation can be abbreviated as `:<` without a name.

`@.is_initialized(`*connector_spec*`)` Determine whether the input or output is in the operating phase, i.e. has been *successfully* (i.e. `@success==TRUE`) initialized by `$input_init` oder `$output_init` (e.g. indirectly via `$instconn`). If used inside `$*_init`, you can prevent double initialisation (or senseless de-initialisation).

5.5 Macros inside `$brick_init`

When you implement your own `$brick_init`, you can (and in fact, should!) call `$input_init` and `$output_init` for all of your inputs and outputs by hand. This could be cumbersome for large bricks with a huge number of inputs and outputs. Therefore you may call some of the following predefined macros at any place where you need it:

`INIT_ALL_INPUTS(BRICK)` Call `$input_init` for all existing inputs. The return status is relivered in `@success`.

`INIT_ALL_OUTPUTS(BRICK)` Call `$output_init` for all existing outputs.

`INIT_ALL_CONNS(BRICK)` Call `$*_init` for all existing inputs and outputs in one single sweep (slightly more efficient). NOTE: when @constr==TRUE, the order of initialization is exactly as specified by `input` and `output` statements in your sourcecode; otherwise the order is reversed.

`INIT_ALL_INSTANCES(BRICK)` Call `$brick_init` for all sub-instances.

When you don't implement `$brick_init` explicitly by yourself, a default version is automatically created which does nothing but simply calling `INIT_ALL_INPUTS(BRICK)`, `INIT_ALL_INSTANCES(BRICK)` and `INIT_ALL_CONNS(BRICK)` in sequence.

6 How to Implement Statelessness

7 Threads and IPC

The term "thread" means something more generic in ATHOMUX than in conventional sense. ATHOMUX itself has no concept of a thread; in particular there is nothing like a PID or threadID. If you want to implement the *behaviour* of threads (aka concurrency) or if you want communication among different threads / `mmu` instances, you can (and must) do so inside the blackboxes of some brick implementations.

Creation of threads is *implicitly* just by calling an operation *asynchronously*. In ATHOMUX, the notation for asynchronous calls is the same as for synchronous ones; only the *semantics* is different. The discrimination between synchonous and asynchronous calls is just by calling an operation on an input implementing *asynchronous behaviour* instead of synchronous behaviour. The asychronous behaviour itself is implemented in `thread_*` bricks.

When a `thread_*` instance is wired into an arbitrary wire path, it creates a new thread each time it receives some operation call at its output, and returns immediately to the caller. The new thread is then executing that operation in parallel to the caller, asynchronously. If you need some cooperation between the threads, you have to implement it yourself (e.g. via locking).

A `thread_*` can be instantiated locally in an arbitrary other brick, and its input can be statically wired to an internal (hidden) output. This way, the enclosing brick has full control over anything going on.

Locking is implemented in `lock_*` bricks.

The internal implementation of different `thread_*` types as well as of `lock_*` will usually be very different, depending on the *execution environment* where it runs. When ATHOMUX is hosted by an ordinary Linux userspace process, it may be implemented by the `pthreads` library. When running in Linux kernel space, you may use kernel threads, spinlocks, wait queues, and the like. In a native standalone ATHOMUX implementation, you will have an architecture-dependent `cpu_*` instance acting as "device driver" for the physical CPUs, and you may communicate with it via traps or other interrupts (e.g. page faults when "blindly" calling into non-mapped executable code). Note that a single `cpu_*` instance should be able to serve an arbitrary number of `thread_*` and `lock_*` instances. Communication between these instances may be outside the standard nest interface of ATHOMUX, and in some cases necessarily must be so due to hardware restrictions.

Interprocess communication (IPC) is implemented via communication bricks. To get the *abstract functionality* of `remote_*`, you will usually implement a client and a server part, each running in a different execution environment. There may be many different pairs of client and server brick types, depending on the underlying communication mechanism. There may be many different LRPC implementations based on host operating systems, traps, or hardware specific mechanisms such as Intel call gates. In the area of networking, many different implementations based on different protocols and technologies may exist.

I believe that not prescribing a specific way for implementation of parallelism, synchronization and communication is the most flexible and universally generic way. You may exploit the information hiding of blackboxes for interfacing and communicating with and via nearly anything imaginable on earth.

7.1 Mutual Exclusion inside Bricks

Currently, you are responsible for implementing a brick in a thread-safe fashion. You can always ensure thread-safety by calling `$lock` "by hand". Sometimes you need not do anything, e.g. when your operations are working on the stack solely.

TODO: some future brick attributes will produce automatic code for thread-safety in an automatic way. Then you will not need to worry about mutual exclusion if your brick is not *extremely* performance-critical.

8 Generic Types / Operations

Although the genuine Athomux philosophy encourages reuse of the standard nest interface at each possible opportunity, in some cases a richer set of operations is required (e.g. simulation of `ioctl()` operations). In other areas, access to foreign data structures with a particular *layout* is required, e.g. in networking code.

Both problems are solved via *generic types*. A generic type is much like a C `struct`, but more flexible. It can not only express arbitrarily overlapping structures (even with holes) and variant records, but it can also *delay* the offset computation until runtime.

If you need to implement a generic operation, define a generic type for its parameters. Then implement a standard `$trans` operation (e.g. on a dedicated section of the nest such as section `(:2:)` which is often reserved for generic operations), such that the passed data block is interpreted via the generic type access operation `@*->`.

8.1 Defining an own Generic Type

After an `output` statement, you may declare a generic type via the following syntax:
`define TYPE` *typename* `"`*string*`" ;`
`define export TYPE` *typename* `"`*string*`" ;`

The second variant exports the typename for use via `use` statements (see section 8.6).

All defined *typename*s must be distinct (note: this may change later when/if scoped defintions are introduced). The string must be a comma-separated list of type declataions of the following form:

type fieldname`:`*offset*

where *type* is a valid C type, *fieldname* is the designated field name, and the optional `:`*offset* (which must be a constant C expression) can be used to directly specify the offset of this fields in units of bytes. When `:`*offset* is omitted, the field will immediately follow after the previous field without any gap (NOTE: some architectures like SPARC require alignment to machine words, which is *not* ensured by our generic mechanism; anyone needing this???)

In the constant expression `:`*offset*, you may use symbolic pseudo-references to previously defined fields via the syntax `OFFSET(`*otherfieldname*`)` and `LENGTH(`*otherfieldname*`)`, with obvious meaning. By placing some fields at the same offset, you may define arbitrary variant structures or overlapping structures.

In the code of arbitrary operations, access to the fields is possible via the following syntaxes:
pointer_identifier`@*`*typename*`->`*fieldname*
`(`*pointer_expression*`)@*`*typename*`->`*fieldname*

The semantics is much like the `->` operator of C, except that the *typename* must be provided each time. The generated code is as efficient as in ordinary C (resulting in addition of a constant displacement to a pointer value) when using `define TYPE` this way.

WARNING: the `@*`*typename*`->` operator is interpreted by the Athomux preprocessor, not the C compiler. It is much like a macro substitution. It has no knowledge of the context, in particular of C operator precedence. When in doubt, use parentheses around the pointer expression.

HINT: in conjuction with the `@.deftype` directive of the macro preprocessor, you may use the abbreviated form `@*->` (without specifying *typename* each time) for shorter and better readable code.

8.2 `@.sizeof(`*typename*`)`

The default field name `LASTFIELD` is automatically added to each defined *typename*. It captures the offset of an empty field having length 0. When you write `&(`*pointer*`@*`*typename*`->LASTFIELD)`, you will get the first free address after all defined fields. By writing (len_t)`&(NULL@*`*typename*`->LASTFIELD)` or using the standard macro `@.sizeof(`*typename*`)` (see `common.ath`), you can determine the total size of a generic type.

8.3 Variant Structures

In place of a field declaration, the *string* may contain a variant definition:
`.`*label* `{`*substring*`}`

where *substring* is a comma-separated list of further type definitions. Usually, you will specify a comma-separated list of multiple definitions like `.label1{`*substr1*`}, .label2{`*substr2*`}`. The fields of each *substr* are (by default) overlayed with the fields of the other *substr*s; notice that all label names and all field names must be distinct since they belong to a flat namespace. The total size of the variants will be the maximum size of all alternatives. When some ordinary field is following the `.`*label* list, its default offset will start at that maximum.

NOTE: when some *substr* contains fields with zero or negative offsets (e.g. produced by `OFFSET(`*prior_field*`)` or the like), the total size may be zero, but never become negative.

When the *substring* of some `.`*label* definition contains further `.`*sublabel* directives, the total result is a flat label namespace with labels of the form `.`*label*`.`*sublabel*.

8.4 Variant Selection

Variants can be removed from a generic type defintion *string* by appending a comma-separated list of selection specifiers to the *string*:

`=.`*label*

This removes the fields of all other labels, such that only the fields belonging to `.`*label* will survive. If you need the survival of multiple variants, you may specify a comma-separated list like `=.label1,=.label2`.

`!.`*label*

Definitely remove the fields of `.`*label*.

Variant selections are particularly useful in combination with OO-like extension of generic types:

8.5 OO inheritance

The following syntax allows the extension of existing generic types by additional fields:
`define TYPE` *typename* `from` *othertype_list* `"`*string*`" ;`

where *othertype_list* is a comma-separated list of already defined generic types.

Obviously, you can even simulate multiple inheritance this way. However, recall the Athomux philosophy (also called LEGO principle) and please don't abuse this mechanism for building large OO class hierarchies. Athomux thinking is purely instance oriented, not object oriented.

NOTE: when *string* is empty, you may define another name for an already defined generic type. When *string* contains variant selection specifiers such as `=.`*label*, you may easily derive a specialized type from a more general one.

8.6 Use of Foreign Type Definitions

After an `input` statement, you may declare a foreign type via the following syntax:
`use TYPE` *typename* `from` *othertype_list* `"`*string*`" ;`

In contrast to `define export TYPE`, you *must not* specify any `:`*offset* in the *string*. A foreign type declaration just tells the preprocessor that the mentioned fields will *exist*, but not at what offset they will exist! Thus the order of the fields inside *string* does not matter.

The concrete offsets will only be known after a `$connect` operation (or an internal `wire` statement) which connects the input to another output. The output must have a `define export TYPE` statement with the same *typename*, and at least all fields mentioned at `use TYPE` must be present at the foreign `define export TYPE`; however much more fields may be defined than used. When you `$connect` or `wire` to different outputs with different `define export TYPE` statements, the resulting offsets of your fields may be different in each case.

NYI: when the `use TYPE` declaration contains a field name not present in the `define export TYPE` (or a mismatch of the C type strings), an error will occur. Thus type-safety is enforced.

9 Debugging

During the debugging phase, you often need to write traces to some log files. In order to that in a *portable* way for different Athomux build environments, you should use the following macros:
`@.trace(`*name*`, "`*printf-string*`", `*args*`...)`

The *name* denotes a freely chosen name of a debug output file (or channel). The build system automatically retrieves all names occurring in all *.ath source files and maintains to open all the channels for you.

10 Commodity Library Routines

These library routines are intended for simplifying your task of programming ATHOMUX bricks. You don't need to use them, but they can probably save you a lot of time and code to write. They are not inteded to be extensible; if you think further functionality is needed at the common infrastructure, you should discuss that with all other developers.

10.1 The Pointer Cache (PC)

In both stateless and pseudo-stateless brick types, you will have to manage local state *explicitly*, i.e. you normally keep your state in an input nest instead of in @# variables.

Keeping state this way bears some traps: you **must not use C pointers** for addressing objects out of other objects! This is because a C pointer is only valid after a `$get` or `$gettranswait`. Once you have done a `$put`, the next re-`$get` of the same logical page at the same logical address may deliver it at a *different physical address!* Moreover, the physical address may be totally different at a remote site in a network of computers. Physical addresses are *meaningless* even on persistent nests!

Thus you **have to** use logical addresses (type `addr_t`) instead of C pointers inside of your state keeping `struct`s when you want to reference other state keeping objects!

As long as the space requirement for your state is statically bounded, it is easiest to allocate one or a few pages "by hand". This method should be preferred whenever possible.

Another simple way is using *arrays of pages*. To address smaller logical array elements than DEFAULT_TRANSFER, you can place of bunch of the smaller objects them in each page and maintain index computation by hand.

Although this is nearly trivial, writing code for such a thing will increase redundancy unnecessarily. The pointer cache (PC) will automate that task for you.

10.1.1 Basic Pointer Caches

You can declare one are many independent pointer caches after an `input` statement via the following kind of declaration:
`use PC` *name* `[`*max*`] ;`

The names of all pointer caches must be distinct. When the constant *max* in brackets is omitted, 16 is used as default for the number of cache entries; for best efficiency *max* should be a power of 2.

NOTE: when ATHOMUX is compiled for a 32-bit Linux kernel environment, 64-bit division and modulo operations are not available (since libgcc is missing). In this case, all constants *must* be powers of 2, since only then bit-operations can be substituted for divisions and modulo.

After the declaration, you can access the PC via the following macros:

res_ptr = `PC_GET(`*name*`,` *log_addr*`,` *log_len*`)`

When the physical address for *log_addr* is not already present in the cache, it will be fetched via `$gettranswait`. As a result, you will get a `void*` pointer.

When your access patterns bear high locality, a `$gettranswait` will not be issued every time once again, but rather some previously cached pointers will be reused. Hence the name “pointer cache”.

NOTICE: when the underlying `$gettranswait` fails due to some reason (e.g. undefined *log_addr*), you will get a `NULL` pointer. Don’t forget to check for this condition!

`PC_DIRTY(`*name*`,` *addr*`)`

Mark your object as dirty. You *must* call that at least once after `PC_GET()` whenever you have modified your data; forgetting this may cause strange effects! When you know *in advance* that you will touch the data, please use `PC_GET_DIRTY()` instead (for saving some overhead). Please use `PC_DIRTY()` only if you don’t know in advance whether you will modify the data.

res_ptr = `PC_GET_DIRTY(`*name*`,` *addr*`,` *len*`)`

Like a combination of `PC_GET()` followed by `PC_DIRTY()`, but saves some overhead. Always calling `PC_GET_DIRTY()` when you actually don’t modify the data is no good idea, because writing back of dirty pages may involve high costs, e.g. when your state is kept remote in a network (which is *possible*, and you don’t know to which other brick instances your state-keeping input will be connected at runtime).

`PC_SET(`*name*`,` *log_addr*`,` *log_len*`,` *phys_ptr*`,` *dirty*`)`

Set a cache element “by hand”. This makes sense after you have called `$gadrcreateget` by hand, in order to avoid useless traffic for re-getting the same data afterwards. See notes in next subsection.

`PC_UNSET(`*name*`,` *log_addr*`)`

Release a single cache element via `$put` (if present).

`PC_FLUSH(`*name*`)`

Release all internally cached data via `$put`. After that, no physical pointer is valid anymore. You *must* call `PC_FLUSH()` for all your caches at your implementation of `$init` when the argument `destr` is `TRUE`. Otherwise your brick will not only be stateful instead of pseudo-stateless, but also be *incorrect* (due to non-returned resources).

DISCUSS: should the code for calling `PC_FLUSH()` at `$init` be automatically generated by the preprocessor?

10.1.2 Pointer Caches with Allocation

Allocation “by hand” is cumbersome and error-prone. For your convenience, here are routines imitating the classical `malloc()` / `free()` routines:

phys_ptr = `PC_ALLOC(`*name*`,` *log_len*`) => (`*log_addr*`)`

This calls `$gadrcreateget` for you and inserts the new element into the PC, already marked as dirty. The logical address is returned as result parameter, and a valid C pointer as function result. In general, you will need to store *log_addr* somewhere, because it remains valid after `PC_FLUSH()`, but *phys_ptr* will not be valid anymore.

`PC_FREE(`*name*`,` *log_addr*`,` *log_len*`)`

This clears the cache element (if present) and returns the memory to the underlying nest via `$putdeletepadr`. Note that in general you *must* supply the same size as formerly at the allocation; this is needed because the cache element need not be present and then we will not know the former size any more.

10.1.3 Granularity Considerations

In many cases, memory elements will be multiples of some block size or some data structure size. The following declaration generates a more efficient implementation for that case:

`use PC` *name* `[`*max*`] aligned ;`

Now, all logical addresses and sizes *must* be multiples of `DEFAULT_TRANSFER`. Doing that wrong is an error. Therefore we supply a further variant:

`use PC` *name* `[`*max*`] aligned round;`

Conceptually, all logical addresses and sizes are also multiples of `DEFAULT_TRANSFER`. However, if you supply “inequal” addresses or sizes, they will be *automatically corrected* by the pointer cache implementation. This means, the lower bits of the logical address will be internally cleared (i.e. “rounded down”), and the size will be internally rounded up to the next multiple of `DEFAULT_TRANSFER`. This way, you can access smaller objects inside larger pages *without fear*. If two smaller objects happen to live in the same page, the same cache element will handle that case for you (leading to better performance due to coincidence of cache hits).

However note that this also applies to `PC_ALLOC()` and `PC_FREE()`. If you don't manage the sub-space inside your pages "by hand", you will probably either waste space or produce incorrect deallocation side effects due to different granularities.

You may "round down" an address by hand via the macro `PC_BASEADDR(`*name*`,` *addr*`)`, if you are interested in doing so.

`use PC` *name* `[`*max*`] aligned(`*size*`) ;`

This variant (which is combinable with `round`) allows you to define other granularities than `DEFAULT_TRANSFER`. Although it works for any *size*> 0, it is advantageous to use a constant power of 2: bit-operations will be used internally instead of division or modulo operations.

DISCUSS: Should we supply variants of `PC_ALLOC()` and `PC_FREE()` which don't round (but the others do)? Then the underlying nest implementation could automate the task of sub-allocation in pages.

10.1.4 Current Problems

Currently, a new `PC_GET()` may invalidate a prior pointer delivered by a prior call to `PC_GET()` if both happen to conflict in the internal hash table of the PC. TODO: This will be resolved by introduction of a syntactic construct which a) limits the accessability of the physical pointer and b) "locks" the hash entry so it can't disappear.

10.2 Cyclic doubly-linked lists

Our list implementation is based on a pointer cache (PC) which *must* be declared as `aligned(`*elemsize*`) round`, where *elemsize* corresponds to the `sizeof` of the list members struct (or, better, a round-up to the next power of 2).

Usually, you will declare some `struct elem` which contains at least one `struct link` (see `common.h`).

Doubly-linked lists cost more space than single-linked ones (because two `addr_t` pointers must be kept instead of one). But they have the advantage that removal of *any* element from the list costs only $O(1)$ time instead of $O(n)$. This occurs frequently, e.g. at LRU lists.

Cyclic lists are slighly more difficult to understand than 0-terminated ones, but easier to program and more efficient, because special cases at the edges are eliminated.

This implementations assumes that the anchor of type `struct link` is kept in a blind element, and that all addresses directly refer to the next (or previous) `struct link` inside each `struct elem`, including any offset. The enclosing structure of your `struct link` is always addressed by rounding down to the *elemsize* by the underlying PC.

`LI_INIT(`*name*`,` *anchor_addr*`)`

Initialize the blind list anchor at logical address *anchor_addr* such that it points to itself (cyclic loop). *name* refers to the name of a valid pointer cache which is internally used for updating the data stucture. Note that *anchor_addr* must be the true address of the anchor (including any offset in its *elemsize*), not rounded down to *elemsize*.

`LI_APPEND(`*name*`,` *anchor_addr*`,` *link_addr*`)`
`LI_PREPEND(`*name*`,` *anchor_addr*`,` *link_addr*`)`

Appends or prepends the element specified by logical address *link_addr* to the blind list anchor *anchor_addr*. The *link_addr* must include the offset of the `struct link` in its `struct elem`, not rounded down to *elemsize*[4].

`LI_REMOVE(`*name*`,` *link_addr*`)`

Removes the element specified by *link_addr* from the list. When one of the neighbours is the anchor, no special case must be considered; it is automatically correct even for that case.

phys_ptr `= LI_GET(`*name*`,` *elem_addr*`,` *dirty*`)`

Returns a pointer to the `struct elem`, rounded down modulo *elemsize*.

10.3 Hashes (HASH)

NYI

[4]If the corresponding `struct link` is the first member of your `struct elem`, then of course the offset is 0. In general, multiple `struct link` instances may be embedded in one `struct elem` instance; they will be automatically discriminated by their offset modulo *elemsize*.