The ATHOMUX Preprocessor User's Guide

Thomas Schöbel-Theuer

Version 0.30, 5 Dec 2004

Contents

1 Purpose

The ATHOMUX preprocessor automates many programming tasks which would be very tedious when done by hand. It generates C code from high-level brick specifications, intermixed with ordinary C code.

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 A lot of stuff is missing; this document may soon be outdated.

2 Invocation

`pre.pl` [*options*] *filename*`.ath` [*filename2*`.c`]

Generates files *filename*`.h` and *filename*`.c` from *filename*`.ath`. For inspection and debugging purposes, a *filename*`.pre` is also created, which contains the result of the macro processor expansion phase. When *filename2*`.c` is present, the `.c`, `.h` and `.pre` files will be derived from *filename2* instead. Input and output files may reside in different directories by specification of absolute or relative paths.

2.1 Options

`-d` generate debug code

`-l` generate `#line` directives

`-i` indent the trace produced by the C compiler flag `-DTRACE`

`macroname=value` define a parameterless macro on the command line

3 Structure of an ATHOMUX brick

The following description is no formal language description, but an informal intuitive description for hackers. At almost any places in the sourcecode, you can insert whitespace and C comments (either `//...\n` or `/*` recursively nested comments allowed `*/`). The recursive nesting of `(...)` `[...]` `{...}` pairs is obeyed by the preprocessor; most syntax errors will result from incorrect nesting! Currently the preprocessor will issue syntax errors

for all wrong syntax *at the preprocessor level*, but the messages may not be very enlightening. Whenever some C code appears in your Athomux sourcecode, it is passed to the C compiler nearly unchecked; bad C syntax and semantics will be catched by the C compiler.

Bad *semantics* at the preprocessor level is checked in many cases, but not always.

TODO: after the experimental stage is left and the language became stable, write a more formal description.

3.1 Brick header

3.1.1 Legal Issues

Each `*.ath` file *must directly*[1] start with a header, indicating the author, copyright and license information for this file:
`Author:` *name*`\n`
`Copyright:` *name*`\n`
`License: see files` *file1*,*file2*`\n`

Notice that according to European law, the author(s) *must* be *personally* mentioned. For multiple authors, please repeat the `Author:` line. Under European law, removal / replacement / transfer of author information is generally not permitted, not even with consent of the author itself, and not even by contract. Notice that the GPL / LGPL is *necessarily* based on that European law whenever a European author starts to write a new brick under (L)GPL.

In contrast to the author information, the `Copyright:` section can name an institution such as the Athomux Society (which has yet to be founded), if the copyright has been transferred accordingly. The transfer may even take place after publication of sourcecode.

After the `License: see files`, a list of file names residing in the root directory of the Athomux release should be specified. As an author of a new brick, you can freely choose whatever license you like. However, it is strongly recommended to use both the default `SOFTWARE-LICENSE` (which encompasses both the GPL and LGPL) and `PATENT-LICENSE` (which permits free use of my patents provided that you put your software under GPL or LGPL). Otherwise, you are not only at your own for obtaining the necessary rights (e.g. in the field of software patents), but you also may complicate legal issues a lot when a larger Athomux system is composed out of incompatible licenses.

3.1.2 Build versions

The following optional statement is ignored by the preprocessor, but used by the `make` system to include or exclude the current brick source code from a particular build version:
`context` *name*`:` *list*... `\n`

Currently, a further keyword *name* and a colon must follow immediately after `context`: either `pconf` (denoting a preprocessor configuration specified by a file `pconf.*`), or `cconf` (denoting a C compiler configuration specified by a file `cconf.*`), or `target` (denoting a `make` target). After this second keyword and the colon, a comma-separated *list* of names denotes the configurations where the current input file is to be included. For example, the name `default` after `context cconf` indicates that the current brick should be included in the configuration `cconf.default`. By specifying an `!` before a name, the *exclusion* from the specific configuration is specified. When at least one `!` appears in the list, any configuration not mentioned in the list will be *included* by default. Otherwise, any configuration not mentioned in the list will be *excluded*. As a consequence, it is recommended that you should uniformly specify either exeptions for inclusion or exeptions for exclusion, but not mix them up.

A more detailed description can be found the documentation titeled *The Build System of Athomux*.

[1] This means, at this place no comments are allowed.

3.1.3 `buildrules`

After optional `context` statements, any number of `buildrules` statements may follow:
`buildrules` *kind*`:` *makefile-rules-text*....`\n endrules`

A detailed description can be found the documentation titeled *The Build System of Athomux*.

3.2 `brick` statement

After the header information, a `brick` statement must follow:
`brick` *#brick_name*
`purpose` *short-description*`\n`
`description` *long-description* ... `enddescription`
`example` ... `endexample`
`attr` *name* = *value*`\n`
...

The documentation part is indicated by keywords `purpose`, `description` and `enddescription`. The `purpose` is described by exactly one line; please keep it short. The `description` should tell the user anything necessary for *using* the brick (intended environment etc.). Note that in this section the correct nesting of braces need not be observed, since it is textual description, no sourcecode. Inside the description, the keyword `enddescription` is forbidden, since it indicates the end.

All documentation phrases are optional (by omitting them completely), but are recommended for any useful brick. An `example` for use cases of the brick may also be added.

The `attr` list may be empty. It specifies static brick attributes (currently NYI). Each line ist terminated by a linefeed.

The `brick` statement may be preceded by the keyword `strategy`. In this case `strat.h` is automatically included and some further code is automatically generated.

TODO: write some tools for automatic extraction of docs from the sources (literate programming).

3.2.1 Static header and implementation definitions

`static_header {`*global_defs*`}`
`static_data {`*global_code*`}`
`static_init {`*global_init*`}`
`static_exit {`*global_exit*`}`

The *global_defs* is an optional part containing pure C code (with properly nested parens and braces) which is copied to the start of the *.h output file *unmodified*. Similarly, the optional *global_code* is copied unmodified to the start of the *.c output file. Normally this should be used *only* for pseudo- or dummy-bricks running in the context of a foreign operating system such as Linux. Don't misuse for bad things! True ATHOMUX bricks should *never* #include foreign header files! If you *really* need that part, you should only place typedefs there or define some constants such as array dimensions, but no static or external variables (if not *absolutely* necessary).

The optional *global_init* and *global_exit* parts may be used for more sophisticated initialization and finalization upon loading / unloading of static module code. You cannot access instance variable from that code, but only static variables (you are *strongly* advised to avoid them at all! Whenever using static data, be sure that *logical* statelessness is never violated!). You must not call any operations (including `@.abort` macros), since this code will be executed at an early stage where *no inputs are connected!* This is only for *advanced usage* such as creating self-describing meta information, self-contained strategy nests and the like. Please use the `static_*` keywords *only* if you *really* know what you are doing!

3.2.2 Instance variables / fields

Instances of bricks may contain "local" variables which will exist during the lifetime of the instance. They are declared
`data {`*instance_var_declarations*`}`
`init {`*instance_var_initcode*`}`
`exit {`*instance_var_exitcode*`}`

The *instance_var_declarations* are C declarations, later copied to inside a C `struct`. This means: nothing else than C field declarations are permitted.

The optional *instance_var_initicode* may contain arbitrary C code which is executed once at brick instantiation time (when `init_`*brickname* is called by the `$instbrick` operation). The instance vars *must* be accessed by the special notation `@#.`*variable_name*, since you will not have an ordinary C pointer for accessing the instance variables in conventional sense. Please don't circumvent the official syntax for accessing instance variables, since the generated code may change in future releases.

NOTICE: the concepts of instance variables is in conflict with the concept of statelessness! Try to avoid instance variables as much as possible! If you use them, always ensure that any state is flushed to an input by the `$init` operation when the `destr` parameter is set (see programming guide)!

3.3 `input` and `output` statements

After the `brick` statement, a sequence of `input` and `output` statements may follow:
`input :<`*name*`(:`*max_sections*`:)`
`attrib` *name min_value max_value step_value* `\n`
...
`data {`*input_var_declarations*`}`
`init {`*input_var_initcode*`}`
`exit {`*input_var_exitcode*`}`

Declares an input as part of the preceding brick specification.

The optional *input_var_declarations* and *input_var_initcode* / *input_var_exitcode* parts are similar to brick instance variables, but accessed via `@:<.`*var_name*.

An input may consist of multiple nest instances called sections, which are always wired in parallel; typical usage is for meta nests. The number of sections must be a constant evaluable by the perl preprocessor (TODO: allow C-evaluable constant expressions). When the *max_sections* number and the `(::)` are omitted, 1 is used as default (e.g. when a meta-nest is not used); this saves some static space (not at *each* instance). Note that the input variable instances are *common* for all sections!
`output :>`*name*`(:`*max_sections*`:)`
`attrib` *name min_value max_value step_value* `\n`
...
`data {`*output_var_declarations*`}`
`init {`*output_var_initcode*`}`
`exit {`*output_var_exitcode*`}`

Declares an output. The *max_sections* is optional as above. Output vars are acessed by the special notation `@:>.`*var_name*.

3.3.1 Arrays of inputs / outputs

The output or input name may be followed by an optional `[`*constant_expression*`]` suffix, declaring an array of outputs or inputs (examples see `dir_simple.ath`). Their local vars are duplicated for each array member, so be careful with their space requirements.

You may access the fields of other array members via the syntax `@:>output_arrayname[`*index*`].fieldname` or

`@:<input_name[`*index*`].fieldname` in the code of your operations.

When the output or input name is followed by empty brackets `[]`, a *dynamic* array of outputs or inputs is declared. The space for each array member will be *dynamically* allocated by `control_*` *after* `$instbrick` when `$instconn` is executed.

NOTICE: addressing of array members via the syntax `@:>arrayname[`*index*`].fieldname` works only for fixed arrays! Also, notice that `alias` and `wire` statements (see section 6.2) will not work for dynamic arrays.

HINT: address calculation, total overhead and processor cache pollution (i.e. working set behaviour) is better for *fixed* arrays in many cases. Whenever a fixed bound for the number of array members is known, please prefer fixed arrays. Use dynamic arrays only if you *really* know that an unlimited number may occur!

3.4 `use` statements

After an `input` statement, a list of `use` statements may follow. Their specific syntax and purpose is described in the Programming Guide. Currently, some commodity library routines like transparent access to data blocks (Pointer Cache PC), cyclic doubly-linked ring lists with $O(1)$ element removal (LIST), and hashes (HASH) are planned/implemented. A `use` statement is always terminated by a semicolon.

3.5 `section` statements

May be optionally used after output statements to switch to operations on the meta-nest, the operation nest, or the operation-meta-nest of the currently active output. These nests are distinguished by numbers. The interpretation of the numbers is currently not fully fixed, but preliminary use it as follows: 0 (the default)is used for ordinary data nests, 1 for meta nests in the filesystem.

`section (:`*sec_nr*`:)`

Switches both the *default* section number for `operation` statements als well as the default for `@=call` statements. The default is only used if you omit the section number in a specifier. You may always override any default section number by explicitly specifying it.

By specifying the reserved word `ALL` for *sec_nr*, the following `operation` declarations will (by default) be automatically assigned to all sections, without producing code bloat. Inside such a multi-section operation, you may access the actual section number via `@sect_code` at runtime. When a section specifier for the target of a `@=call` statement is `ALL` (either by default inheritance, or by explicitly specifying `ALL`), the `@sect_code` from the caller is forwarded to the callee.

3.6 `operation` statements

Output statements may be followed by operation statements declaring elementary operations as described in the Programming Guide (and in some architecture papers).

`operation $`*op_name* `{`*code*`}`

op_name must be one of the official names as described in the Programming Guide[2]. In the *code*, the arguments can be accessed via `@`*arg_name*, where *arg_name* must be mentioned in the operation description (see Programming Guide[3]).

[2] In case of doubt (inconsistent description etc), please consult the Perl hashtable `%::op_args` in `pre.pl`.

[3] In case of doubt, consult the comma-separated lists in `%::op_args`. The first list tells the input parameter names of the operation, the second tells the output parameter names. When a third list is present, it tells the arguments which may be clobbered by that operation. Any arguments not mentioned in this list must not be clobbered (BUG: the current implementation violates this at some points! CHECK!). TODO: only permit output or clobber arguments as C lvalues! Currently not yet checked!

The *code* may also contain references to brick instance variables denoted `@#.`*brick_var* and to output vars `@:>.`*output_var* of the current `output` (if the operation belongs to an output) or an input variable `@:<.`*input_var* (if your operation belongs to an input).

Calls to elementary operations of inputs or other outputs may also be performed, as described in section 5.

When `$`*op_name* is prefixed by a section specifier `(:`*sect_nr*`:)`, the operation will belong to the specified section and not to the default section (e.g. as specified by a previous `section` statement). It is recommended to use a `section` statement for a series of `operation` statements instead of individually specifying the section of each one.

When `$`*op_name* is prefixed by the section specifier `(:ALL:)`, that operation implementation will be assigned to *all* sections. However, in addition to specifying `operation $`*op_name*`(:ALL:)`, you may override a single section with ad different implementation like `operation $`*op_name*`(:0:)` or similar.

4 Specifiers

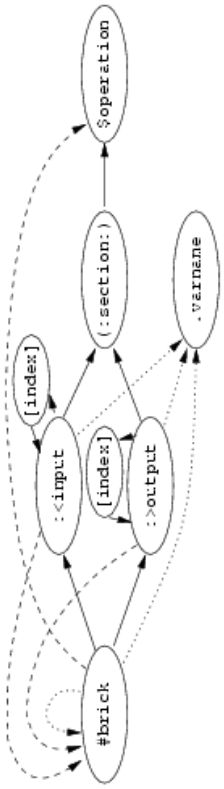
Basic specifiers have been already used in the preceding description. Bricks are denoted `#`*brick_name*, inputs `:<`*input_name*, outputs `:>`*output_name*, sections `(:`*sect_nr*`:)`, and operations `$`*op_name*.

Full specifiers (in contrast to basic specifiers) may comprise multiple basic specifiers pasted together in *ascending* order (starting from brick to specifier), *without any whitespace* between them. A full operation specifier may e.g. be denoted `#`*brick_name*`:>`*output_name*`(:`*sec_nr*`:)$`*op_name*. Similarly, full section specifiers contain a basic brick specifier, basic output and basic section specifier. Full input and output specifiers are preceded by a basic brick specifier, and brick specifiers always stand for their own.

If any prefix of a full specifier is missing, the current context is automatically filled in. For example, if a specifier `$get(:1:)` is used in the context of operation `#test_brick:>out(:0:)$put`, the brick and output part is automatically added to yield the full specifier `#test_brick:>out(:1:)$get`.

4.1 Proposed New General Specifiers

The following general specifier syntax is not yet fully implemented. Dashed lines denote currently non-implemented extensions. Dotted lines denoted a proposed new syntax which is different from prior versions, but already parsed syntactically correct at least in some cases:



The following special cases will be treated uniformly:

- An empty `#brick` part (denoted simply # without a brick name) means the current brick name.
- An empty `:<input` part (denoted simply `:<`) means the current input name where the specifier occurs (current scope)
- Dito for empty `:>output`.

In future, abbreviated specifiers not starting with # will be interpreted in the following way: first, the specifier of the current context scope is prepended. If that does not yield a

valid specifier syntax, the last part of the current context specifier is stripped, and the test is recursively tried again with a shorter scope specifier until either a possible completion with a shorter scope specifier is found, or until no possible completion has been found at all. This is similar in style to shadowing of identifiers in scoped programming languages: always take the outmost possible definition of an identifier.

IMPORTANT: the new specifier syntax requires changes to the old `@#brickvar`, `@<inputvar` and `@>outputvar` syntax. It is now denoted `@#.brickvar`, `@:<.inputvar` and `@:>.outputvar`, respectively. Currently it works only for abbreviated brick, input and output parts, but full variable path specifiers (even for accessing variables of nested brick instances) will be introduced soon (hopefully).

5 Call Syntax

5.1 Basic Syntax

Other output operations are called via the following basic syntax:
`@=outputcall` *op_specifier* `(`*in_args*`) => (`*out_args*`)`

where *in_args* is a comma-separated list of expressions which are assigned to the formal input parameters of the operation in the same order as described in the Perl hashtable `%::op_args` (see `pre.pl`), and *out_args* is a comma-separated list of C lvalues which receive the results.

The boolean output parameter `@success` exists at each operation. It denotes success or failure of the called operation. Prior to actually calling, it is initialized to `false`. If the callee does not explicity set it to `true`, it will remain `false`. Thus failure of an operation may be simply indicated by prematurely exiting it (e.g. via normal C `return`) before `@success` has been set to `true`.

Please insert a blank before the first argument parenthesis, because the *op_specifier* may also contain a paren at the section part. You should do that anyway to increase readability.

The *op_specifier* may contain a runtime-evaluatable expression at the section part. In such a case, the target nest is computed at runtime. TODO: allow computable operation names, probably even computable connector names.

There is another short form of the call syntax:
`@=outputcall` *op_specifier args_pointer*

where *args_pointer* must be an expression of type `struct args*` (see file `common.h`). There is one standard parameter of each operation named `@args`, which is the default argument buffer for that operation. You may directly use `@args` in place of *args_pointer*, leading to a shortcut argument passing. The advantage of this method is that no new argument buffer needs to be constructed on the stack (and finally destructed) as is necessary with the full calling syntax. The shortcut syntax is thus much more efficient. You can access the default parameter `@success` equivalently via `@args->success`. When declaring own argument buffers of type `struct args`, you cannot use the `@` notation but have to access to the fields as in conventional C.

When a section specifier for the target of a `@=call` statement is `ALL` (either by default inheritance, or by explicitly specifying `ALL`), the `@sect_code` from the caller is forwarded to the callee.

Instead of `@=outputcall`, the short form `@=call` may be used synonymously for both call variants; most calls will be `@=outputcalls` in practice.

5.2 Extended Syntax

Both call syntaxes may be extended by an optional `:` *arg* after the parameter list (or after the arg in short form syntax), which will be replaced for the default `@param` string argument of the called operations. By default, the `@param` of the caller is just passed through.

5.3 Mandates

As explained in the monography, *mandates* are general-purpose descriptors denoting *owners* of resources. Mandates may be *transferred* among brick instances. Currently, mandates are only used for locks (but this may soon change!).

After the *op_specifier* of the default call syntax, an optional bracket expression `[`*mandate*`]` may be added. It tells the callee under which mandate the operation should be executed. When omitted at the long call syntax, the default mandate `@#._mand` of the caller instance is automatically used by default. However, when omitted at the short call syntax, the `@mandate` parameter may be uninitialized when you forget to initialize it explicitly. When you just forward `@args`, the old mandate value supplied by your caller will be forwarded, which may be just your intended behaviour.

Be sure to check whether you want an operation to act under the mandate of your own brick instance or under a foreign mandate.

DISCUSS: these special-case syntaxes are irregular and should be replaced by a more systematic syntax!

5.4 Calls of Input Operations

Operations defined on inputs (such as `$input_init` or `$retract`) may be called by denoting `@=inputcall` instead of `@=outputcall`. This results in forwarding of calls *against* the ordinary direction of wires; when multiple inputs are connected to a single output, the specified operation is called at *each* of those inputs. For more details, see the Programming Guide.

6 Nested Brick Instances

Locally nested brick instances can be inserted into the enclosing brick instance via the following syntax (after the brick definition, but before defining inputs and outputs):
`instance` *brick_type* `as` *instance_name* `;`

You may create multiple instances of the same type, but they must have distinct names.

The Inputs and Outputs of a nested instance can be accessed via the following extended specifier syntax: *#mybrick_name#instance_name* or *##instance_name* for short. You may directly call an operation on an output or input of the nested instance just by using the extended specifier syntax. When calling an operation of the nested instance from code of the outer level, a slightly more efficient *direct* procedure call is produced by the preprocessor instead of an *indirect* call. However, the converse ist currently not optimized, because we don't generate code for a new version of the inner instance where the indirect calls could be replaced by direct ones. Doing that could easily lead to *code bloat* and is expected to be counter-productive in *most cases* due to processor cache pollution!

TODO / DISCUSS: by introduction of keywords `specialized` and `macro`, some (extremely tiny!) operations could be marked for inline expansion when they are used as an inner instance.

6.1 Wiring of Plain Sub-Instance Connectors

This section treats only aliases and wires for non-array inputs/outputs.

Instead of explicitly calling the operations of a nested output, you can generate an *alias* for a nested output such that it *directly* appears as an external output of the enclosing instance:
`alias` *#mybrick_name#instance_name*`:>`*sub_output* `as` *#mybrick_name*`:>`*my_output* `;`

or shortform

`alias :>`*sub_output* `as :>`*my_output* `;`

In the shortform, missing parts of the first specifier are always automatically completed with the nested sub-instance name, and the second specifier is completed with the enclosing brick name. Please prefer to create an alias wire whenever possible, because it does not consume any ressources, it just creates an alias-specifier which makes the inner output to appear as an output of the enclosing instance.

Analogously, if you want a nested input to appear as an externally accessible input of the enclosing instance, you can do that also:

`alias` *#mybrick_name#instance_name*`:<`*sub_input* `as` *#mybrick_name*`:<`*my_input* `;`

or shortform

`alias :<`*sub_input* `as :<`*my_input* `;`

In case of inputs, you cannot intercept any operation calls produced by the local instance, because it is directly forwarded to the outside as if it had been called on an ordinary input. If you want to intercept calls, you can do so by declaring a local output:

`local output :>`*my_localname* `{} {}`

This produces an output in the enclosing instance which is however not visible from the outside, i.e. you cannot call `$connect` at it at the strategy level. Now you can implement your operations on the local output *my_name* and redirect the nested input to your local output via the following statement:

`wire :<`*sub_input* `as :>`*my_localname* `;`

The keyword `wire` indicates that one of the specifiers is an input while the other is an output. The result is the same as if a `control_*` had performed a wiring operation; however this is more efficient because the wire connection is created *locally* at instantiation time of the enclosing instance.

Note that in this case, we currently re-use the indirect procedure calls of the nested instance and just redirect them to the outer code.

Of course, the same kind of redirection will also works with non-local (visible) outputs and with local outputs of (other) nested instances.

You can also create local inputs by prefixing the keyword `input` with the keyword `local`. However, the rules for connecting are slightly different: while a visible input must not be forwarded to any other (local or nested) output (since a connection can only be made by `control_*`), a local input *must* always be wired locally (since there is no `control_*` which could create a connection). Currently, operation calls on local inputs are always generated as *indirect* procedure calls; when you want more efficient direct calls, just directly call the wired output of the nested instance.

You can even directly `wire` an input of a sub-instance to an output of another sub-instance.

TODO/DISCUSS: allow non-local outputs to be additionally wired internally such that parallel wiring of both internal and external wires may occur.

WARNING: `$init` of nested instances is *not* automatically called at instantiation time. You have to do that "by hand". A good place is from some `$init` implementation at the outer level.

WARNING: be sure that you connect *all* the inputs of the sub-instances (whether to a local instance or to the outside). Otherwise they will never be wired. Calling an operation on an unwired input will crash!

6.2 Wiring of Arrays of Connectors

When your sub-instance has fixed arrays of inputs or outputs, you should read the following carefully. Dynamic arrays are generally not treatable by `alias` and `wire`.

Exporting arrays to the outside works only for *arrays as a whole*. You have to explicitly denote this case by empty brackets as in the following examples:

`alias :<`*sub_input_array*`[] as :<`*exported_array*`[] ;`
`alias :>`*sub_output_array*`[] as :>`*exported_array*`[] ;`

Analogously, you can create internal bunches of wires to local arrays of connectors as follows:
`wire :<`*sub_input_array*`[] as :>`*local_array*`[] ;`
`wire :>`*sub_output_array*`[] as :<`*local_array*`[] ;`

However ensure that your local array declaration has the same number of elements. Otherwise you will get a crash at instantiation time. TODO: automatically check bounds by the preprocessor or compiler, even in presence of arbitrary constant expressions for the array sizes!

You can also create a wire for a *single* array member of the sub-instance to a non-array local connector. This is denoted as follows:
`wire :<`*sub_input_array*`[`*17*`] as :>`*local_output* `;`
`wire :>`*sub_output_array*`[`*const-expr*`] as :<`*local_input* `;`

Be sure to connect all array members of the sub-instance on which an operation could (potentially) be called. Otherwise you will get a crash.

Wires to individual array memebers of local connectors are also possible. Be sure to connect any of the array members somehow.

7 Macro processor

The ATHOMUX preprocessor comes with its own macro processor, independent from the conventional C preprocessor. It should be both more comfortable and more capable.

New macros are defined via one of the following syntaxes:
`@.define` *macro_name* `(`*formal_inparams*`) {`*code*`}`
`@.define` *macro_name* `(`*formal_inparams*`) => (`*formal_outparams*`) {`*code*`}`

The formal parameters is either a comma-separated list of names, or a list of pairs *type_spec name* where *type_spec* is an ordinary C type-expression. When types are present, the macro expansion will later behave differently: the actual argument is assigned to a temporary variable of type *type_spec* and thus evaluated only once, regardless how often it is used inside *code*. This is much similar to `inline` functions in C and different from the C preprocessor. When *type_spec*s are omitted, the actual argument is evaluated at each occurrence in *code*.

Instead of writing the *code* in braces, you may write it in parens instead. In this case, the whole macro should be a comma-expression, and should be called in place of an expression. A third variant is `({`*code*`})` which acts also as an expression, but contains statements (see docs for proprietary extensions of the GNU C compiler).

Attention: the expansion can be used only in place of statements or expressions inside other code, because it opens a new lexical scope. When you want to write macros containing other preprocessor instructions such as `section` or whole `operation`s, you have to use `@.macro` instead of `@.define` as explained later.

Macros are simply called via *macro_name* `(`*actual_inparams*`)` or *macro_name* `(`*actual_inparams*`) => (`*actual_outparams*`)`.

Note: the *macro_name* may be a whitespace-separated list of multiple names, where the first may start with the special characters `@` and `@.` as well as `@=.` This way, even the `@=call` syntax can be emulated by the macro processor.

`@.undef (`*macro_name*`)`

This removes the defined *macro_name* (if it was defined). When *macro_name* was not actually defined, nothing happens.

`@.isdef (`*macro_name*`)`

When *macro_name* is currently defined, yield the text `1` else `0`. Useful in combination with `@.compute` and `@.if`.

7.1 Local operations

These are variants of the macro syntax, just by saying `@.func` instead of `@.define`. Instead of expanding a macro at each call, a local function is generated with standard arguments `@args` and additional arguments as specified by the formal parameters (where type specifiers are mandatory for proper generation of function prototypes). At each call, the `@args` of the caller is tranferred to the callee implicitly and automatically, to allow access to *@arg_name* like with ordinary macro expansion. In comparison to macros, code bloat is avoided. However, there are subtle semantic differences: when `return` is executed inside a macro, the *callee* is abandoned, because the code is inserted inline into the callee. In contrast, `return` inside `@.func` only exits from that local function.

DISCUSS: should the same `return` behaviour be implemented with macros, to make `@.define` a true drop-in replacement for `@.func` and vice versa? (the syntax is already the same) Then existing code must be revised.

7.2 Generic macros

When you want to place other preprocessor constructs inside a macro, you have to use `@.macro` in place of `@.define`. The disctinction is necessary, because expansion of a `@.define` macro generates a new C scope in braces or parens with correct lexical nesting of identifiers. The generic `@.macro` variant omits the braces and can thus be used in places where preprocessor constructs such as `input`, `output`, `section`, `operation`, `attr` and so on are to be generated by macros.

You *should not* use type specifiers for parameters of generic `@.macro`s, since there may be no scope or the wrong scope where argument expression evaluation can be bound to. When you do anyway, be sure that you know what you are doing, and check the generated code.

7.3 Include Files / C preprocessor

You may include any other file via the following syntax:
`@.include "`*filename*`"`

The following directive will first feed the *text* through the C preprocessor before expanding it again in the Athomux preprocessor:
`@.cpp {`*text*`}`

Note that the result of the C preprocessor expansion is expanded once again by the Athomux macro processor (usually without causing harm).

7.4 Stringification

You may convert any sequence of characters to a C string via the following directive:
`@.string{`*text*`}`

The *text* will be first expanded for further macro occurrences. The result is then surrounded by quotation marks. Internal quotation marks from the *text* will be escaped by a backslash. Percent signs will be doubled, and backslashes be escaped by another backslash.

7.5 Expression Evaluation

Computation of arbitray Perl expressions is supported by the following directive:
`@.compute{`*expr*`}`

First, the *expr* is expanded by the macro processor. The resulting text is then interpreted by Perl `eval()`. The final result must be a Perl scalar, either a string or a number, according to the Perl rules.
`@.subst (`*regular_expression*`) {`*text*`}`

The *text* will be processed with the *regular_expression* via Perl's =~ operator. Note that *regular_expression* is not expanded, but the substituted text will be further processed.

7.6 Conditional Expansion

`@.if (`*expr1*`) {`*text1*`} @.elsif (`*expr2*`) {`*text2*`} @.else {`*text3*`}`

First, *expr1* is expanded in the current macro scope and then interpreted by Perl `eval()`. When it results in a non-empty string or a non-null number (according to the Perl rules), the *text1* (without braces) will be the result of the expansion of the whole statement. Otherwise, the optional `@.elsif` or `@.else` parts (if any) will be expanded according to the usual rules of modern programming languages, as you will expect.

7.7 Lexical Scoping

`@.scope {`*text*`}`

This will treat the *text* in a new sub-scope. Any definitions inside *text* will be lost after closing the scope.

By default, any macro is also expanded in a new scope. When you define a (parameterized) macro inside another macro, the definition will not be visible from the outside.

The default behaviour of `@.define` and `@.macro` may be changed by the following extension:

`@.define[`*option_list*`] (`*param_list*`) {`*body*`}`

where *option_list* is a comma-separated list of one of the following option names:

`flat` When the macro is expanded later, the expansion will occur in the scope of the caller. In particular, any sub-definitions from the *body* will remain valid.

`preexpand` Before defining the maco, the body is expanded in the current scope and expansion mode. At substitution time, it will be expanded once again by default.

`prescope` Do the `preexpand` in its own scope (i.e. discard any macro definitions resulting from it).

`postprotect` When the maco is expanded later, the body will not be expanded. However, ordinary arguments will be substituted. This is often useful in combination with `preexpand`.

DISCUSS: anyone needing a *full* equivalent of `\gdef` from TEX?

7.8 Expansion Order

Although the ATHOMUX macro processor does not aim in sophistication, it can control the expansion order to some degree.

Normally, the macro processor does *deep expansion* of macros in the following way: Whenever a macro is found, its body will also be expanded. Endless recursion is avoided by allowing any macro to occur in an expansion at most *once*; trying to expand an endless recursion will simply not work.

If you want to expand something *exactly once*, you can do so by saying

`@.step` *further_text*

`@.step {`*expanded_text*`}` *following_text*

In the first form, the *further_text* will be expanded exactly once and only at the beginning, i.e. the next token *must* belong to a macro call; later macro invocations will not be expanded. In the second form, the expansion may take place only at *expanded_text*; the *following_text* is treated the same way as if `@.step` was not present at all (depending on the calling context).

`@.expand` *further_text*
`@.expand {`*expanded_text*`}` *following_text*

Any occurrences of any macro will be expanded anywhere in *further_text* resp. *expanded_text*, but each expansion will be exactly one level (not deep).

By placing further `@.expand` directives into the *expanded_text*, you may expand more times, but you will have to mark those places explicitly with `@.expand` each time.

`@.deep` *further_text*
`@.deep {`*expanded_text*`}` *following_text*

You may use this for switching back to deep expansion when you are in (possibly repeated) single expansion mode.

To protect against expansion, you may use the following forms:

`@.protect` *further_text*
`@.protect {`*protected_text*`}` *following_text*

The *further_text* resp. *protected_text* will not be expanded at all, i.e. copied verbatim. This will work even in case of deep expansion (which is the default). However notice the difference between *deep expansion* and *repeated expansion*: when you nest two `@.expand{}` around some text, the text will be expanded exactly twice; if there is some `@.protect` in it, it will be removed the first time, and thus be expanded upon the second *repeated* evaluation.

Instead of nesting arbitrary numbers of `@.expand` or of another evaluation order directive, you may use the following abbreviated syntax:

`@.expand(`*count*`)` *further_text*
`@.expand(`*count*`) {`*expanded_text*`}` *following_text*

The *count* must be a numeric constant, interpretable by Perl. It also works with `@.step`, `@.protect` and `@.deep`.

`@.copy(`*count*`)` *further_text*
`@.copy(`*count*`) {`*expanded_text*`}` *following_text*

The *further_text* resp. *expanded_text* will be expanded in the current expansion mode. The result is then textually repeated *count* times. This is different from `@.expand(`*count*`)` because the *expansion itself* is repeated exactly once, while the *result* will be repeated *count*-fold. When *count* == 0, the result of the expansion is discarded. This is useful for catching the side effects of the expansion, e.g. further macro definitions inside *expanded_text*. Here is an example for transferring a C preprocessor definition into the Athomux preprocessor:

```
@.copy(0){@.cpp{
#include <stdio.h>
@.define my_bufsiz (BUFSIZ)
}}
```

`@.shuffle [`*number_list*`] {`*body0*`} {`*body1*`}` ...

Use this to change the order among the bodies. First, all the bodies are expanded in their original order, but not copied to the output. The *number_list* must be a Perl-interpretable comma-separated list of numbers. Each number n in the list indexes the expanded *body*n; the result will be the expanded *body*n in the given order. When the same number appears multiple times in *number_list*, the corresponding body will appear each time. When a body number does not appear at all, the body will not show up in the result.

`@.expshuffle [`*number_list*`] {`*body0*`} {`*body1*`}` ...

This does not change the order of the bodies, but determines the order in which each body will be expanded, similar to TEX's `\expandafter` (but for arbitrary number of bodies and expansion permutations). After individual expansion of each body in the given order (possibly even multiple times), all bodies (whether previously expanded or not) will appear in the original order. Usage is only recommended for TEX gurus who know what they do.

7.9 Parameter Expansion

At a macro definition, each parameter may be individually prefixed by an indicator of the parameter expansion mode:
`@:` Substitute the actual parameter *unmodfied*, i.e. don't expand the actual parameter before it is substituted. This is the default!
`@!:` First expand the actual parameter before it is substituted; the expansion is done in the old scope of the *caller*.

The *point in time* where a parameter is substituted can be controlled *independently* from the above:
`@<-:` First substitute the actual parameters into the macro body *before* expanding the body. As a consequence, the parameter will *only* be expanded in the scope of the callee, but not in (recursively) nested scopes (if you don't *explicitly* pass it on as a parameter, which is of course possible). This expansion rule is much like the *dynamic* scope rules of C, not of a classical macro processor. This is the default!

You may combine `@<-:` with `@!:` via the compound syntax `@!<-:` in order to evaluate the actual argument twice, once in the scope of the caller and once again in the scope of the callee (together with other macros).
`@->:` First expand the body of the macro, and only *after* that substitute the actual parameter. If you don't use the combination `@!->:`, the actual parameter will not be expanded at all before substitution (however you may force a re-expansion of the already substituted parameter by an external `@.deep(2)` around the whole macro call).
`@def:` define a parameterless macro at the scope of the callee (which will later substitute the formal parameter name with the actual parameter argument), then expand the macro body. This will not only result in *lazy* expansion, but it can also result in strange side effects: normally, expansion will carry over to nested macro calls (which is often the desired effect of "pumping in" parameters), but if anyone re-defines the same macro name inside nested macros (e.g. with another `@def:` on the same name), the old definition will be *shadowed* (i.e. obey the scope nesting, but get a totally different meaning), which might not be expected or could look strange (if you don't know what you are actually doing). To avoid some strange effects resulting therefrom, you should use the combination `@!def:` which will first expand the actual parameter at the *old* scope before potentially overwriting an old macro definition at the new scope (however notice that this results in eager evaluation).

WARNING: when using `@def:` in a `flat` macro expansion, the symbol will remain defined even *after* the flat expansion!

7.10 Computed Identifiers

You may create *computed identifiers* via the special syntax `@@`:
`name1@@`*name2*

where *name2* is an argument of your macro. Assume that `replace2` is the actual argument for *name2*, then you will get a compound identifier `name1replace2` where the `@@` is removed.

When the name created this way is itself appearing in another @@ context, the first @@ is expanded first. For example, *name1*`@@`*name2*`@@`*name3* will first concatenate the expansions of *name1* and *name2*, yielding a new identifier (which may itself expand to yet another identifier) before appending the expansion of *name3*. If you need a different expansion order, use `@.expand` (see previous subsection).

7.11 Misc Preprocessor Aids

The generic type system (see Programming Guide) often leads to tedious repetitions of `@*`*typename*`->` operators where always the same *typename* must be denoted. For better

convenience and better readable code, you may use the following shortcut writing:
`@.deftype` *typename* `{` *body including* `@*->` *operators ...* `}`

After expanding the *body* in the current expansion mode, all occurrences of `@*->` in the expanded *body* will be replaced with `@*`*typename*`->`.

8 Additions

8.1 Avoiding code bloat

Multiple operations sharing the same *code* may be declared by
`operation $`*op1*`,$`*op2*`,$`*op3*

This avoids code duplication in the `*.ath` sourcecode. The following predefined placeholders may be used inside such common code:
`BRICK_NAME` is substituted with the current brick name
`CONN_NAME` is substituted with the current input or output connector name
`SECT_NAME` is substituted with the current section number
`OP_NAME` is substitued with the current operation name

When `OP_NAME` is not used inside the body, only one single C function is generated which is shared by all operations from the operation specifier list. When `OP_NAME` is present, a separate version with different subsitute values for each *op1*, *op2* and so on is generated.

Advice: try to avoid these predefined low-level substitutions by using `$op` instead (which avoids code bloat).

9 Not yet implemented

A lot of stuff....

9.1 Variants

Proposed syntax:
`variant` *attrib_name value1*`,`*value2*`,`*value3*`,`...

By invoking `pre.pl` with a parameter *attrib_name*`=`*value*, specialized code is generated as if the attribute *attrib_name* had only that one value. The first value is treated as default value when the invocation parameter is omitted.

Conditional code generation should also be permitted. Proposed syntax:
`??`*attrib_name* `==` *value* `{`*code*`}`

Probably further variants like != or set inclusion relations could be added later. Usage of the attribute value in ordinary C code must also be possible, e.g. by the syntax `@?`*attrib_name*.

9.2 Provide / Require

In addition to static and dynamic brick/input/output attributes, functional and non-functional requirements and provisions should be easily denotable and automatically checkable by `control`. Details have to be worked out.

9.3 Generic Types

The current generic type system should be revised and improved.

10 Philosophy

The architectural philosophy of ATHOMUX is explained in more depth in the German monography, in some English papers, and in diverse presentations. These are for an academic audience. Here is a quick abstract with emphasis on hacker's interests:

Please forget anything you know on object oriented design, and forget many things (but not all!) you know on operating system architecture, in particular on how to build higher-level abstractions. ATHOMUX is almost the converse of OO, and very different from classical OS kernel architectures. It does not fit in such categories like "microkernel" or "exokernel" or "monolithical kernel", because it can be *configured* to simulate any of those models. If you need a headline, call it an *instance-oriented* operating system architecture.

ATHOMUX does anything with two basic abstractions: nests and bricks. The nest is a universally generic address space abstraction. If *necessary*, higher-level abstractions *may* be built *on top of* nests, but if possible try to avoid this and remain at the universally generic level as long as possible! When you have to raise to a higher level of abstraction, do this *as late as possible*, and remain universally generic *as long as possible*.

Instead of creating new abstractions, create *new functionality*!

Create new functionality *either* inside brick implementations (i.e. use bricks as black-box wrappers and *reuse* the universally generic nest interface), *or* (even better) create it by *composing* brick instances to new brick network configurations (aka compositorical genericity, i.e. *reuse* existing components). This is similar to using the Unix shell for creating pipelines of filter instances, such as `grep`, `awk`, `cut`, `Perl` etc. Unix pipes are universally generic, by allowing *any* stream of bytes flowing through them, and by providing a *uniform interface* for all filters (aka "pipe and filters style" in software architecture). Good filters are universally generic, by processing a large class of possible data formats. The shell creates compositorical genericity by allowing nearly arbitrary combinations of filter instances. However, note that nests are slightly higher-level than Unix pipes and even more generic for a much larger class of applications (the pipes are mentioned only as examples of a *similar* architectural style, not to explain nests!).

When you need a metaphor from real life, call ATHOMUX a LEGO-like brick system. Current mainstream OS technology is at a stage where each part is manufactored *individually*, with indidual interfaces, leading to high development costs and an inflexible design. The next step after that is using *components*: According to Szyperski, a component is characterized by explicitly interfacing between *anoymous* software components which sometimes even don't know with which partner (from which manufacturer) they are interacting. However, components are often built according to the metaphor of a *puzzle*: a single piece may be replaced by another one having the same shape, but there exist a lot of different interface shapes.

ATHOMUX is already the next generation *after* component software: by using a *uniform interface type* (similar to the LEGO principle of using always the same interface tenon even for very different brick types), there are far more compositions than with current component software. While component software (e.g. plugin architectures) deals with composition of *program code*, ATHOMUX deals with composition of (a potentially unlimited number of) *runtime instances*.

Another fundamental difference of the ATHOMUX architecture to other architectures (in particular OO) is *statelessness* or *pseudo-statelessness* of brick instances. Of course, it is possible to implement bricks as stateful ones. However, statelessness will ease reconfiguration, fault tolerance, migration, network transparency, and much more by treating state *explicitly*. Statelessness means simply that state is kept in the inputs of a brick instance, not in the instance itself. State keeping is *delegated* to other instances, until finally to the hardware. Explicit state has the advantage that you can copy and migrate it!

ATHOMUX may be labelled an "instance oriented operating system". What is instance orientation?

Composition of brick instances to networks is regarded as a basic building principle,

similar to LEGO brick systems in the area of toys or other LEGO-like mechanical systems (e.g. look at the mechanical / pneumatical components of the German engineering company Festo, and others like Bosch-Rexroth motion kits). We deal with composition networks *explicitly*. Wired networks of bricks are described by nest instances, recursively. This is done at a separate level called *strategy level*. There you can do *transformations* on brick networks, e.g. create new *views* on brick networks or even *virtual brick networks*. That's the basic idea of instance orientation.

Some examples for such transformations may be found in the paper on merging operating systems and databases. A simple example is creation of *location transparency* in a network of computers: a transformation on the strategy level may automatically insert `remote` and `mirror` bricks wherever necessary, and provide a view where the concrete location of brick instances is hidden. You can get a true *distributed operating system* (in contrast to a network OS) with that, but when orthogonally combined with database functionality, you get something no word has been coined for yet.

The *far-distant* goal is thus: *beyond* Unix and current mainstream OSes, *far beyond!* The architecture of ATHOMUX aims to provide the construction principles for the next 30 years, as a successor of the currently established OS building principles which have been successfully used for more than 30 years now, and have been extended and balconized many time (but showing up their limits more and more often). ATHOMUX is thus starting from scratch. Its native interface to applications could and should be different from other OS interfaces. Only for the sake of compatibility, in particular with Unix / Linux (but not limited to that), *personalities* shall be added, based on *adaptors*. Such adaptors should be principally *optional configurations*, interoperable with each other.

At current stage, ATHOMUX is at its very infancy. Like any OS, it will require many years and much effort until it becomes mature enough to compete with current mainstream technology. It will become valuable if people become interested and intrigued by its *potential*.

If you can see the potential very far *beyond* current OS architectures, please come and join its delevopment!

A Appendix: Predefined Names

`@args` The default arguments of an operation, of type `struct args`. The members may be accessed via the syntax *@member*.

`@param` The default parameter string

`@#._mand` The default mandate number of the current brick instance. Default numbers are automatically generated at brick instantiation time.