Reference manual of SynDEx code generation

Methodology and tools for generic code generation

Yves Sorel, Simon Nivault
INRIA Rocquencourt,
BP 105 - 78153 Le Chesnay Cedex, France
yves.sorel@inria.fr
www.syndex.org

Contents

Introduction

This document aims to explain how we generate code from files produced by SynDEx. It will describe the methods, tools and libraries used. This documentation targets anyone who expects to generate code using SynDEx and who wants to know more about how the code generation works or wishes to upgrade existing tool to generate code more in line with its needs. This document also targets SynDEx developers as it is a "code generation" equivalent of the SynDEx reference manual.

1 General principles

In the world of software development, mainly in the industrial context, the traditional development model is the V-model. The V materializes a Specifications / Modeling downward slope to the development itself and an upward slope with tests and validation. The existence of this upward slope is due to the fact that the development is done by humans and may be incorrect. it is necessary to reconcile with hindsight development with specifications and modeling to obtain a consistent and error-free application.

The purpose of SynDEx is to transform this V-model into a I-model by code generation. The idea is to ensure by construction the consistency between specifications and development. It improves the safety and the speed of the development, the I-model being shorter than the V-model.

SynDEx performs an adequation between the twooriginal specifications, the functional specification, also called algorithm, and hardware specification, also called architecture. From this adequation, SynDEx produce at first time what is called a distributed[1] and heterogeneous[2]generic macro-code. This macro-code is specific with regards to the algorithm and the architecure for which it was produced, but it is generic with regards to the various methods that we could use to carry out this program if it had the will to write manually.

In a second time, and this is precisely the purpose of this document, there is a compilation chain that can transform the SynDEx generated macro-code to a specific code chosen by the user. Indeed, the choice of the target language is left to the user, the objective of being able to integrate SynDEx with its own development chain.

The figure 1 gives an overview of this file transformation. The black arrows represent the act of producing a file. Dashed red arrows mean "is included in/by".

SynDEx produce a macro-code, which is made up of a set of m4 files (`.m4` extension). The macro-code is called as it is because it is intended to be processed by m4, which is a macro-processor. This set of files is composed by an m4 file describing the architecture (applicative macro-code) named `appli.m4`, `appli` being the name of the application, and by one m4 file per processor of this architecture named `proc.m4`, `proc` being the name of the processor, describing the program that will be executed by the processor previously mentioned (executive macro-code). Thus, SynDEx generate two type of macro-code, an applicative one and an executive one. From an

[1]The architecture can be composed of several calculation operators.

[2]The different calculation and communication operators of the specified architecture can be differently typed.

initial Makefile written by the user, the macro-processor m4 will transform the applicative macro-code into an helper makefile, specifically generated to describe how to transform and compile the executive macro-code.

This transformation of files uses m4 files (`.m4x` and `.m4m` extension) where we define the macros contained in the macro-codes. These m4 files are called executive kernels. these ones can be classified among two axis :

- one axis related to which type of macro-code is concerned by the executive kernel,
- one axis related to the genericity of the executive kernel,

Among the first axis, one has:

- the executive kernels describing the macros used in the applicative macro-code (makefile generation, `.m4m` extension),
- the executive kernels describing the macros used in the executive macro-code (code generation, `.m4x` extension),

Among the second axis, one has:

- a generic executive kernel which is the entry point of the other executive kernels, called `syndex.m4x` and `syndex.m4m`,
- several executive kernels specific to the architecture which describe the way one will generate the code and the context where it will be executed (description of the language, of the OS, of the system libraries, etc), called `Archi.m4x` and `Archi.m4m`, `Archi` being the name of the used architecture,
- several executive kernels specific to the application called `Appli.m4x` and `Appli.m4m`, `Appli` being the name of the application, which mainly contain the definition of the macro corresponding to the domain specific code. These definitions are usually specifically written for a given application but this work can virtually be factorized by creating applicative executive kernel libraries (some are distributed with SynDEx). Besides, these executive kernels can contain some general execution options (number of itération of the application, names of the hosts where the application will be executed in case of distributed application), some specific options to tune the behavior of other executive kernels, or whatever.

Outline The main tool is the macro-processor m4. Thus, the first section will talk about it and will present its general principles and its most used features. The reader looking for details is invited to look deeper in the user manual, available on the web[3]. The next section will talk about the transformation itself. This transformation happens in two step, both based on m4. The first step is the generation of the build system (Makefiles), the second one is the code generation followed by his compilation using the build system previously generated.

[3]http://www.gnu.org/software/m4/manual/

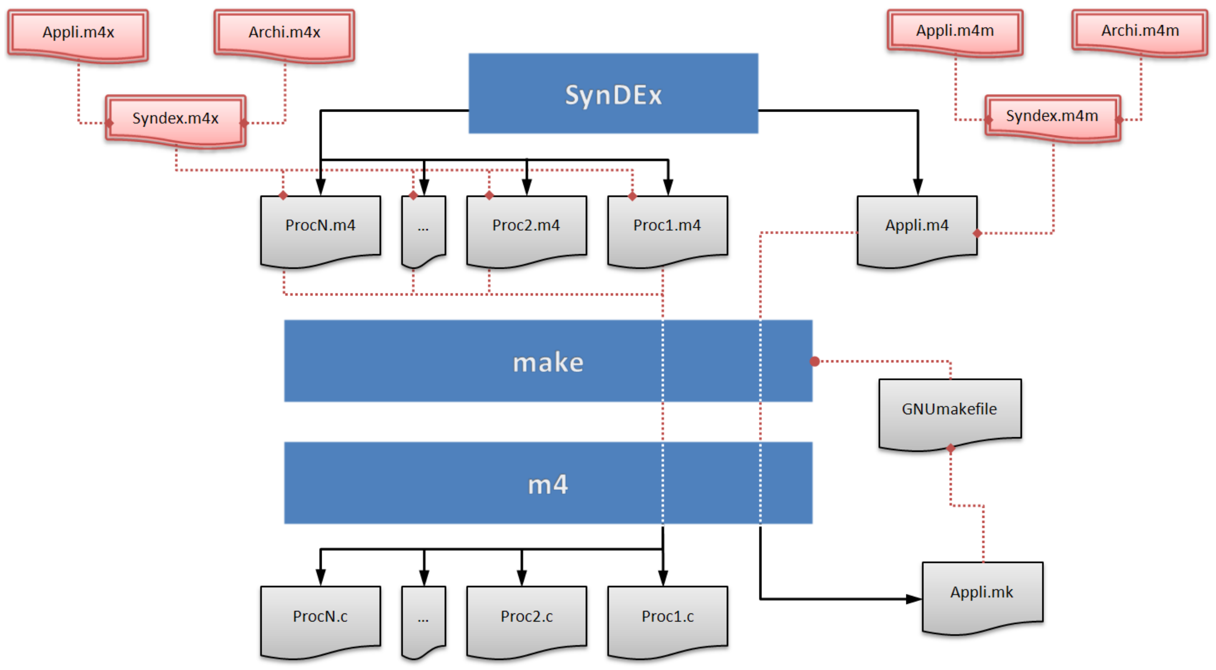

Figure 1: Overall view of code generation

2 Starring m4

Basically, m4 read a text from input and reproduce this text to the output with possible modifications. These ones are defined in the input text itself in the shape of macros.

m4 invocation:

```
$ echo foo | m4
foo
$
```

Here we call m4 giving "foo" on its standard input (line 1). m4 get "foo" and write back on its standard ouput (line 2).

m4 can get input from a file:

```
$ echo foo > test.m4
$ cat test.m4
foo
$ m4 test.m4
foo
$
```

Here, we first put "foo" into the file `test.m4`, then we call m4 giving this file in its parameters. Thus, instead of reading on its standard input, m4 reads from the file given in parameters and writes back again "foo" on its standard output.

Since in m4, all is based on macros, let's explain what they are.

2.1 Macros

An m4 macro is a definition which link the macro name with the replacement text. As soon as a macro is defined, every time m4 read this macro name, it replace the macro name with its replacement text.

Here is an example:

Entrée :

```
foo
define(foo, bar)
foo
```

Sortie :

```
foo

bar
```

On the first line, no macro has been defined, thus the word `foo` is reproduced to the output.

On the second line, the macro `foo` is defined. The text of the definition is deleted, thus the second output line is empty.

Finally, on the last line, the word `foo` matches the name of the macro previously defined and so it is replaced by its replacement text, here `bar`.

2.1.1 Parameters of macros

Macros can take parameters to benefit from more dynamic replacement.

Entrée :

```
define(OP1, var1)
define(OP2, var2)
define(add, $1 + $2)
add(OP1, OP2)
```

Sortie :

```



var1 + var2
```

Macro parameters are called by $1, $2, etc. These parameters are given to the macro between parenthesis and comma separated. You can notice that `define` is a macro itself and so it is replaced, here by the empty string.

Warning: Parenthesis must be immediatly after the macro name when the macro is called, else:

Entrée :

```
define(OP1, var1)
define(OP2, var2)
define(add, $1 + $2)
add (OP1, OP2)
```

Sortie :

```



 +  (var1, var2)
```

2.1.2 Quoting

The characters `(backquote) and '(quote) allow the quoted text to be reproduced without any modification, i.e without calling macro or taking into account special characters.

Here is an example without quoting, the result is not the expected one:

Entrée :

```
define(add, $1 + $2)
add 1,2 and 3:
add(1,2 , 3)
```

Sortie :

```

 +  1,2 and 3:
1 + 2
```

Here is how to have the expected result, using quotes:

Entrée :

```
define(add, $1 + $2)
`add 1,2 and 3:'
add(`1,2' , 3)
```

Sortie :

```

add 1,2 and 3:
1,2 + 3
```

Nevertheless, quotes can introduce unexpected behavior:

Entrée :

```
define(OP1, `1,2')
define(OP2, 3)
define(add, $1 + $2)
add(OP1, OP2)
```

Sortie :

```



1 + 2
```

To uderstand what happens, it's important to know how m4 read its input.

- First, m4 read word for word. Thus, in the example above, it begins to recognize the word `define`.
- Since `define` is a macro and the following character is an opening parenthesis, m4 will decide to read the following to fetch the comma separated parameters of the macro until it matches a closing parenthesis.
  - M4 reads the following word, here `OP1`.
  - M4 reads a comma, so it accepts `OP1` as first parameter.
  - M4 reads `, then it directly goes to match the corresponding ', then accepts the content as is.
  - M4 reads the closing parenthesis. It is ready to call the macro `define` with `OP1` and `1,2` as argument.

  Let's go directly to the last line:
- M4 reads the word `add` then the opening parenthesis then fetches parameters...
  - M4 reads the word `OP1` et matches the macro name. Since the macro name isn't followed by a parenthesis, the macro is called without parameters and so is replaced by `1,2`. M4 always processes again the result of a macro replacement, thus instead of matching the following comma:
  - M4 reads the word `1`, then the comma, and so the first parameter is `1`, and so on:

– The next parameters are `2` and `3`.
– The macro **`add`** is called `1` (resp. `2`) as first (resp. second) parameter. The third parameter is passed over.

To avoid these mistakes, there is a simple rule to remember: Always quotes one time the parameters for macro calls, thus these parameters are processed after the macro replacement.

Entrée :

```
define(`OP1', `1,2')
define(`OP2', `3')
define(`add', `$1 + $2')
add(`OP1', `OP2')
```

Sortie :

```



1,2 + 3
```

Last thing to know about macro calls: white spaces around parameters.

- White spaces placed after the parameter belong to it:

Entrée :

```
define(`add',`[$1]␣+␣[$2]')
add(OP1␣,OP2␣)
```

Sortie :

```

[OP1␣]␣+␣[OP2␣]
```

- White spaces placed before the parameter are passed over:

Entrée :

```
define(`add',`[$1]␣+␣[$2]')
add(␣OP1,␣OP2)
```

Sortie :

```

[OP1]␣+␣[OP2]
```

2.2 Predefined macros

As we can see, m4 is based on text replacement by macros. These macro can be defined by the user, using the macro **`define`**. However m4 owns lots of predefined macros that enable specific behavior. We won't take up these macro exhaustively, but only the most used ones in SynDEx.

2.2.1 include

The macro **`include`** take a file name in parameter give its contents to m4 to be processed immediately. Basicly, we often have files containing an include macro in the beginning of the file, followed by macro calls, defined in the included file.

Fichier foo.m4

```
This file is foo.m4
define(`foo', `bar')
```

Entrée :

```
include(`foo.m4')
foo`'foo`'e
```

Sortie :

```
This file is foo.m4


barbare
```

Notice that m4 search the given file in the current directory. But if the `M4PATH` environment variable is set, it is expected to contain a colon-separated list of directories, which will be searched in order. This features is used in SynDExbecause the files which define our macros are in the SynDExinstallation folder, so it is put in `M4PATH`.

2.2.2 dnl

We can see in the last exemple a troublesome effect. The two empty lines (line 2 and 3) aren't actually desired but are here for good reasons. Theses two empty lines corresponds to the two line feed after the macros `define` and `include`. Fortunately, there is a macro which is able to "eat" these line feed: `dnl`. This macro delete itself and all the following characters to the end of the line, including line feed.

Entrée :

```
Rewritten text dnl Forget me!
Hi!
```

Sortie :

```
Rewritten text Hi!
```

Or:

Fichier foo.m4

```
This file is foo.m4
define(`foo', `bar')dr
```

Entrée :

```
include(`foo.m4')dnl
foo`'foo`'e
```

Sortie :

```
This file is foo.m4
barbare
```

In the context of code generation, it's important to master the removing of these line feed. Otherwise, a lot of line feeds will remain which are as many fence to the readability of the generated code.

2.2.3 divert

The macro `divert` is one of the most complicated of m4. Indeed, this macro creates kinds of buffers where m4 is able to temporarily write instead of standard output. However, `divert` does not allow to redirect the standrad output towards a file or anything else, this is not his job (it's the job of a shell). `divert` take a number as parameter which select the buffer where m4 will write from now on. If the parameter is left out, it is defaulted to 0 which is the normal output stream. Consequently, when `divert` is called with a non zero parameter, the text that should be output won't be output anymore. Example:

Entrée :

```
Normal ouput.
divert(1)
Diverted output.
divert
Back.
```

Sortie :

```
Normal ouput.

Back.
```

As we can see, all between `divert(1)` and `divert` hasn't be output. The empty line corresponds to the line feed after the `divert`. Most of the time, this line feed is unwanted and this is achieved like that:

Entrée :

```
Normal ouput.
divert(1)
Diverted output.
divert`'
Back.
```

Sortie :

```
Normal ouput.
Back.
```

Actually what is written in a numbered buffer isn't lost. By contrast, this is stored to be used later thanks to the macro `undivert`. The exception is when a negative number is used, the text is discarded. We can use a negative divert to define our macro, thus we needn't `dnl`. Finally, unlike all other macros, m4 doesn't process again the result of the macro `undivert`.

Entrée :

```
Normal ouput.
divert(1)dnl
foo
divert(-1)
define(`foo',`bar')
divert`'dnl
Back.
undivert(1)
```

Sortie :

```
Normal ouput.
Back.
foo
```

3 Génération de code

SynDEx produce two kinds of m4 files:

- One file[4] named `APPLI.m4` relating to the application, `APPLI` being the application name.
- One file[5] per operator which contains the macro-code of this operator. We will name them `PROC1.m4`, `PROC2.m4`, ... `PROC1` and `PROC2` being the name of the operators of the architecture.

Each kind of files has a specific job. `APPLI.m4` is useful for the generation of the build system (makefiles) while `PROCN.m4` files are useful for the executive code generation. These files are composed of standard macro calls and these standard macros are defined in an m4 file, included in the beginning of the macro-code. Thus, the first line of `APPLI.m4` file is `include(syndex.m4m)` and the first line of `PROCN.m4` files is `include(syndex.m4x)`. `syndex.m4m` et `syndex.m4x` are the generic macro files of code generation. One of their main purpose is to include, in accordance with the content of `APPLI.m4` and `PROCN.m4` files, the application specific macro files.

3.1 Build system generation

In this section, we will talk a lot about Makefile, because the build system is composed of two Makefiles, one of them including the other. Despite that, we will not explain the concept of Makefiles and we will consider that this knowledge is owned.

[4]The Applicative macro-code.
[5]The executive macro-code

First of all, only one of these two Makefiles are generated. The other Makefile is the main one, is minimalist[6] and must be handwritten.

This main Makefile can be tuned in accordance with your needs but must respect some condition to be operational. Being the gateway towards the generated Makefile, it must contain:

- the include command of the generated Makefile. This one is named `APPLI.mk` by convention. (The `mk` is by convention the suffix for makefile helpers).

  This command is line 18, appendix A;

- One rule to make this makefile. For that purpose, we call `m4` on the file `APPLI.m4` and the outpus is stored in `APPLI.mk`.

  All these are line 16 and line 17, appendix A;

- A variale which tell to `m4` the path of specific macro definitions of SynDEx. This variable is named `M4PATH` and must contain a colon separated list of absolute paths where these definitions are (See the macro include at 2.2.1).

  All these are from line 5 to line 8, appendix A.

Finally, the generated makefile must have two phony target, `APPLI.all` and `APPLI.run`. `APPLI.all` has the purpose of generating and compiling code while `APPLI.run` has the purpose of running it.

To be able to call these command from the rules `all` and `run`, we add the lines from 11 to 14 in the appendix A.

Comment about application execution: Now, in several executive kernels, macros and generated code are written in such a way that the whole application can be lauched from a single command, even for distributed application. This mecanism is called the "downloader". Currently, this downloader is designed in such a way that it is very interlinked with the applicative source code of each operator. Thus the set of operators of the application make up a tree where each node is the executive code of an operator. Each node is supposed to be lauched by its parent and is supposed to launch its children. This mecanism has the advantage of being able to simply execute a distributed application from only one workstation. By constrast, it has two major drawbacks:

- the ability of an executable to launch another one on a remote machine is based on service which is not always supplied by the operating system. This is relatively easy on a network of Unix-based computer thanks to `ssh` but is still quite complicated about the `ssh` configuration. This decrease the impact of the advantage above. Besides it's even more complicated on system without `ssh` like Windows. It's possible to give Windows a Unix behavior using solution like CygWin, but this is too intrusive. But what about the application distributed on Unix and Windows in the same time ?

- the current implementation of the download is as well too intrusive with regard to the generated executable because there is only one site where the downloader code and the application code stands. Thus we can't part with the downloader to only use the core application. This drawback exacerbate the previous one because we can't part with the downloader when it is too hard to implement.

[6]An example of minimalist Makefile is given at the appendix A

3.1.1 APPLI.mk generation

As seen earlier, the file `APPLI.mk` is generated by `m4` from the file `APPLI.m4` which include `syndex.m4m`. So we will present the macros defined in `syndex.m4m` and used in `APPLI.m4`.

3.1.1.1 Macro `architecture_`:

This macro takes three parameters in the following order:

- application name,
- copyright,
- date and time of the file creation.

It generate the makefile rule `APPLI.run`, `APPLI` being replaced by the first parameter. The recipe of this rule is the concatenation of two variable `prefix.run` et `APPLI.root`.

`prefix.run` is defaulted to `./` which is the beginning of the command line to run an executable placed in the current directory. This variable exists to support other execution environment. For example, under Windows/RTX, an executable in a file with type RTSS which is supposed to be launched by RTSSRun. In that case, `prefix.run` must be `RTSSRun`, the RTX.m4m kernel executive is in charge of this setting.

3.1.1.2 Macro `processor_`:

The macro `processor_` take $2 + 2n$ parameters. The two first ones are respectively the type and the name of an operator. The pairs of parameters that follow are the type of a communication port of the operator and the name of this port respectively.

This macro appears one time for each operators belonging to the architecture for which we generate the application. Thus, this macro is supposed to generated the part of the makefile which will generate and compile the code associated to the operator.

Most of the time, we have one executable for each operator. This executable is added to the list of the dependencies of the target `APPLI.all`.

Then, in order to build the command line, we join the name of the executable and the address of the operator (hostname, IP, etc.) into the `APPLI.root` variable. Theses pieces of informations are convenient for the downloader (see comment page 11). When the main executable is supposed launch the other executables, it gets the name of these executable as well as the site where they are supposed to be launched from its command line. Besides, since these pieces of makefiles are written in the same order as the order of the different macros `processor_`, the macro of the root operator (which is supposed to launch the others) must be the first one.

Finally, one or more makefile target must be written to generate the executable from the file `PROC.m4`, `PROC` being the name of the operator. Most of the time, this is done in few steps:

- Transformation of the macro-code (`PROC.m4`) into source code (for example `PROC.c`).
- Compilation of the source code into object files (`PROC.c` $\rightarrow$ `PROC.o`).
- Link edition of the object file (`PROC.o`, alone or with others) into an executable (`PROC.exe`).

3.1.1.3 Macro `connect_` and `endarchitecture_` :

The macro `connect_` takes 2 + 2n parameters. The two first parameters are the type and the name of a medium respectively. The pairs of parameters that follow are the name of the operator connected to the medium and the name of the port whereby the operator is connected to the medium respectively. These macro are replaced by empty strings and so they are useless at present.

3.2 Executive code generation

Like the file APPLI.m4, each source file of each operator is generated by m4 from its corresponding file PROC.m4, which include syndex.m4x. We will show the macro defined in syndex.m4x and used in PROC.m4 according to the order they are written.

Before, in addition of the file syndex.m4x, PROC.m4 may also include others files of extension .m4x corresponding to the standard library of SynDEx . These libraries belongs to the applicative kernel of the generator.

3.2.1 Generic macro

3.2.1.1 Macro processor_ and endprocessor_ These macros delimit the executive. All other macros must take place between them. The macro processor_ take five parameters :

- the processor type,
- the processor name,
- the application name,
- the version of the SynDExwhich generate this executive and a copyright,
- the date and the time of the generation.

The use of processor_ produce:

- The definition of two new macros:
  - processorType_,
  - processorName_,
- The mandatory[7] inclusion of the file TYPE.m4x, TYPE being the first parameter of the macro,
- The optional inclusion of the following files:
  - APPLI.m4x APPLI being the third parameter of the macro,
  - PROC.m4x PROC being the second parameter of the macro.

The file syndex.m4x contains only generic pacros. All the macros defining the architecture dependent code are placed in .m4x files included here.

3.2.1.2 Macro semaphores_ This macro declares the set of semaphores used in the executive to synchronize the computation thread with the different communication threads.

The file syndex.m4x doesn't define this macro. It must be defined in the architecture dependent executive kernel.

This macro take parameters::

[7]The lack of this file leads to an error.

- for each communication thread name X, `Semaphore_Thread_X`. These parameters are historic and useless;
- the name of each semaphore. That name is composed of the concatenation of the following elements in the opposite order, joined by underscores (`_`):
  - `empty` or `full` like the buffer supposed to be protected by the semaphore,
  - the name of the communication port whereby the data will get through,
  - the operator name,
  - the name of the buffer protected by the semaphore (see below the name of the buffer).

3.2.1.3 Macro `shared_` and `endshared_`

These macros delimit the declaration of the semaphore and the declaration of the buffer that will be used for a shared memory communication.

Originally, that allow to distinguish the semaphores and buffers for intra-operator data exchanges from the ones for inter-operator data exhanges. From now, the macros for inter-operator data exhanges (ie for shared memory communication) has been renamed to `semaphoresR_` and `allocR_`.

The file `syndex.m4x` does not define this macro. It must be defined in the architecture dependent executive kernel.

3.2.1.4 Macro `alloc_`

This macro declares a buffer. The rule is, a buufer is created for each data production in the algirithm graph. Thus a buffer is associated to each output port of each atomic function.

The name of this buffer is the concatenation of the following elements in the opposite order, joined by underscores:

- the port name,
- the definition name of the atomic function,
- the name of the hierarchical function whose the instance of the atomic function belongs,
- an underscore.

This macro takes three parameters:

- the data type,
- the buffer name,
- the vector size.

The file `syndex.m4x` defines this macro so that it call a more specific macro to allocate this memory:

- if a macro `NAME_alloc_` exists (`NAME` being the buffer name), it is called,
- else, if a macro `TYPE_alloc_` exists (`TYPE` being the data type), it is called

- else, the macro `basicAlloc_` is called.

In any case, the called macro take the buffer name in parameters.

The file `syndex.m4x` does not define `basicAlloc_`. It must be defined in the architecture dependent executive kernel.

3.2.1.5 Macro `semaphoresR_`

See `semaphore_`

3.2.1.6 Macro `allocR_`

See `alloc_`

3.2.1.7 Macro `alias_`

This macro allow to alias a buffer. No additional memory will be allocated, but a new buffer name will refer to an existing buffer.

The file `syndex.m4x` define this generic macro so that it checks its parameters before calling a more specific macro, `basicAlias_`, with alll its parameters.

These parameters are:

- the alias name,
- the referenced buffer name,
- the offset,
- the size of the referenced data.

The file `syndex.m4x` does not define `basicAlias_`. It must be defined in the architecture dependent executive kernel.

3.2.1.8 Macro `thread_` et `endthread_`

These macros delimit and define a communication thread.

The macro `thread_` take the following parameters:

- the medium type of the architecture,
- the port name of the operator,
- the operator name which are connected to this medium.

The call of this macro results in:

- the definition of new temporary macros:
  - `mediaType_` is the medium type (here the first parameter),
  - `mediaName_` is the port name (here the second argument),
  - `threadArgs_` is the whole parameters given to the macro `thread_`,
- the inclusion of an executive kernel specific to the medium type: the file name must be `TYPE.m4x`, `TYPE` being the medium type. This inclusion is optional;

- the inclusion of an executive kernel specific to the port of the operator: the file name must be `TYPE.m4x`, `TYPE` being the port name of the operator. This inclusion is also optional.

Finally, the call of this generic macro cause the call of few specific macros:

- `TYPE_shared_`, `TYPE` being the medium type. This macro is called with all parameters of `thread_` except the first (medium type);
- `basicThread_`, this macro is called with the second parameter of `thread_` (the port name),
- `TYPE_ini_`, in the same way as `TYPE_shared_`.

The macro `basicThread_` is systematically called unlike the two others that are called only if they exist.

The macro `endthread_` has an symmetrical behavior in relation to `thread_` :

- a macro call `TYPE_end_` as `TYPE_shared_`,
- a macro call `basicEndthread_` as `basicThread_`,
- deletion of the following definitions `mediaType_`, `mediaName_` and `threadArgs_`.

3.2.1.9 Macro `loadDnto_` et `loadFrom_` Each thread section contains at least one of these macro as first lines. These macros concern, among other things, the domwloader (see comment 3.1) and must be in charge of these two purpose:

- the launch (remotly or not) of one or more operators, which are supposed to be launched by the current operator,
- the establishing of the connection between the current operator and those that are supposed to connect with.

3.2.1.9.1 Comment about the possible romoving of these macros: The dissymmetry between the two macros is useful for the two purpose above. The first purpose requires it to know from two connected operators, which one is supposed to launch the other. The second purpose also needs it to match the dissymmetry of most of network protocols that work on a client-server model[8]. Even if we achieve to part the downloader with the executive, these macro may stay for the establishing of communications between operators.

The macro `loadDnto_` is present for the operators that are in charge of launching another operator and that are the server in the communication establishing with the other operator. The first parameter is empty and the following ones are the list of the other operators mentionned above.

The macro `loadFrom_` is present for the operators that aren't launched by hand and are supposed to connect to an operator that waits his connection. This macro has only one parameter which is the operator that is supposed to launch the current operator.

The macro `loadFrom_` can appear at most one time, because:

[8]In the case of a shared memory, only one operator creates this memory, the others just connect with it. In the case of a TCP connection, only one operator waits and listens on the port where the other operators are supposed to connect.

- the operator must be launched by at most one other operator,
- the establishing connection has been initialized by at most one other operator.

By contrast, the macro `loadDnto_` can appear several times because an operator may launch several other operators and because many of them are supposed to connect to it.

Finally, both macros can appear at the same time, beginning the `loadFrom_` one.

These two macro are generic and are defined in `syndex.m4x`. They check that they have been called from a `thread_` section and then call specific macros (`TYPE_loadFrom_` or `TYPE_loadDnto_`), `TYPE` being the name of the medium type used[9]. These specific macro are called with the same parameters as the generic ones.

3.2.1.10 Macro `saveFrom_` and `saveUpto_` These macro are the symmetric of the previous ones and are at the last lines of the `thread_` sections. For each `loadDnto_`, there is a `saveFrom_` with the same parameters, and for each `loadFrom_`, there is a `saveUpto_` with the same parameters.

In the same way, these two macros check that they have been called from a `thread_` section and then call specific macros (`TYPE_saveFrom_` or `TYPE_saveUpto_`).

3.2.1.11 Macro `loop_` and `endloop_` These two macro appear in `thread_` sections and in `main_` section. They delimit the instruction cycle of communication threads and calculation thread that needs to be runned.

`loop_` define the MGC (see appendix B) as LOOP and call the specific macro `basicLoop_`. `endloop_` defines the MGC as END and calls the specific macro `basicEndloop_`.

3.2.1.12 Macro `trans_` and `endtrans_` The macros are actually similar to `loop_` and `endloop_` to this loan that they highlight the transient phases when they exist instead of permanent phases. Therefore the macros are only shown when one generates the code of multiperiodic application. These two macros do not call for any specific macros and stick to define the MGC (LOOP for `trans_` and END for `endtrans_`).

3.2.1.13 Macro `main_` and `endmain_` As macros `thread_` and `endthread_`, these macros are used to delimit the main thread, also called calculation thread.

`main_` defines the MGC as INIT and calls on the macro `basicMain_`. `endmain_` deletes the definition of MGC and calls the specific macro `basicEndmain_`.

3.2.1.14 Macro `spawn_thread_` and `wait_endthread_` The different communication threads, being managed by the main thread, this must have instructions to spawn this threads and others to wait for their ending. The macros are there to generate these instructions.

These macros are not defined in syndex.m4x and should therefore be defined in the executive kernels dependent on the architecture.

The macro `spawn_thread_` takes in parameter the name of the thread. The macro `wait_endthread_` takes in parameter the name of the semaphore of the thread (see the macro `semaphores_`)

[9]It corresponds to the first parameter of the macro `thread_`.

3.2.1.15 Macro `Suc1_`, `Suc0_`, `Pre1_`, `Pre0_`

The macros are intra-operator synchronization macros. They take in parameter the semaphore on which they operate. Macros that starts with `Suc` release their semaphore; those starting with `Pre` wait for them. The 0 and 1 sticked to the macros are details relating to priorities for arbitration purposes of sequencers when they deal with computing thread and communication threads in a non symmetrical way. On the systems which manage multitasking as Unix and Windows, we can ignore this and make the same substitution for both variants of `Suc` and `Pre`.

3.2.1.16 Macro `SucR1_` and `PreR1_`

These macros work just the same way as the previous and are dedicated to the inter-operator synchronizations in the context of communications by shared memory.

3.2.1.17 Macro `send_`, `recv_` and `sync_`

These macros are the functions of communications et apparaissent in the or section thread. They take into first parameter the name of the buffer to send or receive. The two following parameters are the kind and the name of the operator that produced the data (which is redundant, they may seem therefore no need for the contrary). A optionel last parameter indicates the operator with which this communication (which amounted also seems redundant, the macro `thread_` providing already this information). In the case of a multipoint communication, the macro `sync_` means that when sending to a given by another operator, the current operator is not the receiver of this and can therefore ignore it. Thus, instead of in the name of the buffer in first parameter, the macro `sync_` receives the type of the given and its size.

These two macros are generic and defined in `syndex.m4x`. They check that they are called in a section `thread_` then call for the specific macros `TYPE_send`, `TYPE_recv` or `TYPE_sync`, as the case may be, "TYPE" is the name of the medium type used[10]. These specific macros are called with same parameters that the generic macros.

3.2.1.18 Macro `write_` and `read_`

These macros are instead of `send_` and `recv_` in the case of shared memory communications. Therefore they receive in parameter, in addition to the name of the local buffer, the name of the shared buffer by which the data will go through.

[10]This corresponds to the first parameter of the macro Thread

Annexes

A minimalist Makefile

```
A = control
M4 = m4

# these path have to be modified by user
SDX_MACRO_PATH=/path/to/syndex/macros
export Algo_Macros_Path = $(SDX_MACRO_PATH)/algo_libraries
export Archi_Macros_Path = $(SDX_MACRO_PATH)/archi_libraries
export M4PATH = $(Algo_Macros_Path):$(Archi_Macros_Path):$(S2s_Files_Path)
VPATH = $(M4PATH)

.PHONY: all $(A).all run $(A).run
all : $(A).mk $(A).all

run : all $(A).run

$(A).mk : $(A).m4
        $(M4) $< >$@
include $(A).mk
```

B MGC : Macro Generation Context

The MGC is a variable to give the context in which an applicative macro is called. These applicative macros may be referred to as for the initialisation phase, the transitional or permanent phase and after the permanent phase. For these macros to be substituted differently depending on a phase when they are called, they can test the content of the variable MGC.

MGC contains :

- INIT at the initialisation,
- LOOP during transitional and permanent phases,
- END after permanent phase.

Thus, here is the common way to write an applicative macro:

Applicative macro template

```
define(`my_function', `ifelse(
MGC, `INIT', `// Initialisation',
MGC, `LOOP', `// Permanent phase',
MGC, `END', `// Ending)')
```