Ellidiss
www.ellidiss.com

Stood

Coding in C User Manual

revision B

Pierre Dissaux

Contents

1. C mapping overview

The **Stood** architectural model may easily be transformed into an appropriate **C** source code file structure. **C** standard code generation rules cover efficiently the following basic transformations:

- **Components** (or **Classes**) -> a pair of source files (`.h` and `.c`)
- **Include** and **Use** relationships -> `#include` directives
- **Delegate** and **Implemented_By** relationships -> `#define` directives
- **Operations**, **Types**, **Constants** and **Data** -> respectively functions, `typedef`, `const` and `static` variables

However, the use of an automatic code generator, brings additional benefit to usual **C** development practices:

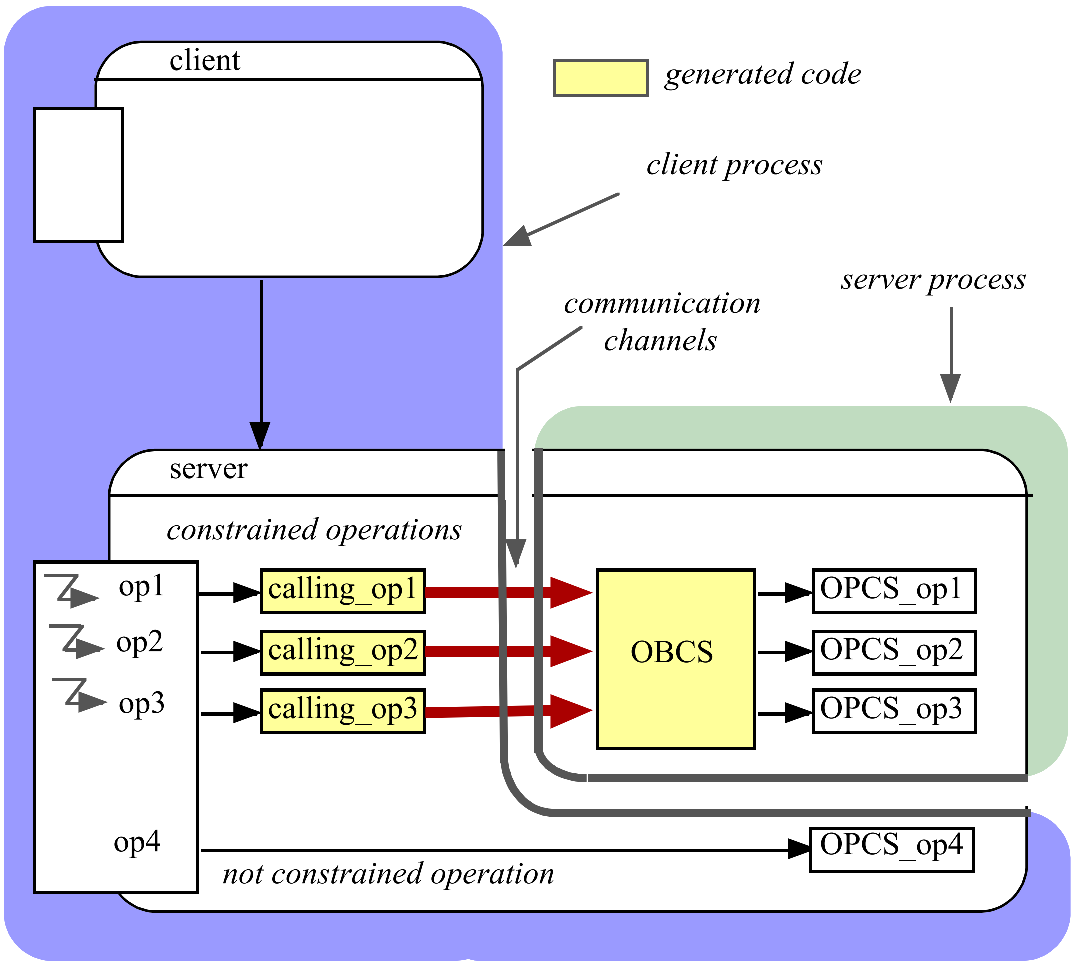
- handling name spaces using **Component** name as a prefix.
- building `makefile` dependencies from the **Required Interfaces**.
- producing commented code using the textual descriptions found inside the Object Description Skeletons (**ODS**) .

On the contrary, a few **Stood** concepts are generally more difficult to process, due to the known weaknesses in the **ANSI C** language definition: **Exceptions**, **Generics**, **O**bject-**O**riented concepts, **Virtual Nodes** (for distributed systems) and of course, tasking which support is mandatory for **R**eal-**T**ime applications.

Support of tasking for a **C** program requires access to **RTOS** entries. **Stood** supports two different strategies for **R**eal-**T**ime code. The first strategy is known as the *explicit* mapping, and consists in accessing the **RTOS** as a library that must be made visible from the applicative **Design**, as an **Environment**.

This library is also represented by a **Root Component** in the current **System**, so that it can be shared by other **Designs**. Access to **RTOS** entries must then be managed in the same way as any other **Type** or **Constant** reference or **Operation** call. This solution is very flexible, but doesn't provide the maximum of value added to the designer. The second strategy, called the *implicit* mapping, attempts to hide most of the **R**eal-**T**ime complexity to the **Application** designer. To reach this goal, all the **RTOS** calls are generated automatically by the code generator. Two possible implementations are provided. The first one consists in a direct insertion of **RTOS** primitives calls into the generated code, whereas with the second implementation, generic macro-commands are inserted, to be expanded later by the **C** pre-processor. To support tasks and inter-task communications, the **C** code generator has been extended with the following generation rules:

- Threads are created for each **Component** providing at least a protocol constrained operation (**HSER**, **LSER**, **ASER**).
- Two communication channels are created for each constrained operation: a channel for the control flow, the **in** and **in out** data flow, and a channel for **out**, **in out** dataflow and the return of the control flow). They are handled by appropriate calling procedures.
- An unique separate piece of code (the **OBCS)** is generated to encompass all the protocol, state and time out constrained operations of a given **Component**.
- Thread initialization and finalization code is automatically generated.
- **LICE** monitoring messages are elaborated to trace each constrained operation activation.
- Code generator options may be controlled by dedicated **pragmas**.


client
generated code
client process
communication channels
server process
server
constrained operations
op1
op2
op3
op4
calling_op1
calling_op2
calling_op3
OBCS
OPCS_op1
OPCS_op2
OPCS_op3
OPCS_op4
not constrained operation

2. C coding sections

While accepting **C** source within the coding sections of the design structure, **Stood** performs a lexical analysis of the entered code to identify all the local dependencies related the edited section. This local dependency information is stored into a symbol table which is stored at the same location than the code section.

Before starting the code generation process, all the symbol tables are compiled into a cross-reference table that provides valuable information to the code generator to perform all the appropriate ordering and inclusions operations that will let the produced code be compiled.

Like for all the other sections of the design data structure, the name and storage location of the **C** coding sections may be customized by editing the `DataBase` descriptor file. The names and storage locations specified below are related to the default configuration of the tool, and may differ from the actual configuration if some customizations have been performed.

The **C** coding sections that can be handled by the **Stood** design data structure (**ODS**) are presented below in three sub-groups:

- Headers
- Declarations
- Behavioural & procedural code

2.1. Headers

Headers may be defined for both `.h` and `.c` generated files. These header information can be fully hand written within the appropriate section of the design data structure or can be automatically generated if these sections are left empty and the **pragma** *comment* is set. A global code file header section can also be added at the beginning of all the generated files. These three header sections may be defined for each **Component**.

The spec and body headers may be substituted or appended by one of the standard design documentation section. The **pragmas** *subtitute_header* and *append_header*, that are described in § 3.3, must be used for that purpose.

2.1.1. Spec file header (c)

Contents of these sections must be valid **C** code. In particular, it must include the appropriate comments separators if necessary. When this section is left empty and the **pragma** *comment* is set, the header is automatically generated, and looks as follow:

```
/*******************************************************************/
/* PROJECT        : < $PROJECT variable set in initialization file > */
/* COMPANY        : < $COMPANY variable set in initialization file > */
/* FILENAME       : < actual name of the file >                    */
/* DESIGN NAME    : < name of the current design >                 */
/* MODULE NAME    : < name of the current module >                 */
/* DESCRIPTION    : < contents of the "Statement of the Problem" >   */
/* FUNCTION NAMES : < list of the provided operations >            */
/*******************************************************************/
/* Creation date   (MM/JJ/AA): < date of the day >                 */
/*******************************************************************/
```

Note that `$PROJECT` and `$COMPANY` represent two variables that will be expanded by their value during the generation process. The value of these variables can be customized within the initialization file (`stood.ini` on **Windows** and `.stoodrc` on **Unix**), in the `Environment` category.

2.1.2. Body file header (c)

Contents of these sections must be valid **C** code. In particular, it must include the appropriate comments separators if necessary. When this section is left empty and the **pragma** *comment* is set, the header is automatically generated, in a similar way as the *spec file header*.

2.1.3. Code file header (text)

Contents of these sections doesn't require to be compliant with the **C** syntax, as the comment separators will be automatically added by the code generator. Unlike the two previous sections, this header is part of the main design structure (**ODS**) and is shared by by all the code generators.

This text is included at the top of each generated **C** file associated to the **Component**, and before the *spec* and *body file headers*. No default text header is generated when the section is left empty.

2.2. Declarations

C declarations can be directly inserted within the design structure. A declaration must be given for each **Operation**, **Type**, **Constant** and **Data**. Function prototypes are always automatically generated from the **HOOD Operation** declarations, so there is no specific **C** coding section for that. **Features** belonging to the **Provided Interface** are generated in the `.h` file whereas those belonging to the **Internals** are generated in the `.c` file.

2.2.1. Operation declaration (hood)

Operation declaration must be entered in compliance with the **HOOD** syntax, restricted to the features supported by the **ANSI C** language. For instance, although allowed by the **HOOD** syntax, the use of a string literal instead of an operation identifier is only valid for **Ada** operators, and the use of the `&` type specifier is only valid for **C++** parameters passed by reference. These valid **HOOD** syntax would produce uncorrect **C** code. The syntax that can be used for a correct **C** code generation of an **Operation** declaration, is formally described below:

```
operation_declaration ::= identifier
    [op_parameter_part] [op_return_part] ;
op_parameter_part ::=
    ( op_parameter {; op_parameter} )
op_return_part ::=
    RETURN type_name {*}
op_parameter ::= identifier :
    op_parameter_mode type_name {*}
    [:= expression]
op_parameter_mode ::= in | out | in out
type_name ::= [identifier .] identifier
```

These **Operation** declarations will be translated into proper **C** function prototypes by the code generator. The `op_parameter_mode` is included as a comment within the **C** prototype. When the `in` mode is used, then the corresponding parameter will be generated as a `const`.
Example of a **HOOD** operation declaration:

```
O(
    x : in int := 0;
    y : out int;
    z : in out int;
    p : in int*
)   return M.T;
```

and of its translation into a **C** function prototype:

```
M__T M__O(const int x /*in*/, int y /*out*/,
          int z /*in out*/, const int *p /*in*/);
```

Note that **HOOD** enforces the use of explicitely defined types for all the parameters of **Operations**. In fact, due to this syntax, it is not possible to pass directly a complex type within a function prototype. This is in particular true for array types that must be properly defined as a explicit type, and used in the prototype by its type reference. Also note that, although supported by **HOOD**, function overloading and default values for parameters are not supported by the **C** syntax.

HOOD supports multiple name spaces, and implicitely considers that each **Component** defines a new name space for identifiers. That's why type references must include the name of the **Component** that actually provides the **Type**. This dot notation may be omitted if the **Type** is provided by the local **Component**, or by a standard **C** library, or another remote **C** file that doesn't use this convention. The dot separator is translated into a "__" separator by the **C** code generator. Use **pragma** *prefix_none* to enforce a single name space.

If the **pragma** *return_const* is used for an **Operation**, then the return type reference of the corresponding generated **C** prototype will be preceded by the keyword `const`. Here is the generated code for the previous example with the **pragma** *return_const (operation_name => O)* set for the **Component** *M*:

```
const M__T M__O(int x /*in*/, int y /*out*/,
         int z /*in out*/, const int *p /*in*/);
```

If the **pragma** *incomplete_prototypes* is set, then all the generated **C** prototypes will only list the parameter types, without showing the parameter names. Here is the generated code for the previous example with the **pragma** *incomplete_prototypes* set in the **Root Component**:

```
M__T M__O(int  /*in*/, int  /*out*/,
         int  /*in out*/, const int * /*in*/);
```

If the **pragma** *pointer* is set for a given `op_parameter_mode`, then the type of the all parameters of this mode will be interpreted as pointers. Possible values for this **pragma** are: *in_mode*; *out_mode* and *inout_mode*. Here is the generated code for the previous example with the **pragma** *pointer(parameter_mode => out_mode)* set in the **Root Component**:

```
M__T M__O(int x /*in*/, int *y /*out*/,
         int z /*in out*/, const int *p /*in*/);
```

If the **pragma** *not_ANSI* is set, then no prototype will be generated:

```
M__T M__O();
```

Note that the default generation rules put an explicit external linkage to all the provided and internal functions. The **pragma** *static* (refer to §3.5.5), must be used to get an internal linkage for internal functions.

2.2.2. Type definition (c)

The *type definition* section must contain a valid **C** type declaration. A summary of this syntax is given below. Please refer to the **ANSI C** Reference Manual for further details.

```
type_declaration ::= typedef
    type_definition type_name ;
type_definition ::= type_name
    | struct_definition
    | enum_definition
    | union_definition
type_name ::= [identifier __] identifier
```

If this section is left empty, the code generator will attempt to use any other appropriate information from the design data structure. If the *type attributes (hood)* section doesn't contain **NONE**, then it will be automatically translated into a **C** `struct_definition`. Otherwise, if the *type enumeration (hood)* section doesn't contain **NONE**, then it will be automatically translated into an **C** `enum_definition`.

Example of **HOOD** definition with **Attributes**, for a **Type** `T` in **Component** `M`:

```
ATTRIBUTES x : int := 0, y : M.A
```

and the corresponding **C** type definition that is automatically generated by **Stood**. Note that default values are not allowed in **C** struct declarations:

```
typedef struct { int x; M__A y } M__T;
```

However, only the dependencies found in the *type definition* section, are properly managed by the code generator. If this section is left empty, remote types of attributes could be ignored by the compiler, and cause parse errors.

Example of a **HOOD** definition with Enumeration elements, for a **Type** `T` in **Component** `M`:

```
ENUMERATION red, yellow, green
```

and the corresponding **C** type definition generated by **Stood**:

```
typedef enum { red, yellow, green };
```

While completing the *type definition* section, it is also possible to ask for a *template* by using the relevant item of the contextual menu of the text editing area. If the t*ype attributes (hood)* or *type enumeration (hood)* sections don't contain **NONE**, they will be used to propose the corresponding templates of **C** type definitions. These templates contain several type definition proposals, but only one **C** type definition must be kept in the *type definition* section.

HOOD supports multiple name spaces, and implicitely considers that each **Component** defines a new name space for identifiers. That's why type references must include the name of the **Component** that actually provides the **Type**. This dot notation may be omitted if the **Type** is provided by the local **Component**, or by a standard **C** library, or another remote **C** file that doesn't use this convention. The dot separator must be an actual `"."` character in all expressions compliant whith the **HOOD** syntax, and will be translated into a `"__"` separator by the **C** code generator. The dot separator must be entered in its definitive form `"__"` in all expressions compliant with the **C** syntax, and won't be altered by the **C** code generator. Use **pragma** *prefix_none* to enforce a single name space.

Design `std` that is included in the `libs` directory provides predefined **C** types (`int`, `signed`, `unsigned`, `short`, `long`, `float`, `double`, `char`). To use it, add this **Design** to your **Project** and create it as an environment object in the graphical editor.

2.2.3. Constant definition (c)

The *constant definition* section must contain a valid **C** constant declaration. A summary of this syntax is given below. Please refer to the **ANSI C** Reference Manual for further details.

```
constant_declaration ::=
      #define constant_name value
    | const type_name constant_name = value ;
constant_name ::= [identifier __] identifier
type_name ::= [identifier __] identifier
```

Stood accepts constant definitions to be given either by a `#define` directive or by a `const` qualifier on data declarations. These declarations, including the keyword `#define` or `const`, must be entered in the *constant declaration* section, and will be pasted into the generated code files.

Examples of code to be typed in the section *constant definition (c)*.to declare a **Constant** `C` in the **Component** `M`. Such examples can be automatically pasted to the editing area, with the paste from template contextual menu:

```
#define M__C 0
```
or:
```
const int M__C = 0;
```

Note that the default generation rules put an explicit external linkage to all the provided and internal **Constants**. The **pragma** *static* (refer to §3.5.5) must be used to get an internal linkage for internal **Constants**. When declared `static`, the "dot" notation is not mandatory for internal **Constants**.

2.2.4. Data declaration (c)

The *data declaration* section must contain a valid **C** variable declaration. A summary of this syntax is given below. Please refer to the **ANSI C** Reference Manual for further details.

```
variable_declaration ::=
    type_name variable_name = value ;
variable_name ::= [identifier __] identifier
type_name ::= [identifier __] identifier
```

The standard design rule promoted by **HOOD** is to avoid global variables This means that the default configuration of **Stood** doesn't allow to create provided **Data**. It means that with this default configuration, only internal **Data** are allowed and all dataflows between **Components** must be controled by access functions, in order to cope with data hiding principles. However, this rule may be by-passed by appropriate changes in the configuration files, to comply to alternate project wide or company wide processes.

Note that the default generation rules put an explicit external linkage to all the **Data**. The **pragma** *static* (refer to §3.5.5) must be used to get an internal linkage for internal **Data**. When declared `static`, the "dot" notation is not mandatory for internal **Data**.

2.3. Behavioural and procedural code

The **HOOD** method distinguishes behavioural from procedural parts of the implementation. Although it is not always so easy to separate these two sub sets while coding, it is much easier to manage that at the design level. The procedural code refers to function bodies implementing sequentially executed code that is not dependent upon its environment. On the contrary, behavioural code contains all the statements handling the change of behaviour due to the change of the internal state or to inter tasks communication.

The procedural piece of code is defined for each individual terminal **Operation** and is called the **OPCS** (OPeration Control Structure), whereas the behavioural code is global to a terminal **Component** and is called the **OBCS** (OBject Control Structure). With **Stood**, the **OPCS** is always hand-coded within the *operation code (c)* section. On the contrary, the **OBCS** may be automatically generated from the design model, either from a **State Transition Diagram** for state constraints on **Operations**, or from inter-tasks communication protocol defined for constrained **Operations** provided by **Active Components**. The **OBCS** code may also be hand coded through the *obcs code (c)* section.

The automatic implementation of a **State Transition Diagram** is always available, whereas the automatic code generation for inter-tasks communication protocols needs to refer to a dedicated **RTOS** (Real Time Operating System) library, which must be specified by a code generation **pragma**.

Inputs for behavioural and procedural code generation consist in four sections related to the **State Transition Diagram**, a section for the **OBCS** and a section for the **OPCS**.

2.3.1. State assignment (c)

The *state assignment* **C**
A summary of this syntax is given below. Please refer to the **ANSI C** Reference Manual for further details.

```
state_assignment ::= assignment { assignment }
assignment ::= variable_name = value ;
variable_name ::= [identifier __] identifier
```

The designated variables, that acts as state variables, must be either existing internal **Data** of the current terminal **Component**, if it is an **Object**, or existing **Attributes** if it is a **Class**. Assignments can be replaced by **Operation** calls.

2.3.2. State test (c)

The *state test* section must contain a valid **C** logical expression applied on a set of variables. A summary of this syntax is given below. Please refer to the **ANSI C** Reference Manual for further details.

```
state_test ::= logical_expr { logical_expr }
logical_expr ::= [ logical_oper ]
   variable_name rel_oper value
log_oper ::= ! | && | ||
rel_oper ::= == | != | < | <= | > | >=
variable_name ::= [identifier __] identifier
```

The designated variables, that acts as state variables, must be either existing internal **Data** of the current terminal **Component**, for an **Object**, or existing **Attributes** for a **Class**. **Operations** returning proper values can also be used.

2.3.3. Transition condition (c)

The *transition condition* section must contain a valid **C** logical expression applied on a set of variables. A summary of this syntax is given below. Please refer to the **ANSI C** Reference Manual for further details.

```
trans_cond ::= logical_expr { logical_expr }
logical_expr ::= [ logical_oper ]
   variable_name rel_oper value
log_oper ::= ! | && | ||
rel_oper ::= == | != | < | <= | > | >=
variable_name ::= [identifier __] identifier
```

This capability to add conditions to **Transitions** is a way to simplify the **State Transition Diagram** by reducing the number of explicit **States**. It can also be useful to express dependencies upon remote state variables.

One typical use of conditions for **Transitions** is demonstrated by the implementation of a counter which internal state variable is the value of the counter. Instead of defining a **State** for each possible value of the counter, the **HOOD** approach will be to specify only the **States** of receptivity of the provided **Operations**, that is here `increment` and `decrement`. The **State Transition Diagram** can thus have only three **States**, `init`, `max` and `intermediate`, whatever the size of the counter is. This introduces non-determinism for **Transitions**, that will be resolved by adding *transition condition* sections.

2.3.4. Transition exception (c)

The *transition exception* section must contain any valid **C** executable statement Please refer to the **ANSI C** Reference Manual for further details.

Default behaviour is that nothing is done when an **Operation** is called while the current **State** doesn't accept it (that is there is no **Transition** triggered by this **Operation** to exit from the current **State**). If a particular action is required in such cases, the section *transition exception* can be used.

Example:

```
void file__write(char* text /*in*/)
{
  if ( file__state != is_locked ) {
    if ( disk__not_full() ) {
      OPCS_write(text);
      file__state := is_locked;
    }
    else
      printf("file is already locked\n");
  }
}
```

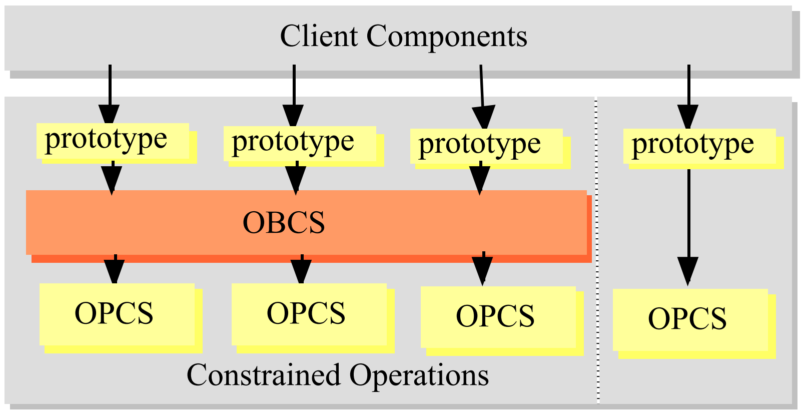
This sample of generated code used the following information from the design model:

state test:	`file__state != is_locked`
transition condition:	`disk__not_full()`
state assignment:	`file__state := is_locked;`
transition exception:	`printf("file is already locked\n");`

2.3.5. Obcs code (c)

The **OBCS** (OBject Control Structure) should contain all the middleware code between the external **Operation** interface (function prototype) and its body located in the section operation code of the **OPCS** (OPeration Control Structure). The *obcs code* section is shared by all the **Operations** of the **Component** and is the most appropriate location to implement multitasking and communication code. This section is usually generated.automatically. However, it can also be hand coded, in which case it must contain a set of valid **C** declarations and or function definitions. If the section *obcs code* is not empty the corresponding output of the automatic code generation will be substituted by the contents of this section.

The *obcs code* section can also be used to capture the generated code, customize it, and make this changes persistent for the future code generations. To feed this section back from the generated code, please use the **pragma** *reverse* and accept the changes with the reversor tool.



The default code generation rules for the obcs code will be applied only for **Active Components** providing protocol **Constrained Operations**.and when a target **RTOS** (Real Time Operation System) has been specified with the **pragma** *target* In all the other cases, no code will be generated for the **OBCS**. These default rules consist in set of type and variable declarations, and function definitions. Only the main structure of the **OBCS** generation rules is summarized below:

```
obcs_body ::=
    obcs_task_declaration
    { server_task_declaration }
    { i_o_channel_declarations }
    obcs_stop_channel_declaration
    { server_task_stop_channel_declaration }
    { i_o_message_declarations }
    { entry_calling_procedure }
    obcs_init_procedure
    { server_task_init_procedure }
    obcs_exec_procedure obcs_stop_procedure
    { s_task_exec_procedure s_task_stop_procedure }
    obcs_term_procedure
    { server_task_term_procedure }
```

Note that when the automatic code generation of the **OBCS** is activated, additional code is also included to the main function, in order to properly elaborate and initialize the **Real Time** features at run time.

The main thread of an **Active Component** is called the **OBCS**. Each protocol **Constrained Operation** acts as an entry for the thread **OBCS**. When an entry is accepted, the corresponding function body (**OPCS**) is executed. Intertasks communication is managed by channels. Channels carry the control flow (`call_` and `return_` parameters) and the various function arguments if there are any. There is one pair of channels for each **Constrained Operation**: an incoming channel carrying the `in`, `in out` and `call_` parameters, and an outcoming channel carrying the `out`, `in out` and `return_` parameters. To perform proper data copy, a message data structure is created for each channel.

An additional thread is created for each (non interrupt) asynchronous (**ASER**) **Operation**. These threads are called `server_tasks` and are used to release the calling task, as quickly as possible.

The generated code can be expanded either by generic macro commands that can be customized remotely, or by direct calls to the **RTOS** library. This choice must be specified with the first parameter of the **pragma** *target*. The second parameter must be used to specify the directory containing the generic macro command expansion file `taskCom.h`, when the first parameter in *generic*. Additional information can be specified with the **pragma** *target_param*.

These default **Real Time** code generation rules are only provided as templates. The code generator has been designed to be customized for any particular set of alternate rules or **RTOS**, as easily as possible. Please contact the technical support to customize the **OBCS** code generation rules to fit the requirements of your project.

2.3.6. Operation code (c)

The *operation code* section must contain a valid **C** function definition for each **Operation**. This section must not contain the prototype of the function, as it will be generated automatically from the *operation declaration* section to avoid any risk of mismatch, but must include the opening and closing braces.

This code will be used as is for unconstrained **Operations**. For **Constrained Operations**, the contents of this section will be used to fill in the body of the corresponding **OPCS** function, whereas the function call will be redirected to the proper **State Transition** or **Real Time** behavioural code.

Note that the argument list in the header of the function definition is generated by default in a "modern" style, that is it is a copy of the function prototype. To get this argument list with an "old" style, the **pragma** *not_ANSI* must be used.

Example of default generation for a function definition header:

```
M__T M__O(int x /*in*/, int y /*out*/,
          int z /*in out*/)
{ }
```

Same example with the **pragma** `not_ANSI`:

```
M__T M__O(x /*in*/,y /*out*/,z /*in out*/);
int x;
int y;
int z;
{ }
```

3. C generation options

The code may be generated without setting any **pragma**. In that case, the standard code generation rules will be applied. However, it may be necessary to use some of the available tuning parameters, either to modify the default generation rules, or to add information that can't be found in the design model.

These options may all be controled from the *code generator* window, are stored in a dedicated file, called `PRAGMA`, for each **Component**. **Pragmas** may have different scopes. Some of them are active for the whole **Design**, others are only valid for the selected **Component**, with sometimes a parameter to limit its scope to a particular **Feature**. This scope is specified for each **pragma** in the description below, and it is a good pratice to allocate to the **Root Component** all the **pragmas** having a **Design** wide scope. For the purpose of their description, the **pragmas** and have been grouped into five categories:

- The main options, to control the automatic production of the `makefile`, and the insertion of comments and round-trip engineering tags.
- The **RTOS** options, to specify which Real-Time Operating System **API** must be used to support multi-tasking.
- The presentation options, to tune line width, indentation, file headers, and apply project specific rules.
- The functions options, to give additional details about the way functions and their parameters must be implemented.
- The others options, that haven't been put in any of the above categories.

3.1. main options

3.1.1. PRAGMA main

This **pragma** must be used to specify the main entry point for the **Application**. The parameter `operation_name` must be set to the name of an existing **Operation** provided by the **Root Component** of the **Application**.

scope:

Its effect is global to the **Design**, but it must be set in the **Root Component** only.

example:

If `root` is the **Root Component**, providing an operation named `start`, and the **pragma** *main* has been set as follow:

main (operation_name => start)

then the following code will de generated inside the file `root.c`:

```
/* ---------- include provided------------------ */
#include "root.h"
/* ---------- main entry point: ---------------- */
int main(int argc, char *argv[ ])
{
  return root__start(argc, &argv[0]);
}
```

restriction:

As the `.c` filename for the **Root Component** is used to implement the `main function`, it is not currently possible to apply the **pragma** *main* for a **Terminal Root Component**.

3.1.2. PRAGMA cc

This **pragma** may be used to specify the name of the **C** compiler that must be used when executing the generated `makefile`.

scope:
Its scope is the whole **Design**. It is recommended to set it for the **Root Component** only.

example:
If the **pragma** *cc* has been set as follow:
cc (compiler => gcc)
then the generated `makefile` will contain the line shown below:

```
CC=gcc
```

note:
It may be necessary to temporarily modify the execution path to let **Stood** launch the compiler after having generated the **C** source code. The initialization file (`stood.ini` for **Windows** and `.stoodrc` for **Unix**) allows the environment variable `C_PATH` to be set for that purpose. For **Stood** to be able to launch the specified compiler, this variable must contain its directory pathname. This is useful only if the compiler is not reachable with the default execution path.

3.1.3. PRAGMA cflags

This **pragma** may be used to specify **C** compiler options that must be used when executing the `makefile`.

<u>scope:</u>
Its scope is the whole **Design**. It is recommended to set it for the **Root Component** only.

<u>example:</u>
If the **pragma** *cflags* has been set as follow:
cflags (flags => -c)
then the generated `makefile` will contain the line shown below:

```
CFLAGS=-c
```

3.1.4. PRAGMA ldflags

This **pragma** may be used to specify **C** linker options that must be used when executing the `makefile`.

<u>scope:</u>
Its scope is the whole **Design**. It is recommended to set it for the **Root Component** only.

<u>example:</u>
If the **pragma** *ldflags* has been set as follow:
ldflags (flags => -L/usr/dt/lib)
then the generated `makefile` will contain the line shown below:

```
LDFLAGS=-L/usr/dt/lib
```

3.1.5. PRAGMA comment

This **pragma** can be used to automatically insert comments inside the **C** source files. The comment lines are copied from the relevant *textual descriptions* that can be found in the detailed design information structure (**ODS**). In addition, when this **pragma** is set, file headers such as shown below are generated:

```
/****************************************************/
/* PROJECT          : Demonstration                */
/* COMPANY          : TNI                          */
/* FILENAME         : root.h                       */
/* DESIGN NAME      : root                         */
/* MODULE NAME      : root                         */
/* DESCRIPTION      :                              */
/* contents of the section :                       */
/*      Statement of the problem                   */
/* FUNCTION NAMES  :                               */
/* main__start                                     */
/* main__stop                                      */
/****************************************************/
/* Creation date    (MM/JJ/AA): 01/23/2003         */
/****************************************************/
```

The `Project` and `Company` variables can be customized in the initialization file (`stood.ini` on **Windows** and `.stoodrc` on **Unix**).

scope:

Its scope is the whole **Design**. It is recommended to set it for the **Root Component** only.

note:

When this **pragma** is set and some *textual descriptions* are missing, then the file `extraction messages` will contain warnings.

3.1.6. PRAGMA reverse

This **pragma** must be used to insert round-trip engineering tags within the generated source code. The users will be allowed to perform any change between the `begin` and `end` tags, but not anywhere else. The code changes will then be introduced into the design model, thanks to the *C reversor* function

The first parameter of this **pragma** has an integer type and takes the value `1`. Other values are reserved for future needs. The second parameter can either take the value `no` or contain a string that will be used to decorate the tag.

scope:
Its scope is the whole **Design**. It is recommended to set it for the **Root Component** only.

example:
If the **pragma** *reverse(option => 1,separator => "===")* is set, following code will be produced:

```
/* ===<begin OpDecl M O>                                   */
void M__O(void)
/* ===<end>                                                */
```

The user can then change the prototype of this function directly within the **C** file, and the corresponding **Design** data will be updated while using the **C** reversor. When a piece of code appears several times, the reverse tags are set only once (i.e. function prototypes are tagged in the body files only).

restriction:
Allowed changes are only those accepted for the corresponding coding section. This is especially true for the operation declarations.

3.1.7. PRAGMA std_file

This **pragma** must be set on **Environment Component**, to denote standard header files. When this **pragma** is set, the appropriate syntax will be used in the `#include` directives that are automatically generated.

<u>scope:</u>
Its scope is the selected **Root Component**. It is relevant when used as an **Environment Component** only.

<u>example:</u>
If the **pragma** *std_file* is set for the **Environment Component** `stdlib`, then the following code will be generated in each file requiring it:

```
#include <stdlib.h>
```

If the **pragma** is not set, the corresponding code will be:

```
#include "stdlib.h"
```

<u>note</u>:
While creating a local copy of an **Environment** in the graphic editor, its **pragmas** can be imported. If the **pragma** *std_file* is set in the referenced **Design**, then it wil not be necessary to reset it within each using **Design**, where it appears as an **Environment**.

3.1.8. PRAGMA location

This **pragma** is used to insert information in the `makefile`, to provide the linker with the actual location of required **Environment Component**.

scope:
Its scope is the selected **Root Component**. It is relevant when used as an **Environment Component** only.

example:
If the **pragma** *location* has been set as follow, for the **Environment Component** `display`:

```
location(where => /home/libs, extension => .o)
```

then the following line will be automatically included into the generated `makefile`:

```
OBJ=/home/libs/display.o
```

3.1.9. PRAGMA except

This **pragma** can be used not to generate code for the corresponding **Component**.

<u>scope:</u>
Its scope is the selected **Component** only.

<u>note:</u>
Any existing source file attached to the selected list of **Components** are usually deleted during the initialization phase of a code generation. In order to avoid these deletions for the **Components** for which the **pragma** *except* has been set, the property `NoCleanUpFor` must be define in the initialization file:

on **Windows** (`stood.ini`):

```
[Languages]
NoCleanUpFor=except
```

on **Unix** (`.stoodrc`):

```
Languages.NoCleanUpFor:except
```

This property is set by default, and can be very usefull when source code files for some **Components** of the **Design** are generated by other tools.

3.2. RTOS options

3.2.1. PRAGMA target

If this **pragma** is not set, no automatic code generation will be done to implement the **Real Time** features of the **Design** (refer to §1 for a description of the mapping). In order to generate the **Real Time** code (i.e. processes and communications) from the architectural **Design** description, a run-time executive target must be specified.

The first parameter of this **pragma** must be used to specify the name of the run-time executive. Currently, only two targets are supported by the default **C** code generator rules: `transputer` and `generic`. When `transputer` is chosen, direct calls to the **C** run time library for transputer based targets are automatically generated. When `generic` is chosen, calls to a set predefined macro command are included into the generated code. These macro commands are expanded in a different way for each native target, within a file named `taskCom.h`. Other direct targets can of course be implemented. Please contact the technical support to submit your requirements

The second parameter specifies the name of the directory containing the `taskCom.h` file for the chosen target. The current possible values for this parameter are: `posix`, `arinc653`, `rtc` and `transputer`. These macro command definition files are provided as is, and need to be customized and validated in the context of the project before any industrial use.

<u>scope:</u>
Its scope is the whole **Design**. It is recommended to set it for the **Root Component** only.

3.2.2. PRAGMA target_param

This **pragma** must be used in association with the **pragma** *target*, to provide additional information to generate the **Real Time** code. The first argument of this **pragma**, called `param`, denotes the name of a key, and the second argument, called **value**, contains the key value. This is a very flexible way to introduce alternate code generation patterns.

As an example, here are the keys and their possible values that have been defined for the target `transputer`. Similar pairs of key-value could be defined for other targets.

target_param(param=>priority,value=high|low)
This **pragma** specifies if the corresponding thread must be run with a high (`ProcRunHigh`), or a low priority (`ProcRunLow`). If this **pragma** is not used, the default function (`ProcRun`) will be used.

target_param(param=>wsize,value=>size)
target_param(param=>stack_address,value=>address)
These two **pragmas** may be used to finely control the thread stack allocation, by providing a value to its size (in bytes, or `0` for the default value), and relative address.

<u>scope</u>:
These **pragma** must be attached to an **Active Component**.

3.2.3. PRAGMA LICE

This **pragma** must be used to let the code generator produce appropriate information at run time for the **LICE** tools. **LICE** is a **R**eal-**T**ime monitoring system that has been developed by the French Space Agency (**CNES**). The aim is to trace the main behavioural events of the program such as stopping threads, calling and returning from communication channels. It produces an output file containing the events encoding. This filename must be specified as a parameter, and the default value `user` will redirect the output to the console. Calls to the predefined function `liceTrace` are also inserted within the executable code.

<u>scope:</u>
Its scope is the whole **Design**. It is recommended to set it for the **Root Component** only.

<u>example of LICE event encoding generated by Stood:</u>

```
=== begin LICE encoding
0x41      call HSER op.
0x42      exit HSER op.
0x43      call LSER op.
0x44      exit LSER op.
0x45      call ASER op.
0x46      exit ASER op.
0x47      call stop task.
0x48      exit stop task.
----------------------------------
0x00000101      transputer.run
0x00000201      client.run
0x00000202      client.notify
0x00000301      cyclic.start
0x00000302      cyclic.timer
0x00000303      cyclic.stop
0x00000401      shared.read
0x00000402      shared.write
=== end LICE encoding
```

3.3. presentation options

3.3.1. PRAGMA insert_pseudo

When used, this **pragma** will let the code generator insert the contents of the *pseudo code* section of an **Operation** or the **Obcs**, as a comment in the **C** source files.

scope:
The scope of this **pragma** is the whole **Design**. It is recommended to set it for the **Root Component** only.

3.3.2. PRAGMA presentation

The **pragma** presentation can be used to control the generated source files layout. Three options are currently supported, and the aim is of course to add others in the future to cope with specific corporate or project recommendations. Note that this **pragma** may be set several times with a different option.

presentation(option=>no_line_feed)
If this **pragma** is set, then compact code will be generated, that is decorative line feeds will be removed.

presentation(option=>indent_param)
If this **pragma** is set, then each argument of function declarations and definitions will be generated on a separate line.

presentation(option=>microsat)
This **pragma** option is an example of what can be done to customize the code generator for a particular project. When it is set, it impacts various code generation rules to comply with the coding requirements of the project micro-satellite developed by the French Space Agency (**CNES**). Please contact the technical support to submit similar requirements that would be applicable for your project.

<u>scope:</u>
The scope of these **pragmas** is the whole **Design**. It is recommended to set it for the **Root Component** only.

3.3.3. PRAGMA line_width

This **pragma** may be used to specify the maximum length of the lines that are produced by the code generator. The default value is `80`.
Note that pieces of code that have been entered manually within the coding sections of the design data structure wont be affected by the value of this **pragma**. It is the responsability of the user to ajust the length of the hand written source code lines.

<u>scope:</u>
The scope of this **pragma** is the whole **Design**. It is recommended to set it for the **Root Component** only.

3.3.4. PRAGMA indent_width

This **pragma** may be used to specify the number of space characters that are used for one code indentation. The default value is `2`.

Note that pieces of code that have been entered manually within the coding sections of the design data structure wont be affected by the value of this **pragma**. It is the responsability of the user to ajust the width of the indentation within hand written source code lines.

<u>scope:</u>
The scope of this **pragma** is the whole **Design**. It is recommended to set it for the **Root Component** only.

3.3.5. PRAGMA substitute_header

Headers may be inserted at the beginning of each source file that is generated (refer to §2.1). This **pragma** may be used to substitute a header by the contents of one of the informal textual description sections of the current **Component**.

The *logical name* of the header to be substituted must be specified as the first argument of this **pragma**, and the *logical name* of the substitute description section must be specified as the second argument. *Logical names* are those defined in the `DataBase` configuration file, and can be shown with the *definition* contextual menu in the list of **Properties**.

<u>scope:</u>
The scope of this **pragma** is the whole **Design**. It is recommended to set it for the **Root Component** only.

<u>restriction</u>:
This **pragma** has no effect if the **pragma** *append_header* is already used.

3.3.6. PRAGMA append_header

Headers may be inserted at the beginning of each source file that is generated (refer to §2.1). This **pragma** may be used to append the contents of one of the informal textual description sections of the current **Component**, to the specified header.

The *logical name* of the header to be extended must be specified as the first argument of this **pragma**, and the *logical name* of the appended description section must be specified as the second argument. *Logical names* are those defined in the `DataBase` configuration file, and can be shown with the *definition* contextual menu in the list of **Properties**.

scope:
The scope of this **pragma** is the whole **Design**. It is recommended to set it for the **Root Component** only.

3.3.7. PRAGMA tabulation

This **pragma** can be used to control the width of the tabutation characters that are found inside the textual information files and that are inserted as comments in the generated source files (refer to §3.1.5). The number of space characters to be used in replacement of each tab character is specified in argument. Default value is 4.

scope:
The scope of this **pragma** is the whole **Design**. It is recommended to set it for the **Root Component** only.

3.4. function options

3.4.1. PRAGMA inline

When this **pragma** is set, the body of the specified function is included inside the header file of the **Component**, instead of including it inside the source file. In addition, the value passed as parameter to the **pragma** *inline_def* (refer to §3.5.4) is inserted at the beginning of the function definition.

<u>scope:</u>
The scope of this **pragma** is the specified **Operation** of the selected **Component**. The **pragma** must be duplicated for each function to be set inline.

<u>restriction:</u>
If the **pragma** *inline_def* is not set, then the **pragma** *inline* has no effect.

<u>example:</u>
Assuming that the following **pragmas** have been set for the **Component** account:
inline_def(keyword=>__inline__)
inline(operation_name=>read)
Then, the file `account.h` will contain the following code:

```
...
__inline__ int account__read(void)
{
  ...
  return value;
}
...
```

3.4.2. PRAGMA pointer

If the **pragma** *pointer* is set for a given `op_parameter_mode`, (refer to §2.2.1), then the type of the all parameters of this mode will be interpreted as pointers. Possible values for the unique parameter of this **pragma** are: *in_mode*; *out_mode* and *inout_mode*.

scope:
The scope of this **pragma** is the whole **Design**. It is recommended to set it for the **Root Component** only.

example:
Given the following **Operation** declaration in the **Component** `buffer`:

```
get(element : in int; bigvalue : out buffer.bigstruct)
```

And assuming that the following **pragmas** have been set:
pointer(parameter_mode => out_mode)
presentation(option=>indent_param)
Then the corresponding generated code for this function prototype will be:

```
buffer__get(
  int element /*in*/,
  buffer__bigstruct *bigvalue /*out*/);
```

3.4.3. PRAGMA return_const

If the **pragma** *return_const* is used for an **Operation**, then the return type reference of the corresponding generated **C** prototype will be preceded by the keyword `const`.

<u>scope:</u>
The scope of this **pragma** is the specified **Operation** of the selected **Component**. The **pragma** must be duplicated for each function to be set inline.

<u>example:</u>
Given the following **Operation** declaration in the **Component** `buffer`:

```
get_size return int;
```

And assuming that the following **pragma** has been set:
return_const(operation_name => buffer_size)
Then the corresponding generated code for this function prototype will be:

```
const int buffer__get_size(void);
```

3.4.4. PRAGMA storage

This **pragma** is an example of what kind of precise customization of the code generation rules can be done to cope with specific compilation contexts. This **pragma** introduces conditional compilation directives associated to memory space allocation qualifiers.

The first parameter denotes the function which the **pragma** applies to. The second parameter carries the symbol that is used for testing the condition, and the third parameter must be a valid pointer memory space qualifier.

scope:
The scope of this **pragma** is the specified **Operation** of the selected **Component**. The **pragma** must be duplicated for each function to be set inline.

example:
Given the following **Operation** declaration in the **Component** `sensor`:
`get_data return sensor__signal*;`
And assuming that the following **pragma** has been set:
storage(operation_name => get_data,
    compiler => __GNUC__, pointer_storage => dm)
Then the corresponding generated code for this function prototype will be:

```
#ifdef __GNUC__
sensor__signal dm *sensor__get_data(void);
#else
sensor__signal *sensor__get_data(void);
#endif
```

3.4.5. PRAGMA incomplete_prototypes

If the **pragma** *incomplete_prototypes* is set, then all the generated **C** prototypes will only list the parameter types, without showing the parameter names.

scope:
The scope of this **pragma** is the whole **Design**. It is recommended to set it for the **Root Component** only.

example:
Given the following **Operation** declaration in the **Component** `buffer`:

```
get(element : in int; value : out buffer.smallstruct)
```

And assuming that the **pragma** *incomplete_prototypes* has been set,
then the corresponding generated code for this function prototype will be:

```
buffer__get(int /*in*/,buffer__smallstruct /*out*/);
```

3.4.6. PRAGMA not_ANSI

When the **pragma** *not_ANSI* is set, **C** function declarations and definitions are generated using the "old" style. No prototype is then produced for the function declaration, and the parameter types are specified on separated lines. If the **pragma** is not set, prototypes and "modern" style will be used.

<u>scope:</u>

The scope of this **pragma** is the whole **Design**. It is recommended to set it for the **Root Component** only.

<u>example:</u>

Given the following **Operation** declaration in the **Component** `buffer`:

```
get(element : in int; value : out buffer.smallstruct)
```

And assuming that the **pragma** *not_ANSI* has been set, then the corresponding generated code for this function declaration and definition will be:

```
buffer__get();
buffer  get(element /*in*/, value /*out*/)
int element;
smallstruct value;
{ }
```

<u>restriction:</u>

This **pragma** has no effect on the code that is automatically generated to support **R**eal-**T**ime features, especially when the **pragma** target has been set.

3.5. other options

3.5.1. PRAGMA no_command_line

When the **pragma** *main* is used (refer to §3.1.1), one of the **Operations** that are provided by the **Root Component** acts as the `main` function of the program. Standard behaviour is then to replace its existing parameter list, if any, by the standard **C** command line parameters. If the **pragma** *no_command_line* has been set, the existing parameter list, if any, is replaced by `void`.

scope:
The scope of this **pragma** is the whole **Design**. It is recommended to set it for the **Root Component** only.

example:
If `root` is the **Root Component**, providing an operation named `start`, and the following **pragmas** have been set:
main (operation_name => start)
no_command_line
then the following code will de generated inside the file `root.c`:

```
#include root.h
/* ---------- main entry point: ---------------- */
int main(void)
{
  root__start();
  return 0;
}
```

3.5.2. PRAGMA no_define

`#define` preprocessor directives are used by the code generator to manage the **HOOD Implemented_By** or **UML Delegate** relationships that are defined between the different levels in the hierarchy of **Components**. In some situations, it is not appropriate to do so. If the **pragma** *no_define* is set, then alternate code generation rules will be used for **Provided Constants**, **Data**(*) and **Operations**.

The effect of this **pragma** is to create a source file for those **Non Terminal Components** that provide at least a **Constant**, **Data**(*) or **Operation**. This source file will contain the appropriate implementation code.

scope:
The scope of this **pragma** is the whole **Design**. It is recommended to set it for the **Root Component** only.

restriction:
Use of this **pragma** doesn't eliminate all the `#define` directives. It just manages the code generation rules for the **Implemented_By** links.
If some **Constants** have been defined explicitely using a `#define` directive, then this **pragma** won't have any effect on them.

note:
(*) **Provided Data** are not recommended by the **HOOD** design rules. However, they may be supported by **Stood** and the **C** code generator provided that the appropriate custimization has been performed.

3.5.3. PRAGMA full_file_names

This **pragma** can be used to control the way filenames are inserted within the `#include` directives and the `makefile`. Default behaviour is to insert short filenames. If the **pragma** *full_path_name* is set, then the full pathname will be used: basename+filename.

scope:
The scope of this **pragma** is the whole **Design**. It is recommended to set it for the **Root Component** only.

3.5.4. PRAGMA inline_def

This **pragma** can be set to specify which keyword must be used to manage the *inline* feature. As it is not supported by the **ANSI** standard, the availability of this feature and the associated keyword may vary depending on the compiling environment.

The parameter of this **pragma** carries the keyword that will be used for each **Operation** for which the **pragma** *inline* has been set (refer to §3.4.1). If the **pragma** *inline* is used and the **pragma** *inline_def* is not set, then no keyword will be inserted, which can lead to compilation errors.

<u>scope:</u>
The scope of this **pragma** is the whole **Design**. It is recommended to set it for the **Root Component** only.

3.5.5. PRAGMA static

This **pragma** must be used to conditionally apply an internal linkage to all the **Internal Features**. The first parameter carries the keyword `static` or the name of a preprocessor macro (typically: `STATIC`). The second parameter contains the condition for the definition of this preprocessor macro.

<u>scope:</u>
The scope of this **pragma** is the whole **Design**. It is recommended to set it for the **Root Component** only.

<u>example:</u>
Assuming that with no **pragma** *static* set, the following code is generated in the source file of the **Component** `buffer`:

```
int buffer__get_size(void);
```

It will become as follow if **pragma** *static(keyword=>static,condition=>no)* is set:

```
static int buffer__get_size(void);
```

And finally, if the **pragma** *static(keyword=>STATIC,condition=>TEST)*, the generated code will be:

```
#ifndef TEST
  #define STATIC static
#else
  #define STATIC
#endif
...
static int buffer__get_size(void);
```

3.5.6. PRAGMA prefix_all

This **pragma** may be used to force the use of the "dot" notation for **Internal Operations**.

scope:
The scope of this **pragma** is the whole **Design**. It is recommended to set it for the **Root Component** only.

example:
Assuming that with no **pragma** *prefix_all* set, the following code is generated in the source file of the **Component** `calculator`, for the **Internal Operation** `process_data`:

```
int process_data(sensor__T_signal value);
```

It will become as follow if **pragma** *prefix_all* is set:

```
int calculator__process_data(sensor__T_signal value);
```

restriction:
This **pragma** has no effect if the **pragma** *prefix_none* is set.

note:
Removal of the "dot" notation for all the other **Internal Features** can be performed directly within the corresponding coding sections of the design data structure.

3.5.7. PRAGMA prefix_none

This **pragma** can be used to prevent the code generator to insert any "dot" notation within **Provided Features** declarations and definitions. Doing so, the whole program will use the same name space.

<u>scope:</u>
The scope of this **pragma** is the whole **Design**. It is recommended to set it for the **Root Component** only.

<u>note:</u>
This **pragma** has no effect on the code that is entered manually within the coding sections of the design data structure. If references to remote **Features** uses the "dot" notation, whereas the **pragma** *prefix_none* is set, this will lead to linkage errors.

Ellidiss

www.ellidiss.com

www.ellidiss.com
stood@ellidiss.com

TNI Europe	**Ellidiss Technologies**
Triad House	
Mountbatten Court	24 quai de la douane
Worall Street	29200 Brest
Congleton	Brittany
Cheshire	
CW12 1DT	
UK	**France**
+44 1260 291 449	+33 298 451 870