GenoM

User's Guide

Sara Fleury
Matthieu Herrb

Contents

G^{en}oM? What for?

G^{en}oM (**Gen**erator **o**f **M**odules) is a development framework that allows the definition and the production of modules that encapsulate algorithms. A module is a standardized software entity that is able to offer services which are provided by your algorithms. Modules can start or stop the execution of these services, pass arguments to the algorithms and export the data produced.

Now you might ask yourself: "why should I bother integrating into modules my own algorithms that *do* work very well?". That's a pretty good question and this introduction will try to advocate on that point and give you some answers.

Your algorithms aim to being embedded into a *target* machine — let's say: a robot. You might not embrace the way this machine works in its whole and, most important, your algorithms will be integrated into a more general software system that includes other algorithms developed by other persons. This set of algorithms share several common properties: they must be configured (don't you have a bunch of parameters you want to adjust?), they must be started, interrupted, started again or stopped and we might expect them to exchange data and communicate with other part of the system.

Consider the example of a mobile robot: depending on the requirements of its mission and the current context of execution, the robot might need to acquire an image, localize itself, build a local map with some sensors and move. If the environment is rather free, the robot could plan a trajectory, but it could also decide to move on the basis of the local data given by its proximetric sensors. To be able to schedule these rather complex (and uncertain!) actions, it is *necessary* to define a protocol that can handle tasks at an abstract level. This protocol will let the robot:

- start an action when it is needed;
- stop an action in a clean manner;
- pass a set of parameters and data to the action;
- coordinate several actions;
- get the results of these actions;
- handle the failures of the actions (yes, actions can fail!) an keep them from taking the whole system with them when they spiral down. Failures can be as general as:
  - low batteries,
  - incorrect algorithm parameters,

– not enough memory to handle that case,
– the target to localize does not appear in the images,
– the algorithm cannot handle that situation yet,
– ...

Therefore, the general concept of *module* and *standard protocols* have been defined. These generic modules can encapsulate almost every kind of algorithm: periodic or aperiodic, synchronous or asynchronous, interruptible or uninterruptible, and even yours!

Of course, you could yourself write a module on the basis of that generic model. But that's a long and difficult story: you will have to port your software on the different systems you want it to run on, you will have to write test procedures to check that your module behaves correctly in every situation, ... and GenoM already does it for you!

The generator of modules comes with a description *language*, and standard templates. The templates will let you describe your module, the services it can offer, and for each service the list of expected parameters, the algorithms (yours!) that will be executed, the results along with their description, the failure messages and a few other items.

With the template file and the code of your algorithms — sorry, you still must write it yourself — GenoM produces:

- *a complete module* that can run on several flavor of Unix or VxWorks,
- *interface libraries* that will let you use the services of the module and get their results back,
- *an interactive test program* that let you send several requests to the module and trigger the execution of the corresponding services.

Now that you have an idea of what GenoM can be used for, this manual will explain you *how* to actually do it. You will learn

1. **How to produce and use a first test module**, with a concrete example.
2. **How the generic modules work**, and **how they are structured**. In particular, some useful vocabulary is explained.
3. **How to describe your own modules**. The complete specification language will be explained.
4. **How to generate your modules**.
5. **How to integrate your algorithms into the modules**.
6. **How to use modules** (from an interactive program, from another module, ...).

Chapter 1

The `demo` module, an overview of G^{en}oM

A module is described with a particular syntax in a file whose name ends in "dot gen" (`.gen`). In this chapter, we will write a test module which we will call "demo". Thus the file describing this module will be called `demo.gen`. The next sections explain how such a module is generated and used.

1.1 Principle of the module generation

One can distinguish two main parts in a module:

- A *server*, which encapsulate algorithms and is generated automatically by G^{en}oM from the description file,
- The set of *algorithms* that you have developed, and that will be included into the module.

To combine these two parts and form a complete module, the definition of functions that will interface the server and the algorithms together is required (at the moment only `C` functions are supported). These functions are called "codels" (which stands for *code elements*), and they represent the *smallest* grain size the server will be able to manipulate.

The first time G^{en}oM is called, it generates empty codels. They are fully functional, but do nothing. It's up to you to fill them in with your own code.

To help you distinguish between the server and the codels, the corresponding files are placed in two separate directories. The first one is called `auto/` and contains all the code generated by G^{en}oM. The second directory is called `codels/` and contains the codels — initially a template generated by G^{en}oM but never touched again once you have filled them in.

The figure 1.1 shows a synthetic view of the separation between `auto/` and `codels/` and also represent a typical development cycle :

1. **module description:** edition of the `.gen` file,
2. and 3. server generation and compilation,

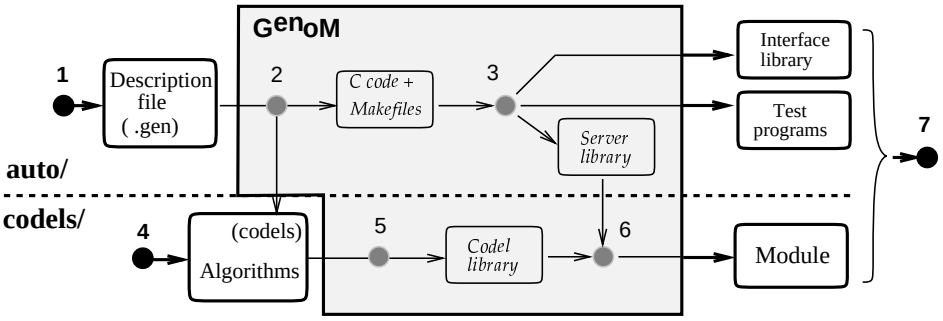


Figure 1.1: Development cycle and separation between sever and codels.

4. **write the codels that will invoke your algorithms**,

5. and 6. compile the codels and link with the server,

7. **test and use the module**.

Only the points **1**, **4** and **7** are in your charge. G^{en}oM writes for you the `Makefile`s and handle the compilation of the various files.

Once you have compiled a module, you can incrementally modify, add or remove services (back to the first point) and codels (back to the point number 4).

1.2 An example

This section will illustrate a concrete use of G^{en}oM with a "demo" module. This module will control a mobile that can translate on a 2 meters long rail. Some of the services the "demo" module offers are:

- select the speed (between two symbolic values `DEMO_SLOW` and `DEMO_FAST`),
- move the mobile for a given distance,
- read the current speed or position at any moment,
- suspend the motion,
- monitor particular positions and inform when the mobile goes through these positions.

To implement this, we first create a directory named `demo/`. In that directory we will write the description file `demo.gen`, which could look like this (see next page):

```
/* -------------------------- MODULE DECLARATION --------------------------- */

module demo {
   number:            9000;            /* module id: unique number */
   internal_data:     DEMO_STR;        /* C typedef (defined below) */
};


/* ------------- DEFINITION OF THE MODULE's INTERNAL DATABASE -------------- */

/* External definitions involved in the database definition */
#include "demoStruct.h"

/* The internal database */
typedef struct DEMO_STR {
   DEMO_STATE_STR   state;           /* Current state of the mobile */
                                     /* (position and speed) */
   DEMO_SPEED       speedRef;        /* Speed reference */
   double           distRef;         /* Distance reference */
   double           monitor;         /* Positions monitors */
}DEMO_STR;


/* ------------------ SERVICES DEFINITION: The REQUESTS -------------------- */

/* Control request: modify the default speed */
request SetSpeed {
   type:                control;             /* request's type  */
   input:               speed::speedRef;     /* input: speed chosen */
   c_control_func:      demoControlSpeed;    /* codel for validity checks */
   fail_msg:            INVALID_SPEED;       /* possible error messages */
};

/* Control request: return the default speed */
request GetSpeed {
   type:                control;             /* request's type */
   output:              speed::speedRef;     /* output: the speed */
};

/* Control request:  interrupt the mobile */
request Stop {
   type:                control;             /* request's type */
   incompatible_with:   MoveDistance;        /* request to interrupt */
};
```

continued on next page...

... continuation of previous page

```
/* Execution request: translate of a given distance */
request MoveDistance {
   type:                exec;                /* request's type */
   input:               distance::distRef;    /* input: distance */
   c_control_func:      demoControlDistance;  /* codel for validity checks */
   fail_msg:            TOO_FAR_AWAY;         /* possible error messages */
   c_exec_func_start:   demoStartEngin;       /* initialization codel */
   c_exec_func:         demoGotoPosition;     /* main codel */
   c_exec_func_end:     demoStopEngin;        /* termination codel */
   c_exec_func_inter:   demoStopEngin;        /* interruption codel */
   incompatible_with:   MoveDistance;         /* incompatible requests */
   exec_task:           MotionTask;           /* task (thread) executing
                                               * the codel */
};

/* Execution request: monitor a particular mobile's position */
request Monitor {
   type:                exec;                /* request's type */
   input:               position::monitor;    /* inputs: position */
   output:              position::state.position; /* outputs: actual pos. */
   c_exec_func:         demoMonitor;          /* main codel */
   fail_msg:            TOO_FAR_AWAY;         /* possible error messages */
   incompatible_with:   none;                 /* no incompatible requests */
   exec_task:           MotionTask;           /* task (thread) */
};


/* ------------------------ POSTERS DECLARATION  --------------------------- */

/* Poster that exports the current state of the mobile */
poster Mobile {
   update:               auto;
   data:                 state::state, ref::distRef;
   exec_task:            MotionTask;
};


/* --------------------- EXECUTION TASKS DECLARATION ----------------------- */

/* Only one task (or thread) */
exec_task MotionTask {
   period:               20;
   delay:                0;
   priority:             100;
   stack_size:           2000;
   c_init_func:          demoInit;
};
```

The file `demo.gen` is made up of five parts, each of them being identified with a keyword (these keywords are explained in detail in chapter 3):

1.	`module`	module declaration
2.	`#include` and `typedef struct`	`C` include statement for the definition of structures and declaration of the internal database
3.	`request`	requests definition: the five services offered by the module
4.	`poster`	posters definition: posters are exported data structures that let information on the mobile state be available for other modules
5.	`exec_task`	execution task declaration (a thread for Unix) that take care of codel execution

The `#include demoStruct.h` statement (the second part above) works as in `C` and includes the corresponding `C` header file. This file contains all the necessary `typedef` declarations for the definition of the internal database. These structures are then used by the `request` and the `poster` declarations.

In this example, the file `demoStruct.h` contains the definition of `DEMO_STATE_STR` and `DEMO_SPEED`. This file is preferably located in the same directory as `demo.gen`, since it contributes to the definition of the module interface.

```
#ifndef DEMO_STRUCT_H
#define DEMO_STRUCT_H

/* Current state of the mobile */
typedef struct DEMO_STATE_STR {
   double position;              /* current position */
   double speed;                 /* current speed */
}DEMO_STATE_STR;

/* Admissible speeds */
typedef enum DEMO_SPEED{
  DEMO_SLOW,                     /* low speed */
  DEMO_FAST                      /* high speed */
} DEMO_SPEED;

#endif /* DEMO_STRUCT_H */
```

1.3 Module generation

The generation step is done through the `genom` command invocation. When a module is to be generated for the first time, `genom` must be invoked with the `-i` option, which installs the initial files (in particular, it installs the codels templates). Here is a sample run, for the `demo` example:

```
r2d2[demo] genom -i demo
Module Generator GenoM
Copyright (C) LAAS/CNRS 1994-1996
No directory auto, install it ? (y/n) y
Entering directory 'auto'
perl demo.pl
Creating directory user
Makefile.vxworks created
Makefile.unix created
Makefile.protos created
Makefile created
demoType.h created
demoError.h created
...
demoPrintProto.h is created
prototype demoScan.c
demoScanProto.h is created
Creating directory codels
codels/ is installed, from now you can just call "gnumake"
Done.
```

Some comments on this run:

1. The `.gen` extension needs not to be explicitely given. GenoM appends it automatically.
2. With the `-i` option, GenoM asked for a confirmation:

   ```
   No directory auto, install it ? (y/n)
   ```

   This is the first and also the last time GenoM asks for it. This is also your last chance to abort the module generation. Answer `n` if you did not want to invoke `genom`.
3. GenoM creates two directories and a great number of files. You do not need to know them all, and they will be described later in this document (they are also described in the appendix C).

From now on, the module is ready to be compiled and run, but let's look at the result of the execution of `genom`:

```
r2d2[demo] ls
Makefile       codels/        demoStruct.h      demo_essay*
auto/          demo.gen       startDemo
```

GenoM created two new directories `auto/` and `codels/`, and three new files `Makefile`, `demo_essay` and `startDemo`:

- The `auto/` directory is entirely dedicated to GenoM (and that is the only such directory). It contains all the *server* code (see figure 1.1), and you do not need to look into it, except if you are looking for some specific information. This document will describe this directory later.

- Algorithms (or a part of them) are grouped in the directory `codels/`. It has been installed by the `-i` option. The files in that directory give you a template to start from, and also let G^{en}oM produce a module even if you still do not have written a single line of code. From now on, that directory belongs to you and G^{en}oM never goes into it again (except if you regenerate the module with the `-i` option).
- The `Makefile` file is also under your control. That's the main file which is used for the compilation of the module. By default, it lets you specify which target you want to compile for, but you can modify it to suit your needs.
- The file `startDemo` is a tiny script for VxWorks, and let you load your module and start it. It also belongs to you.
- The file `demo_essay` is a Unix script that starts an interactive test program for this module.

1.4 Module compilation

Attention: This description of the compilation step might be deprecated at any time.

The compilation step is very straightforward: first, you must edit the `Makefile` in the `demo` directory. Select the target hosts for which you want to build the module (Unix, VxWorks/ppc or VxWorks/m68k) and modify the `all` target accordingly. Then, you just need to invoke GNUmake in the `demo/` directory.

```
r2d2[demo] gnumake
-- Generate module
Module Generator GenoM
Copyright (C) LAAS/CNRS 1994-1996
Your module is up to date.

-- SERVER unix: server and client
...
-- CODELS unix
...
```

We can notice that the Makefile invoked G^{en}oM again. This allows to check if the module is still up-to-date (that's the case in the example above). If the `.gen` file (or a file on which it depends) were locally modified, the module would have been regenerated.

Consequently, you just have to call GNUmake to either regenerate or compile the module. It is not necessary to invoke "manually" G^{en}oM.

1.5 Module execution

Once the server and the codels are compiled and the link edition between them has been done, the `demo` module is ready to be executed. The module is located in the directory `codels/${TARGET}` (see figure 1.1), where `${TARGET} = ${MACHTYPE}-{OSTYPE}`. Under

Unix, this is an executable whose name is the name of the module (under Solaris this would be `codels/sparc-solaris/demo`) and under VxWorks this is an object file (for instance `codels/m68k-vxworks/demo.o`).

The next two sections present a step-by-step tutorial on how to run modules under Unix (section 1.5.1) or VxWorks systems (section 1.5.2).

1.5.1 Execution under Unix

Attention: The description of the execution under Unix might be obsoleted at any time.

Module startup:

1. Launch `h2 init` to initialize communication libraries.

```
r2d2[demo] h2 init
Initializing csLib devices: OK
Hilare2 execution environment version 1.0
Copyright (c) 1999 LAAS/CNRS
```

Note: if you get the following message, it is usually sufficient to answer `n`:

```
r2d2[demo] h2 init
Initializing csLib devices:
Cslib devices already exist on this machine.
Do you want to delete and recreate them (y/n) ?
```

2. Launch the module.

```
r2d2[demo] ./codels/sparc-solaris/demo -b
DEMO :
Spawn control task ... OK
Spawn task demoMotionTask ... OK
demo: All tasks are spawned
```

Note: adapt the path to your system (e.g. `i386/NetBSD` or `i386-linux`).

3. You're done! So... what?

So the module is running and ready to serve requests. We will see how this can be done with the small interactive test program `demoEssay` generated by G^{en}oM in the directory `auto/${TARGET}`.

Client startup: the interactive test program `demoEssay` This client can be launched with the shell script `demo_essay` in the main directory of the module.

```
r2d2[demo] ./demo_essay 1
```

Note: if you launch several clients, remember to change the number (we choose "1" in the example above).

Killing the module:

```
r2d2[demo] killmodule demo
```

At this point you can start a new instance of the module.

Cleaning everything:

```
r2d2[demo] h2 end
```

1.5.2 Execution under VxWorks

Module startup:

1. Log on the rack, using e.g. `rlogin`
2. Change to the module directory

```
cthulhu-> cd "/hilare2/src/modules/demo"
value = 0 = 0x0
```

   Note: check the returned value is '0'. If not, this is an error (you probably misspelled the directory name).

3. Load the module with `ld`

```
cthulhu-> ld < codels/m68k-vxworks/demo.o
value = 3436488 = 0x346fc8 = _semdemoInitExecTab + 0x12c
```

Note: adapt the path to your system (e.g. `ppc-vxworks`).

4. Launch the module

```
cthulhu-> demoTaskInit
DEMO :
Spawn control task ... OK
Spawn task demoMotionTask ... OK
demo: All tasks are spawned
value = 0 = 0x0
```

5. The module is running!

The module is ready to serve requests. We will see how this can be done with the small interactive test program `demoEssay` generated by G^{en}oM in the directory `auto/${TARGET}`.

Client startup: the interactive test program `demoEssay`

1. On the rack (*not* necessarily on the same CPU), you have to load the module's *client library*. This library can send requests to and receive answers from the module. It also contains the test program.

```
cthulhu-> ld < auto/m68k-vxworks/demoClient.o
value = 1289384 = 0x13aca8
```

2. Before you start the test program, you have to start on the remote station the utility `xes_server` that let you create terminals on your local station from VxWorks. Check your DISPLAY environment variable is correctly defined and do

```
pif[] xes_server
xes_server version 2.3
Copyright (C) LAAS/CNRS 1992,1994
```

3. On the CPU where you loaded the client library, you can now tell VxWorks where your `xes_server` lives (`pif` in this example).

```
cthulhu-> xes_set_host "pif"
value = 0 = 0x0
```

4. Last, launch the test program with the `sp` command, with a numeric argument between 0 and 9 indicating the instance number of this client.

```
cthulhu-> sp demoEssay, 1
task spawned: id = 0x1335d0, name = t1
value = 1258960 = 0x1335d0
```

If everything went well, a new terminal with a text menu appears on your screen. Each request you can send is associated with a number, and you can notice that the five first commands (from 0 to 4) correspond to the requests you defined in the file `demo.gen`. The special command labeled '-99' let you kill the module.

A detailed description of this interface can be found in the section 6.1, page 55.

Chapter 2

Modules description

This chapter will explain some vocabulary we use in this document, what are modules, and how they work.

2.1 Vocabulary and general description

A module lets you integrate your algorithms and functions as a set of services in a standard template. It then lets you access those services with a standardized interface. Produced data can also be retrieved in a standard way.

The services are controlled (*i.e.* parameterized, started or stopped) with *requests*, that represent the visible part of the module. Requests can be used either by an operator, another module, or any other program.

To each request can be associated an *input parameter* and an *output parameter* (`C` structures as of today). When a service ends, a *reply* is sent to the client who invoked the corresponding request. To each reply is associated an *execution report*: it describes the execution of the service and lets the client know about problems that might have occured.

There are two kind of requests: the *execution* requests and the *control* requests. Execution requests start an actual service, whereas control requests control the execution of the services. Control requests can set parameters, interrupt services, and so on... Each module has a predefined control request whose name is `Abort`: it can interrupt any running service as well as the module itself.

The execution time of a control request is considered to be zero. Thus, they have only one final reply, which is sent to the client immediately. On the other hand, execution requests can last for an arbitrary amount of time. Thus they send an *intermediate* reply, as soon as the service starts. The final reply is sent when the service ends.

A running service is called an *activity*. Some function, such as monitoring services, can start *several* activity and execute them simultaneously. Other kind of function, such as servoing functions, cannot handle parallelism and can only start *one* activity at a time.

Activities can control a physical device (sensors, actuators), use the services offered by other modules (by the mean of requests) or produce data. These data can be transfered at the end of the execution through the final reply, or at any time by the mean of *posters*.

A *poster* is a structure (`C` structure as of today) that is updated by an activity and shared in the global system. It can be read by any other component of the system (a module, an operator, ...).

Every piece of data that goes through a module (requests or posters) is stored in an internal database which is called *Functional Internal Data Structure* (fIDS for short — you might also find the acronym *SDIf* which is the French translation of fIDS).

2.2 Structure and functioning of modules

As shown in the figure 2.1, a module defines two Internal Data Structures: the functional IDS (fIDS) and the control IDS (cIDS) dedicated to the internal routines. The module also defines *tasks* (threads under Unix) that execute code. There are at least two tasks:

- One (or several) *execution tasks* that run your services and the algorithms they are made of,
- One *control task* that controls the module.

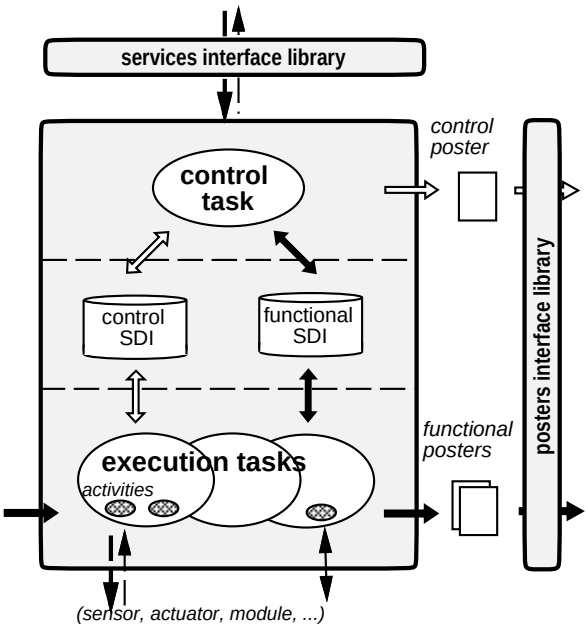


Figure 2.1: Structure of modules.

The control task:

You normally do not have to care about the control task, but it's a good idea to learn what it does. The control task

1. receives the requests for the module,
2. checks the validity of the input parameters and store them in the fIDS,
3. checks if the module can start a given service (handles conflicts between requests),
4. tells the right execution task to start the service,

5. and upon the termination of a service it sends back to the client any output produced by the service (the execution report and — if needed — the C structure declared for that service).

Beside this, the control task maintains a poster (the *control poster*) that contains informations on the current state of the module (running services, activities, and so on).

Conflicts between services are handled by the following procedure: if incompatible services are to be run at the same time, **the last request has the highest priority**. Activities that happen to be incompatible with that request are interrupted. This strategy matches a *reactivity* criterion and is systematically applied.

You must declare yourself which services are incompatible with which services. Note that a service is *very often* incompatible with itself; you must not forget to declare this.

Execution tasks:

Your own code is executed by the execution(s) task(s).

If this code is to be periodical (servoing, monitoring, filters, ...), you will have to use a periodical execution task and specify its period. It is also possible to use a-periodical tasks and a sequential scheduling. Tasks are given priorities (for VxWorks only at this time), depending on their constrains in terms of resources and CPU requirements.

In the `demo` example, there were only one execution task (`exec_task MotionTask`). This is usually sufficient since a single execution task can handle *several* activities in parallel. However, if several activities require different priorities or periods, you will have to declare several execution tasks.

Interface libraries:

Modules provide two standard interface libraries:

- A service library, which handles requests emission and reception,
- A poster library, which contains the necessary functions to read the module posters.

2.3 Integration of the algorithms: the codels

In order to associate your code to the requests, you have to tell G^{en}oM which are the functions that must be executed to handle requests. Your algorithms must be split into several parts (startup, main function, end, interruption, ...). Each of these parts is called a *codel* (elementary code). At this time, codels are `C` functions.

A module is the result of the link edition between the code G^{en}oM has generated and the codels libraries.

Chapter 3

Editing a module

Creating a new module implies writing two distinct parts: the description of the module (the `.gen` file) and codels. This chapter describes the first part, the `.gen` file.

3.1 Using the XEmacs mode `genom-mode`

It is strongly advised you use the `genom-mode` under XEmacs to write your module. Besides the syntactic coloring and automatic indentation, this mode defines several commands that create G$^{\text{en}}_{\text{o}}$M structures (requests, posters, ...). It also includes on-line help.

Commands can be accessed by three means:

- The (X)Emacs menu bar (`GenoM Mode Commands`)
- A pop-up menu with button 3 of the mouse
- The following keyboard shortcuts, beginning with `C-c` (control-c)

`C-c C-m`	create a new **m**odule (*first command to invoke*)
`C-c C-i`	**i**mport a structure
`C-c C-r`	create a **r**equest
`C-c C-p`	create a **p**oster
`C-c C-e`	create an **e**xecution task
`C-c C-b`	indent whole buffer
`C-c C-v`	check that every field is filled
`C-c C-d`	remove optional fields that are empty (**d**elete)
`C-c C-h ...`	on-line **h**elp

Commands that create a G$^{\text{en}}_{\text{o}}$M structure (request, poster, ...) prompt you for a name in the mini-buffer. Additional arguments may be requested, depending on the particular structure you are creating. Once you have supplied all the arguments, a template for the structure is inserted in the buffer. Optional fields are surrounded by single superior and inferior signs (< and >). Mandatory fields are surrounded by two superior and inferior signs (<< and >>). You can refer to the on-line help to know how to fill in these fields.

On-line help can be requested with the `C-c C-h` key sequence, followed by one of the following letter:

`h`	lists the available commands of `genom-mode`
`m`	describes **m**odules
`r`	describes **r**equests
`p`	describes **p**osters
`e`	describes **e**xecution tasks
`i`	describes structures **i**mportation
`g`	describes the module **g**eneration
`c`	describes the **c**odels
`C-h`	help on help (this list)

Help pages are made up of four parts:

- `What is a ...?`: general description of the G$^{en}_{o}$M structure
- `How to create a ...?`: how to create the structure
- `How to instantiate the fields ?`: how to fill in the template of the structure
- `Examples`: some examples.

Last, active zones (updated with the sequence `C-button2`) are defined for each request and each codel of the requests. When clicking (button 2) on these zones, the file containing the corresponding codel(s) is visited and the point is positioned onto the function. If the function does not exist, an empty template can be inserted if you want so (the module must be generated for this to work).

3.2 Writing a module

A module description contains five parts. The five section below will describe these parts, and use the `demo` module as an example. The five parts are:

1. Module declaration
2. `C` structures and fIDS declaration
3. Requests definition
4. Posters definition
5. Execution tasks declaration

All the G$^{en}_{o}$M structures (module, request, poster and task) use the same syntax: a keyword, which characterizes the structure, followed by a name and several fields enclosed between braces (`{` and `}`). The keyword is one of `module, import from, request, poster` or `exec_task`.

In this section, optional fields are surrounded by < and >. Mandatory fields are surrounded by << and >>. *Optional fields that are not instantiated must be removed before the module is generated.*

3.2.1 Module declaration

A module declaration looks like this:

```
module <<module-name>> {
    number:          <<module-number>>;
    internal_data:   <<SDI-type>>;
};
```

This part is mandatory and lets you choose a name for your module. Fill-in the field <<`module-name`>> and choose a number for <<`module-number`>>. This number must be a multiple of 10 greater that 1000. It should be unique and no other module should use the same one.

The last field you have to fill-in is <<`SDI-type`>>, and you must choose the name of a valid `C` type (this probably should be a typedef for a structure type). This will be the module internal database. `genom-mode` provides you with a default value for this field.

3.2.2 `C` structures and fIDS declaration

Note: the *functional internal data structure* (fIDS, or SDIf in French) is a `C` structure that contains all the requests input and output data, as well as the posters definition. When writing a module, this structure can be defined progressively by adding the requests parameters each time a new request is added.

Requests parameters, replies and posters

These are `C` structures you should define in `C` header files. You can include these headers with the `#include` directive, as in plain `C`. Since these headers are parts of the module definition, they should be located in the main directory of your module, in the same place as the `.gen` file.

These structures will be used by other modules. Thus, it is *strongly* advised you prefix their names with, e.g., the name of your module (see the example `demoStruct.h` in the section 1.2).

Lastly, it is also advised you protect your headers of multiple inclusion with the standard strategy:

```
#ifndef FILENAME
#define FILENAME
...
#endif  /* FILENAME */
```

You must respect three rules in order to get your header files working with G$^{\text{en}}_{\text{o}}$M:

1. **Allowed C types**: G$^{\text{en}}_{\text{o}}$M can parse *nearly* all `C` type declarations. The only unknown type is `void` and a few other constructions are forbidden: *i.* `union`s, and *ii.* recursive type definition as in `typedef B A` where `A` is a typedef itself. A workaround for the latter is to use a new structure:

```
typedef struct B {
   A a;
} B;
```

2. **Limitations on pointers**: Requests parameters, replies and posters will travel between several processes, outside the module, possibly on another machine. Given that, the notion of *pointer*, *address* or *list* does not make sense. They should not appear in this context (but you can use such data types internally in your codels).

3. **Alignment considerations**: The structures you define can potentially be transferred across several platforms. You must be aware that different systems do not align data in the same way. In order to avoid problems, you should align yourself your data on `double`s (8 bytes). The following example illustrates this:

```
typedef struct PILO_MOVE {
   int    percentSpeed;    /* percentage of max speed */
   int    padding;         /* ALIGNMENT */
   double distance;        /* distance to travel */
} PILO_MOVE;
```

External structures

It is possible to import, from other modules, external structure definitions. The corresponding headers are included with the `#include` directive but, in that case, it has to be put in an `import from` directive. This tells GenoM from which module the structures come, and avoid duplication of the functions that deal with these structures.

In the example below, the module `pilo` uses structures defined in the module `loco`:

```
import from loco {
#include "locoStruct.h"
};
#include "piloStruct.h"

typedef struct PILO {
     PILO_MOVE move;
     LOCO_REF  reference;
} PILO;
```

It is recommended that you do not hard-code the path to the external headers. Instead, use the option `-I` upon the module generation.

What should (and should not) the fIDS contain?

Requests parameters, requests replies and almost every poster will pass through the fIDS: thus, they must be declared in this structure. The fIDS is also a way to exchange data between tasks (or threads) inside a module. Conversely, data exchanged between codels of the *same* task only do not need to be declared here.

3.2.3 Requests definition

There are threes types of requests: control, execution and initialization. The three types are identified by the field `type` and one of the three keywords `control`, `exec` and `init`.

Examples of requests can be found in the chapter 1.

Control requests

They are defined with the keyword `control` in the field `type`:

```
request <<request-name>> {
     doc:                <"doc";
     type:               control;
     input:              <name>::<sdi-ref>;
     input_info:         <default-val>::"<name>", ...;
     output:             <name>::<sdi-ref>;
     c_control_func:     <codel>;
     fail_msg:           <msg-name>, ...;
     incompatible_with:  <exec-rqst-name>, ...;
};
```

- `doc` is a short string that describes the service usage.
- `input` and `output` define respectively the input parameter and the output parameter of the request. `name` is the name of this variable and `sdi-ref` the name of the corresponding member of the fIDS (e.g. `input: position::state.position`).
- `input_info` lets you define default values as well as a comment for *each* member of the `input` structure. This information is used for interactive requests invocation.
- `c_control_func` is a codel (`C` function) which is executed by the control task and which controls the validity of the input parameter.
- `fail_msg` is a list of possible reports returned by the control codel (the special report "OK" is always implicitly defined).
- `incompatible_with` is a list of requests of *this module* that are declared incompatible with this request. Activities corresponding to the listed requests will be interrupted upon invocation of this service. Two special keywords `all` and `none` let you declare all requests (or none) to be incompatible with this one.

Execution requests

They are defined with the keyword `exec` in the field `type`. As opposed to the control requests, those requests declare services that will be executed and they define a few more fields:

```
request <<request-name>> {
     doc:                 "doc";
     type:                exec;
     exec_task:           <<exec-task-name>>;
     input:               <name>::<sdi-ref>;
     input_info:          <default-val>::"<name>", ...;
     output:              <name>::<sdi-ref>;
     c_control_func:      <codel>;
     c_exec_func_start:   <codel>;
     c_exec_func:         <codel>;
     c_exec_func_end:     <codel>;
     c_exec_func_inter:   <codel>;
     c_exec_func_fail:    <codel>;
     fail_msg:            <msg-name>, ... ;
     incompatible_with:   <exec-rqst-name>, ... ;
};
```

- `exec_task` is the name of the execution task in charge of the codels execution.
- `c_exec_func_start`, `c_exec_func`, `c_exec_func_end`, `c_exec_func_inter` and `c_exec_func_fail` are the codels of this service. All fields are optional, but at least one of `func_start`, `func` or `func_end` must be defined. Codels are further described in chapter 5.
- All other fields serve the same purpose as in control requests. See previous paragraph for a description.

Initialization request

A special execution request is the *initialization request*. It is identified by the keyword `init` in the field `type` (all other fields are the same as for execution requests). This special request can be used to perform some initialization upon module startup. There can be at most one such request and the module will not accept to serve any other *execution* request until the `init` has been invoked. Control requests will still be served, and can be used to set several parameters used by the `init` request.

In order to allow the invocation of the init request from a standard shell (for instance as soon as the module is spawned), G^{en}oM builds an executable called `<module>Init` (where `<module>` is the name of the module) in `auto/${TARGET}`. This executable takes exactly as many parameters as in the structure declared in the input field, in the same order as they appear in the structure.

3.2.4 Posters definition

The posters let you export data, either automatically (you don't have anything to do) or "by hand" inside a codel. Data may be a member of the fIDS or not.

Data from the fIDS

```
poster <<poster-name>> {
     update:              <<update-type>>;
     data:                <<name>>::<<sdi-ref>>, ... ;
     exec_task:           <<exec-task-name>>;
};
```

- `update` indicates whether the poster is updated automatically (`auto`) or by a codel (`user`). The `auto` mode is usually chosen for periodical data such as a position.
- `data` is the list of data you wish to include in the poster. It is given in the same way as the input and output parameters of the requests: a name, followed by a reference to a member of the fIDS.
- `exec_task` is the task which owns the poster. This task is in charge of the update of the poster for `auto` posters. Note that only the task which owns the poster can change its content.

The data structure of the poster is a concatenation of the list of declared data. The corresponding `C` type is defined by G$^{\text{en}}_{\text{o}}$M in the file `auto/<module>Poster.h` and its name is `<MODULE>_<POSTER>_STR` (all uppercase) where `<MODULE>` is the name of the module and `<POSTER>` the name of the poster.

For instance, the `Mobile` poster of the `demo` module (chapter 1) is defined as follow:

```
typedef struct DEMO_MOBILE_POSTER_STR {
  DEMO_STATE_POSTER_STR state;
  double ref;
} DEMO_MOBILE_POSTER_STR;
```

Other data

Data exported by posters are not necessarily members of the fIDS. This can be the case if *i.* data structures are big: it is not advised to put them in the fIDS and copy them several times, *ii.* data structures do not have a predefined size, as for lists for instance.

For this kind of posters, two new fields are defined:

```
poster <<poster-name>> {
     update:              user;
     type:                <<type-name>>;
     exec_task:           <<exec-task-name>>;
     c_create_func:       <codel>;
};
```

- `type` is the name of the `C` type of the data structure.

- `c_create_func` optionally designates the name of a `C` function which is used to create the poster structure. If it is not given, the module performs the memory allocation by itself, using the size of the given `C` type.

3.2.5 Execution tasks declaration

```
exec_task <<exec-task-name>> {
     period:              <number>;
     delay:               <number>;
     priority:            <<number>>;
     stack_size:          <<number>>;
     c_init_func:         <codel>;
     c_end_func:          <codel>;
     c_func:              <codel>;
     cs_client_from:      <module-name>, ... ;
     poster_client_from: <module-name>::<poster-name>, ... ;
     fail_msg:            <msg-name>, ... ;
};
```

- `period` (optional) defines a periodical task. The period is given as an integer number in `ticks` (at the moment, a tick is $5ms$ under VxWorks and $10ms$ under Unix). *Take care:* the period must be *a divisor or a multiple* of 20. (e.g. 1, 2, 4, 5, 10, 20, 40, 60, ...).

- `delay` (optional integer in *ticks*). All periodical tasks with the same period will wake up at the same time. The delay can be used to delay the waking up of a particular task by the amount of ticks specified. `delay` can be `none` for a-periodical tasks.

- `priority` is used by the scheduler of the operating system. It is an integer between 0 (highest) and 255 (lowest). Priorities must be used to make sure that tasks with strong real-time constraints will match their requirements. A common strategy is to use a priority roughly "proportional to the inverse" of the period.

- `stack_size` is the size (in bytes) of the stack for this task. The size you need depends essentially on the size of the local variables you use. A stack which is *too small* will produce unpredictable results, so be sure to largely **overestimate** what you need. A good choice is usually 20.000 bytes. You can then use utilities like `checkStack` for VxWorks to obtain a better estimate. Note that under Unix, stack size are not used at this time (the stack is grown dynamically).

- `c_init_func` is the initialization codel. It is called only once, just before the module is ready to answer requests. It can be used to initialize internal variables (see also the *init request*, which can be used if the initialization requires user inputs).

- `c_end_func` is the symmetric of the `c_init_func`. It is called once, just before the module exits.

- `c_func` is a *permanent* codel. It is executed each time the execution tasks wakes up. Thus, for a periodical task, it is also periodical.

- `cs_client_from` is a list of modules you wish to send requests to. It might be deprecated in future releases.
- `poster_client_from` is a list of posters from other modules that you wish to use in this module. This list is made up of coma separated items, where each item is of the form <module>::<poster>. It might be deprecated in future releases.
- `fail_msg` is a list of reports that can be reported by the permanent activity `c_func`. Since this activity does not belong to a request, its reports are stored in the *control poster* of the module.

Chapter 4

Module generation

4.1 The `genom` command

Synopsis

`genom [-ictxnd] [-I`*path*`] [-D`*macro*`] <module>[.gen]`

Description

`genom` is the command that generates a module. The `module` argument is the name of the file which contains the description of the module. The `.gen` is optional and is automatically appended if omitted.

`genom` accepts the following options:

- `-i` *Installs the directory* `codels/`*:* you should use this option when generating the module for the first time. When called with `-i`, `genom` will install the directory `codels/` as well as templates for the codel files in this directory. It will also create all the `Makefiles` (in the main directory and in the `codels` directory) used to compile the module. If the files that would normally be installed are already present, `genom` will ask for confirmation before overwriting files.
- `-c` *Conditional regeneration:* module is regenerated only if files from which it depends have been modified since last generation (*i.e.* the `.gen` file or files it includes).
- `-t` *Tcl libraries generation:* Generates Tcl interface libraries. They are mandatory if you wish to control this module from a tcl interpreter (see section 6.2).
- `-x` *Propice libraries generation:* Generates Propice interface libraries. They are mandatory if you wish to control this module from a Propice program.
- `-n` Generates the perl script that is used to generate the module, without actually executing it.
- `-d` Turns on debugging mode inside the *yacc* parser.
- `-I`*path* Defines paths for included files (same as `-I` option of `cc`).
- `-D`*macro* Defines macros as for `cc`.

Once the module has been installed (`-i` option) for the first time, you just have to invoke `make` (GNUmake is required) in the main directory to regenerate or compile the module.

The `-i` option modifies files in the `codels/` directory. Use it carefully. If you wish to manually get a new template file for the codels, you can find them in the directory `auto/user/`. These files are always up-to-date. Also consider the XEmacs mode `genom-mode`, which lets you insert codel templates into existing codel files (see chapter 3).

4.2 Product of the generation

The module generation produces files in the two directories `codels/` and `auto/`. The files located in the `codels/` directory are described in the next chapter. This section describes the files located in the `auto/` directory. The `demo` module is used as an example: you can always replace the string `demo` by the name of your module.

Compiling these files produces binary objects and executables in the `${TARGET}` sub-directories. See appendix C for information on the file-system hierarchy.

4.2.1 Interface libraries

Requests library: `demoMsgLib`

This library implements the basic requests functions (request emission, replies reception) and is used by the clients of this module.

The `C` source code of this library can be found in `demoMsgLib.c` and the prototypes in `demoMsgLibProto.h`. The header file `demoMsgLib.h`, which must be included by clients, contains the definitions for the server identification and communication establishment (name and size of the mailbox, ...).

To use this library, you must link your program with `demoMsgLib.o` under VxWorks (note that the library `demoClient.o` already includes `demoMsgLib.o`) and `demoClient.a` under Unix.

Posters library: `demoPosterLib`

This library implements the basic posters functions for this module (read and display functions). The same structure as above is used: `demoPosterLib.c` is the source code and `demoPosterLibProto.h` the prototypes definition. `demoPosterLib.h` defines the name of the posters along with their data structures. The *cIDS* structure is also defined there[1].

To use this library, you must link with `demoPosterLib.o` (or `demoClient.o` if you want to use posters and requests), and `demoClient.a` under Unix.

4.2.2 Useful header files

The file `demoHeader.h` must be included in every codel file. It contains the definitions of several constants that characterize the module (name, period, posters, ...) as well as the two macros that let you access the *IDS* (*SDI_F* and *SDI_C*).

The file `demoError.h` contains the definitions of the error codes (reports declared in the `fail_msg` field) of this module.

[1] the structures included by the *cIDS* are defined in the file `modules.h`, located in the genom source tree.

The file `demoType.h` defines the actual *IDS* structure. It is not necessarily the same as in the `.gen` file, especially if the module defines reentrant requests (*i.e.* compatible with themselves).

4.2.3 Tcl library

The files `demoTcl.c` and `demo.tcl` are used by Tcl to control the module. See section 6.2 in this document.

4.2.4 Propice library

The files located in the `propice/` directory are used by Propice to control the module. See section 6.3 in this document.

4.2.5 Executables: server, test program, initialization request

The executable `demoEssay` is an interactive test program. To use it, you just have to load `${TARGET}/demoClient.o` (VxWorks) or run `${TARGET}/demo_essay` (Unix).

The files `demoCntrlTask.c`, `demoMotionTask.c` and `demoModuleInit.c` contain the module's execution tasks source code. Once compiled, they produce the files `demoModule.o` (VxWorks) and `demoServer.a` (Unix). The file `demoModuleInit.c` produces the function `demoTaskInit` which spawns the module.

The file `demoInit.c` contains the code that invokes the *Initialization* request of the module (if the module defines one). To use it, run `demoInit` (under Unix, the executable is in `${TARGET}` and under VxWorks, it is contained in the `demoModule.o` object file).

4.2.6 Other files

The files `demoScan.h` and `demoPrint.h` contain the definition of the interactive functions that scan the arguments of the module requests. These functions are used by the test program `demoEssay`.

Appendix C gives an exhaustive list of the files produced by G$^{\text{en}}_{\text{o}}$M.

Chapter 5

Writing the codels

Codels are `C` functions (at this time) that interface a module and your algorithms: the module executes the codels, which, in turn, execute your own functions. In particular, codels can retrieve the requests parameters, map them into useful data for your functions, and then call these functions.

5.1 Different kinds of codels

5.1.1 Codels associated to requests

Sending a request to a module ends up in executing the code of the codels associated to the request. Two types of codels can be distinguished:

- **control codels** (defined with `c_control_func`) are executed by the control task. They are essentially used for checking the validity of the input parameters of the request, just before these parameters are actually written into the fIDS.
- **execution codels** (defined with `c_exec_func*`) are executed by an execution task and represent the actual action of the service. Their execution create an activity, which lasts until completion of the service. These codels exist only for execution requests.

The parameters of the `C` functions associated to the codels are the `input` and `output` structures of the request. The input data can thus be passed to your functions, and you can write the results into the output structure.

5.1.2 Codels associated to execution tasks

Three codels can be optionally defined for each execution task:

- **Initialization codel** (`c_init_func`). This codel is executed only once, when the execution task is initialized and just before it starts serving requests. One can use this codel to initialize the fIDS, and set default values to parameters.
- **Termination codel** (`c_end_func`). This codel is executed by the execution task upon destruction of the module, just after it has stopped serving requests.
- **Permanent codel** (`c_func`). This codel is executed each time the task wakes up (therefore periodically if the task is periodic). This codel creates a permanent activity.

5.2 Simple examples of codels

5.2.1 Example of control codel

Control codels are used to check the validity of input parameters of a control or an execution request. They can prevent entering erroneous values into the fIDS. For execution requests, they can also check that the module is in an adequate state before executing the requested service.

Control codels take the input parameter of the request as input and must return either `OK` if the parameter is valid, or `ERROR` if it is not. Warning: in the latter case, you must have defined an error code and you must set it before returning `ERROR`.

If the codel returns `OK`, the input parameter is copied into the fIDS and the execution continues. If the codel returns `ERROR`, the parameter is not copied into the fIDS and the final reply is sent back to the client, along with the report which has been set by the codel. For an execution request, the activity is not started. The error code is fundamental: if it is not set, the client will have no idea of what happened.

As an example, here is the control codel `demoControlSpeed` of the request `SetSpeed` of the module `demo`. This request expects a speed as input (structure `DEMO_SPEED`). Therefore, the codel takes a pointer to this structure as first parameter. The structure `DEMO_STR` is defined in the file `demoStruct.h` (see chapter 1).

```
STATUS
demoControlSpeed(DEMO_SPEED *speed, int *report)
{
   /* Refuse *speed if the value is erroneous */
   if (*speed != DEMO_SLOW && *speed != DEMO_FAST) {
        *report = S_demoCntrlTask_INVALID_PARAMETER;
        return ERROR;
   }
   /* Parameter is valid: it will be entered into the fIDS */
   return OK;
}
```

5.2.2 Example of execution codel

Execution codels are always associated to an execution request. They perform the actual actions of the service.

For this first example, we will write a request that compute the norm of a 2-dimensional vector. The standard math library already has such a function:

```
double
hypot(double x, double y);
```

What we have to do now is to add to the module a request which we call *Hypot*. An execution codel will do the actual computation, and call `hypot()`. The input parameter will be a structure which will contain two members, `x` and `y`: we call it `DEMO_VECTOR_STR`. The output parameter is a single *double*.

In the file `demo.gen`, we write:

```
/* fIDS declaration */
typedef struct DEMO_STR {
   DEMO_STATE_STR     state;          /* Current state */
   DEMO_SPEED         speedRef;       /* Speed reference */
   DEMO_VECTOR_STR    vector;
   double             norm;
   ...
};

/* Hypot request */
request Hypot {
   doc:                "compute sqrt(x*x+y*y)";
   type:               exec;                   /* execution request */

   input:              vector::vector;         /* vector (x, y) */
   input_info:                                 /* default values and */
      0.0::"X coordinate",                     /* description of */
      0.0::"Y coordinate";                     /* parameters */

   output:             norm::norm;             /* norm (result) */
   c_exec_func:        demoHypotExec;          /* execution codel */
   exec_task:          MotionTask;             /* execution task */
   incompatible_with:  Hypot;                  /* incompatibilities */
};
```

In the file `demoMotionTaskCodels.c`, which contains all the codels for this task, we write the `demoHypotExec` codel:

```
ACTIVITY_EVENT
demoHypotExec(DEMO_VECTOR_STR *vector, double *norm, int *report)
{
   *norm = hypot(vecteur->x, vecteur->y);
   return ETHER;
}
```

The `return ETHER` statement tells GenoM that the activity is terminated: the client will get the final reply. We will see later the other values that can be returned at the end of an execution codel.

The next step is to compile the module, and link it with the `hypot()` function. In this case, `hypot` is a function of the standard library, so there's nothing special to do. But it the function were a function of your own, defined in a non-standard library, you would have to edit the Makefiles and complete the variables:

- `CPPFLAGS` for the path to the headers of your library.
- `LIBS` for the path to the library itself.

Here is what it could look like with our example and the Makefile in the `codels/` directory:

```
[...]
CPPFLAGS += -I$(DEMO) -I$(DEMO)/auto -I$(DIRUNIX)  -I$(DIRGENOM)
CPPFLAGS += -I/usr/include
[...]
LIBS = /usr/lib/libm.a
```

If your external functions were defined in a `C` file in the `codel/` directory (instead of in an external library), you would simply add this file to the list of files to be compiled into the codels library:

```
[...]
SRCS = \
        demoCntrlTaskCodels.c \
        demoMotionTaskCodels.c
SRCS += hypot.c
[...]
```

This simple example showed how to integrate your algorithms into codels. The sections 5.7 page 44 and 5.8 page 47 will present more complex examples.

5.3 Codel files and compilation

5.3.1 Splitting codels into several files

To help you write the codels, the `-i` option of `genom` generates empty templates in the `codels/` directory. Then, you just have to complete the templates.

Note that if you use the `-i` option and those files already exist, `genom` asks for confirmation before overwriting any file. It is not possible to fuse locally modified templates with fresh new ones. However, the templates are always generated in the `auto/users/` directory, so that you can fuse parts together by yourself. The `genom-mode` can help you to do this: see chapter 3.

By default, there is one codel file per task. Control codels are grouped in the source file `demoCntrlTaskFunc.c` and execution codels are grouped in the the source file `demo<Exec Task>Func.c`, where `<Exec Task>` is the name of the execution task for those codels.

You are free to define additional files, or change the initial organization. Be sure you update the `SRC` variable in the Makefile to reflect your changes.

5.3.2 Makefiles

Several `Makefile` are generated by `genom`. By default, they compile your codels files, and perform the link edition with the module server (`demoModule.o` for VxWorks and `demoServer.a` for Unix).

Each file is compiled in the `${TARGET}` subdirectory. For unix, the compilation produces the executable `demo` and for VxWorks the object file `demo.o`.

You can change the standard Makefiles to suit your needs. In particular, you can:

- add files to the `SRC` list.
- add libraries to the `LIBS` list.
- add compilation flags such as `-I` or `-D` to the variable `CPPFLAGS`.

5.4 Accessing the IDS

You can access this structure from a codel at any time. A mutual exclusion lock protects the structure against concurrent accesses.

5.4.1 The fIDS

The macro `SDI_F` defined in the `auto/<module>Header.h` header represent a pointer to the fIDS of the module.

Warning: the `input` and `output` parameters of the requests can be those of simultaneous activities (if the request is compatible with itself). In that case, these parameters *are not* pointers to members of the fIDS structure declared in the module. Each activity must have its own set of parameters, and G$^{\text{en}}_{\text{o}}$M generates arrays for that case. The member you have defined in the *fIDS* still exists, but is not used. Thus, you should always avoid reading parameters of a request directly from the fIDS, except if that request is incompatible with itself.

5.4.2 The cIDS

The cIDS contains various parameters of the module (period, poster names, clients ids, ...) and its current state (activities, ...). You can read the members of this structure thanks to macros defined in `auto/<module>Header.h`.

Here is a non-exhaustive list, for a module `pilo` client of another module `loco` and with an execution task `Motion` which updates a poster `Ref`:

name	value
`PILO_MOTIONTASK_NUM`	Id of the exec. task `MotionTask`
`PILO_REF_POSTER_ID`	Poster `Ref` Id
`PILO_MOTION_LOCO_CLIENT_ID`	Client id for `loco`
`CURRENT_ACTIVITY_NUM(`i^*`)`	Current activity number
`EXEC_TASK_PERIOD(`i^*`)`	Execution task period (seconds)

($*$) i is the id of the execution task, for instance `PILO_MOTIONTASK_NUM`.

5.5 Reports

In real situations, every action can fail. On a machine which interacts with its environment, it is *necessary* to think of every such abnormal situation, in order to protect the whole system. In case of an execution error, it is necessary to *i.* restore a sane state locally, to be able to restart another execution and *ii.* report a precise information to the client so that it can take appropriate decisions.

The list of abnormal situations is defined with the field `fail_msg` of every request. These are strings, wich are meant to be human readable: for instance `INVALID_PARAMETERS`,

`POSTER_NOT_FOUND`, `NOT_ENOUGH_MEMORY`, `IMPORTANT_DRIFT`, `CASE_NOT_MANAGED`, `SOLUTION_NOT_FOUND`, ...

Reports must be precise: avoid strings such as `ERROR`. However, it is not necessary to recall the name of the request: the client knows it already. Thus, a report `INVALID_PARAMETERS` could be used for several requests without loss of information.

Modules must also restore themselves into a sane state and handle correctly future requests. For instance, in the case of a `NOT_ENOUGH_MEMORY`, the module should free every unused memory in order to be able to process less demanding requests. If the failure is such that the module cannot restore a sane sate, it should put itself into the `ZOMBIE` state and wait for a human debugger.

5.5.1 Numerical values of reports

Reports which are declared in the `fail_msg` field are mapped into 32 bits integers with the VxWorks format.

The name of this integer is build as follow: `S_<source>_<report>`, where `<source>` is the name of the task or the library which defines the `<report>` string. For instance: `S_demoCnrtlTask_INVALID_PARAMETER`.

The numerical value is computed as follow:

- the 16 highest significative bits encode the id of the task or the library which defines the report. This number is the id of the module N, $N+1$ for the requests library, $N+2$ for the posters library, $N+3$ for the first execution task, ...
- the 16 lowest significative bits encode the id of the report inside the module, and is computed by $G^{en}_{o}M$.

Error codes are stored in the file `auto/<module>Error.h`.

5.5.2 Installing the reports

Warning: this procedure might be deprecated.

By default, clients will display the numerical value of the reports. However, it is possible to install reports in a global database. Client can then display reports as human readable strings. To do so, you must use the command `h2addErrno` located in the directory `/usr/local/hilare2/src/errorLib`:

```
pif[demo]  cd /usr/local/robots/posix/src/errorLib
pif[errorLib] h2addErrno /usr/local/hilare2/src/modules/demo/auto/demoError.h
pif[errorLib] gnumake
```

5.5.3 Displaying the reports

Warning: this procedure might be deprecated.

To display a report in clear text, use function of the library `h2errorLib` (installed in `/usr/local/hilare2/src/errorLib/`).

`void h2printErrno(int bilan);`	display the string associated to the report
`char const *h2getMsgErrno(int numErr);`	return the string associated to the report
`void h2perror(char *string);`	display the string `string` followed by the the string associated to the last report (`errno` variable)

5.6 Updating posters

Posters which are declared as `user` must be updated by the codels. There are several ways to do so: depending on the type of poster, you can use functions of the poster library `posterLib` (see appendix B) or the function generated by G^{en}oM (see below). The function `posterWrite` is of particular interrest:

```
int posterWrite(POSTER_ID posterId, int offset, char *buf, int nbytes)
```

`posterId` is the poster id: e.g. `PILO_REF_POSTER_ID` for the poster `Ref` in module `pilo`. `posterWrite` writes `nbytes` from buffer `buf` in the poster structure, starting at offset `offset`. It returns the number of bytes actually written, normally `nbytes`.

Consider the example of poster `Mobile` in module `demo`. Its structure is:

```
typedef struct DEMO_MOBILE_POSTER_STR {
  DEMO_STATE_STR state;
  double ref;
} DEMO_MOBILE_POSTER_STR;
```

Example 1: you update the `ref` member. Since it is not at the beginning of the poster, you must compute the offset:

```
DEMO_MOBILE_POSTER_STR *mobile;
int offset;
double ref;
...
offset = (char *)(&mobile->ref) - (char *)(&mobile);
posterWrite(DEMO_MOBILE_POSTER_ID, offset, &ref, sizeof(ref));
```

Example 2: G^{en}oM produces a function that do the same as the above example:

```
double ref;
...
if (demoMobileRefPosterWrite(DEMO_MOBILE_POSTER_ID, &ref) == ERROR) {
     /* stop ... */
}
...
```

Another more practical method consists in getting the actual address of the poster structure. This is done thanks to the `posterAddr` function. This function returns a pointer to the structure of the poster, and you can write into it directly. Be sure to protect writings to the poster with `posterTake` before writing anything in it and `posterGive` once you are done.

The three functions take one parameter: the poster id. `posterTake` takes another argument: `POSTER_WRITE` or `POSTER_READ`, for accessing the poster in write or read mode.

```
static DEMO_STATE_POSTER_STR *addrPosterMotion = NULL;

/* Get the poster address */
addrPosterMotion = posterAddr(DEMO_STATE_POSTER_ID);
if (addrPosterMotion == NULL) {
   *report = errnoGet();
   return ERROR;
}

/* Update the poster */
posterTake(DEMO_STATE_POSTER_ID, POSTER_WRITE);
addrPosterMotion->state.speed = state.speed;
posterGive(DEMO_STATE_POSTER_ID);
```

5.7 Splitting algorithms into codels

As we have already mentioned, algorithms must be integrated into codels. In the simple example presented in the beginning of the chapter, the `hypot` function was put in a single codel. For more complex algorithms, it will be necessary to split the functions call into several codels. You will have to write the succession of codel calls.

The choice of the number of codels is generally not unique. But it is important to keep in mind that a codel is the *smallest* entity that a module can handle. In particular, a codel cannot be *interrupted*: during its execution the module cannot do anything else. The way you split your algorithms will thus determine the latency of the module.

Consider the two most common classes of codels: periodical codels and aperiodical codels.

5.7.1 Periodical codels

For periodical codels (servoing, monitoring, filtering, ...), the same function must be called periodically. To do so, an execution request will be associated to a periodical execution task and the codel will be invoked at each period.

Typically, this codel will be the *execution codel* of the request (`c_exec_func`). To let the execution task call the codel periodically, the latter must let the task know that the activity is not finished when the codel returns. This is done by returning `EXEC` instead of `ETHER`. The activity stays in the state `EXEC` and the execution codel will be called at the next task's wake up.

Consider request `Monitor`, which monitorates the position of the mobile and throw an alert when it has the requested value, say, `myPos`. The execution codel `demoMonitor` (see below) is invoked (`return EXEC;`) until the position `myPos` has not been reached with the

DEMO_THRESHOLD precision. Once this position is reached, the activity ends (`return ETHER;`) and the final reply is sent to the client.

```
ACTIVITY_EVENT
demoMonitor(double *monPos, double *pos, int *bilan)
{
   /* Get the current position */
   *pos = SDI_F->state.position;

   /* Test if we are in the monitored zone */
   if (fabs(*monPos - *pos) > DEMO_THRESHOLD) {

      /* if not, we continue with the same codel */
      return EXEC;
   }

   /* The mobile is in the goal area: end the activity */
   return ETHER;
}
```

In this example, we used only one execution codel, which was invoked periodically.

If an algorithm had required an initialization step (e.g. for a sensor, a variable, starting another service, ...), this would have been done thanks to the *start* codel (`c_exec_func_start`). This codel corresponds to the `START` state, and is always executed first if it is defined. At the end of this codel, we would `return EXEC;` to switch to the `EXEC` state, or `ETHER` if we want to stop everything.

Similarily, a termination phasis would be defined thanks to the *termination codel* (the `c_exec_func_end` field) associated to the `END` state. To indicate, at the end of the execution codel, that we want to go through this termination state, we just have to `return END;` instead of `ETHER`.

5.7.2 Aperiodical codels

In the case of an aperiodical codel (e.g. a trajectory planning), the request associated to the codel will be associated to an aperiodical execution task. The codel could be made up of only one part, but if its execution took a while, it would be advised to split it into several atomic pieces. You can then use the different states as mentioned in the previous section to glue pieces together.

5.7.3 Interrupting a codel

An activity that goes through several codels (or several times through the same codel), can be interrupted. (as we already mentioned it, a single codel cannot be interrupted). When this occurs, the activity goes into the `INTER`rupted state, and the `c_exec_func_inter` codel is executed. By default, there is no such codel, and the activity goes immediatly into the `ETHER` state.

A sudden interruption can sometimes be problematic. You might for instance want to stabilize a dynamic system before actually stopping, free some memory, or stop another

activity which is correlated to the one which is beeing interrupted. In that case, define a `c_exec_func_inter` codel which will handle the necessary operations.

5.7.4 States and transitions of activities

The different states an activity can go through are shown on figure 5.1. The external ring correspond to the *normal* sequencing: `ETHER` → `START` → `EXEC` → `END` → `ETHER`. `START` and `END` states are optional. On any transition, one can go into the `INTER` state.

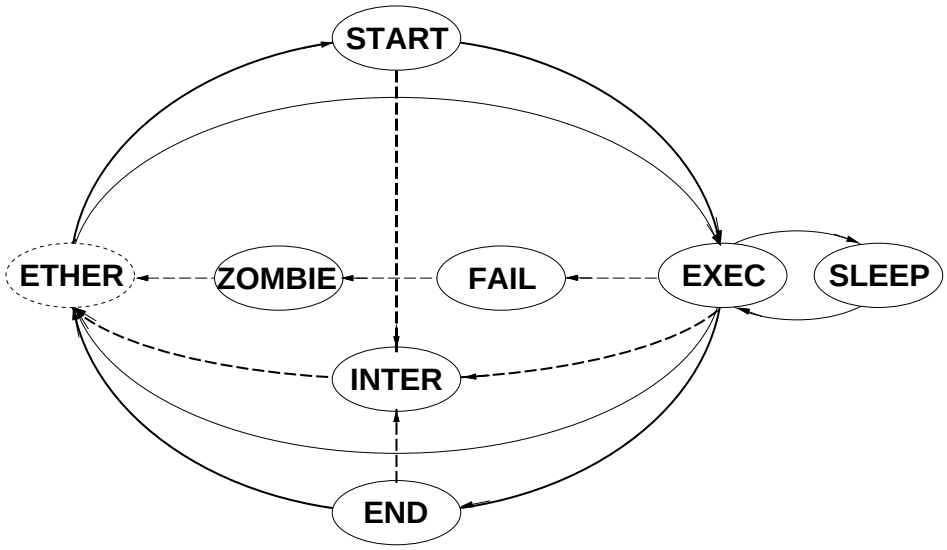


Figure 5.1: States and transitions of activities.

Note: in case of a problem, one can go into the `FAIL` state, or even directly into the `ZOMBIE` state. The module is then frozen.

As of today, the number of states is fixed and a future version might change this. However, a workaround is still possible with the current version, by using internal state variables. The current states are recalled in the following table:

state	comments	codel (if defined)
`START`	startup state	`c_exec_func_start`
`EXEC`	main execution state	`c_exec_func`
`END`	termination state	`c_exec_func_end`
`FAIL`	failure (and module freeze)	`c_exec_func_fail`
`INTER`	interruption state	`c_exec_func_inter`
`SLEEP`	suspended activity (waits an external event to go back into `EXEC`)	
`ETHER`	*terminated activity*	
`ZOMBIE`	*terminated activity and frozen module*	

State description:

- `START` is the first step of the execution. If the codel is not specified, the activity goes directly into state `EXEC`.

It is up to you to define the following transitions. To do so, codels must return one value of the enum `START`, `EXEC`, `END`, `ETHER`, `FAIL`, `ZOMBIE` or `SLEEP`, which corresponds to the state of the same name. One does normally follow the sequence `START` → `EXEC` → `END` → `ETHER`.

- `EXEC`, `END` and `INTER` are decribed in the previous sections.
- `ETHER` indicates that the activity does not exist anymore.
- `ZOMBIE` indicates that the activity stopped due to an abnormal situation. The module is then frozen and will not answer any requests anymore. A special request `Abort` let you resume the activity, which then goes into the `ETHER` state. This state can be useful if you want to debug some problem. It can also be useful if you want to re-synchronize two modules.
- `FAIL` terminates the activity, just before going into `ZOMBIE`. The codel can do some additional cleanup.
- `SLEEP` suspends an activity, and waits for an external event to occur (a request, or an `h2evn` event). Then it returns to the `EXEC` state.

The following table summarize the possible transitions, for each state:

		possible transitions					
state	(codel)	START	EXEC/SLEEP	END	ETHER	FAIL	ZOMBIE
START	(c_exec_func_start)	X	X	X	X	X	X
EXEC	(c_exec_func)		X	X	X	X	X
END	(c_exec_func_end)			X	X	X	X
INTER	(c_exec_func_inter)				X	X	X
FAIL	(c_exec_func_fail)					X	X

Note: a termination state (`END`, `FAIL` or `INTER`) is never interrupted.

5.8 Writing the codels

5.8.1 Control codels `c_control_func`

See example in section 5.2.1, page 38.

5.8.2 Execution codels `c_exec_func_*`

Execution codels have 1, 2 or 3 parameters, depending on the request. These codels can have, in this order and if the corresponding data is defined, a pointer to the input structure, a pointer to the output structure and a pointer to the report (always defined). Codels return an `ACTIVITY_EVENT`, as exposed in the previous section.

Here is the `demoGotoPosition` execution codel (`c_exec_func`) of the `MoveDistance` request of the module `demo`. This codel, invoked periodically, controls the speed of the mobile (according to the one specified with the `SetSpeed` request), and stops when the requested position is reached. We suppose we have two low level functions which control the mobile:

- `STATUS mobileState(double *position, double *speed)` which get the current status of the mobile (thanks to sensors) and

- `STATUS mobileMove(double position, double speed)` which move the mobile to the requested position at the requested speed.

```
ACTIVITY_EVENT
demoGotoPosition(double *goal, int *bilan)
{
   double remain;       /* remaining distance (m) */
   double speed;        /* requested speed (m/s) */
   double increment;    /* position change (m) */

   /* Measure current speed and position */
   if (mobileState(&(SDI_F->state.position), &(SDI_F->state.speed)) != OK) {
        *bilan = S_demoMotionTask_MOBILE_OUT_OF_ORDER;
        return ETHER;
   }

   /* Compute the remaining distance */
   remain = *goal - SDI_F->state.position;

   /* Get the reference speed */
   if (SDI_F->speedRef == DEMO_SLOW)
        speed = DEMO_SLOW_SPEED;
   else
        speed = DEMO_FAST_SPEED;

   /* Compute an elementary move, according to the speed and period */
   increment = speed * DEMO_MOTION_TASK_PERIOD;

   /* Are we done? */
   if (fabs(remain) < increment) {
        if (mobileMove(*goal, 0) != OK) {
            *bilan = S_demoMotionTask_MOBILE_OUT_OF_ORDER;
            return ETHER;
        }
        return END;
   }

   /* Continue */
   if (mobileMove(SDI_F->state.position +
                       SIGN(remain) * increment, speed) != OK) {
        *bilan = S_demoMotionTask_MOBILE_OUT_OF_ORDER;
        return ETHER;
   }
   return EXEC;
}
```

Notes:
The `SDI_F` macro let you acces the fIDS structure.
`DEMO_MOTION_TASK_PERIOD` is the period of the execution task (stored in the cIDS).
We will see later how one can access IDSs and how reports can be used.

5.8.3 Initialization codel `c_init_func`

One initialization codel can be associated to any execution task, by the mean of the field `c_init_func`. It is usually used to initiliaze the fIDS to a known state. It takes only one parameter: a pointer to the report. It returns either `OK` or `ERROR` (in that case the module does not start).

For the `demo` module, this codel chooses a default speed for the mobile and intializes its state. Note that fISD are not automatically initialized with zeros.

The constants and default values used by a module are usually defined in a header file in the main directory of the module. In that case this is `demoConst.h`.

```
STATUS
demoInit(int *bilan)
{
  SDI_F->state.position = 0.;
  SDI_F->state.speed = 0;
  SDI_F->distRef = 0;
  SDI_F->posRef = 0;
  SDI_F->speedRef = DEMO_SLOW;

  return OK;
}
```

5.8.4 Permanent activity codel `c_func`

A permanent activity can be defined for any execution task, by the mean of the field `c_func`. It is executed each time the task wakes up. It is usually used to set up a filtering function (pose computation, sonar echos reading, ...), or a permanent servoing activity which starts and stops with the module.

This codel takes only one parameter, a pointer on the report, and returns a `STATUS` (`OK` or `ERROR`). Be warned that if an error is returned, the execution task is *suspended* (it is resumable with a `taskResume`).

Since this activity is not associated to a request, the report is stored in the cIDS as well as in the control poster. Clients can read the poster to know the status of this activity.

5.9 Parallel activities and synchronization

Execution requests can only be declared *compatible* or *incompatible* with each other. In the first case, their execution becomes completely independent one another. In the second case, they interrupt themselves. There are some intermediate cases, where requests must synchronize, or exchange data. Those cases are to be handled by the codels.

To do so, it is possible to use *activity ids*: each eactivity is identified be a number between 0 and `MAX_ACTIVITIES`−1. From a codel, the current activity number is returned by the macro `CURRENT_ACTIVITY_NUM`.

This id can be used to exchange information between activities. For instance, it would be possible to declare a global (static) array, of size `MAX_ACTIVITIES`, in which each ele-

ment would contain information regarding each activity (current state, order of arrival of the request, number of the previous and next activity, ...).

Consider the following example, where you wish to *concatenate* several motion requests for a mobile. The motion request must be compatible with itself (because it must not interrupt the latest motion request) *but* the execution codel `EXEC` must not start before the previous request has completed. This must be handled internally, and the transition `START` → `EXEC` of a new activity must be synchronized with the transition `EXEC` → `END` of the activity.

Such a synchronization can be achieved in the codel `c_exec_func_start`. This codel can register new activities in a global array, attach to them the previous activity, and stay in the `START` state until the previous activity stops. The latter information will be registered by the codel `c_exec_func_end`.

The following code proposes an example of such start and end codels, along with the global data definition:

```
/* Global array for "Motion" requests */
struct DEMO_MOTION_STR {
    ACTIVITY_EVENT state;
    int next;
} demoMotionTab[MAX_ACTIVITIES] = {ETHER, -1};

/* Latest "Motion" request sent */
static int demoMotionLast = -1;
```

```
/* Start codel c_exec_func_start of the "Motion" activity */
ACTIVITY_EVENT
demoMotionStart(MOTION_STR *params, int *bilan)
{
    int current = CURRENT_ACTIVITY_NUM(DEMO_MOTIONTASK_NUM);

    /* If that is a new activity */
    if (demoMotionTab[current].state == ETHER) {

        /* If there is an active activity: wait */
        if (demoMotionLast != -1) {
            demoMotionTab[current].state = START;
            demoMotionTab[demoMotionLast].next = current;
        }
        /* No activity: one can start immediately */
        else {
            demoMotionTab[current].state = EXEC;
        }

        /* Append ourselves to the end of the list */
        demoMotionLast = current;
    }
    return (demoMotionTab[current].state);
}
```

```
/* Termination codel c_exec_func_end of the "Motion" activity */
ACTIVITY_EVENT
demoMotionEnd(MOTION_STR *params, int *bilan)
{
    int current = CURRENT_ACTIVITY_NUM(DEMO_MOTIONTASK_NUM);
    int next;

    /* Next activity number */
    next = demoMotionTab[current].next;

    if (next == -1)
       /* If there is no next activity */
        demoMotionLast = -1;
    else
       /* Unblock next activity */
        demoMotionTab[next].state = EXEC;

    /* This activity is terminated */
    demoMotionTab[current].next = -1;
    demoMotionTab[current].state = ETHER;
    return ETHER;
}
```

Warning: if a synchronized activity fails (either because it is interrupted or because of a problem), it must signal it to other pending activities in order to also cancel them. It will also be necessary to set up a way to re-synchronize with clients, for instance with a control request.

5.10 Coding advice

5.10.1 General coding rules

A module is designed to be integrated in a complex system: users and maintainers are usually not the same people. For this reason, it is very important to respect a few coding rules.

- Split programs into functions and files of a reasonable size;
- Prototype functions;
- Comment your code while you are writing it:
  - A comment for each function which documents the purpose and the limitations you are aware of.
  - A comment for an average of 3 or 4 lines of code.
- Avoid global variables;
- Avoid magic numbers (use constants and `#define`).

- Choose a uniform style, and follow it. For instance, VxWorks uses all uppercased words, separated by underscores for constants (e.g. `DEMO_SPEED`) and lowercase words, with a first uppercase letter for each word but the first (e.g. `controlSpeed`) for symbols;
- Prefix all exported symbols (types, constants, functions, ...) with e.g. the module name;
- Use explicit names. Avoid short names such as `i`, `nb`, `num`.
- Check validity of input parameters and return a report in case of an error.

5.10.2 Case of embedded real-time systems

Modules are likely to be embedded on a distant machine, where they will interact with other modules and processes. This implies a few constraints since a failure of your module can affect the integrity of the whole system.

Memory limitations: Memory is usually limited on embedded systems: there is not so often a virtual memory system. You must thus avoid big data structures, and *free* as much as possible unused memory. This can be done thanks to the `END` and `INTER` states of the codels.

Memory sharing: Some systems (e.g. VxWorks) do not have private address space for processes. Global data is shared among every processes which runs on the same CPU. You must thus *discriminate* as much as possible global names. In such system, there is nothing that will prevent global variables with the same name to interfere!

Furthermore, there are systems which do not provide memory access checks. It is possible to read or write in the whole memory, even in the system memory. Array read or writes beyond bounds will lead to unpredictable results... It is very advised to process to memory checks with adequate tools such as `workShop` or `purify`.

Temporal constrains: For activities that do have hard temporal constrains,

- Give a high priority to the task,
- Avoid displays such as `printf()`, which can be very time consuming.
- Avoid dynamic memory allocations, expensive and not necesseraly bounded in time. Modules generated by G$^{en}_{o}$M do *no* dynamic memory allocation: they execute in constant time. Moreover, memory allocation can always fail, and thus block an execution. It is safer to do all allocation (static or dynamic) upon module startup.

To check that your activities do no last too much, you can display precise statistics for codels. See chapter 6 in this document.

Error recovery: As opposed to a simulated system, you cannot just display an error message and exit when you encounter an abnormal situation. The message will usually be lost (or not seen) and the whole system can be in danger (with potential dangerous situations for the machine).

You must thus:

1. Think of every possible problem (invalid parameters, case not handled by the function, insufficient memory, ...) and define reports for every such situation.
2. Detect failures: check the parameters, check the results of functions.
3. Always keep a sane state inside your functions: free memory, ...
4. Signal every problem with an appropriate error code
5. And if you display something, do not forget to precise the name of the module and the function implied.

Chapter 6

Using modules

This chapter presents some means of using the services provided by a module and of addressing data in the posters.

6.1 The interactive test program `Essay`

The interactive test program is a client of a module. One can launch several instances of it, provided they are given different *numbers* (see chapter 1).

This program proposes a menu, which associates a number to each command: You just have to enter the number corresponding to the command you want to execute. Pressing the `return` key without giving any number invokes the last command.

6.1.1 Sending a request

The N requests of the module are numbered from 0 to $N-1$. If a request has some input parameters, they must be entered interactively.

The bracketed value is a default value: simply pressing `return` will select it. To interrupt the interactive input, type "`.`" (a single dot): defaults values will be affected to the remaining parameters. Default values are initialized to 0. Then they contain the last entered value.

Once you have entered the input data, you must confirm the execution. Type "`a`" to abort. For an execution request, you must choose between the blocking mode or the non-blocking mode. In the first case, the execution of the interactive program will be kept blocking until the final reply of the request. In the second case, the request is just sent and you will be able to see its final replies later, by yourself (with the command 77, see below).

6.1.2 Other commands

Six other commands are defined:

55: posters *Display posters.* This command displays another menu, which lets view either a whole poster or a poster's sub-structure.

66: abort *Interrupt an activity.* This command displays the list of running activities, and waits for the number of the activity you wish to interrupt. Just type enter to leave this menu.

Note: if there is no running activity, you can stop the module by entering `-99`. `-66` will resume suspended tasks.

77: replies *Read the final replies.* You must read pending replies from time to time to empty the mailbox.

88: state *Display the module state, i.e.* the control poster.

99: quit *Terminate the program.* But not the module!

-99: end *Terminate the program AND the module.*

6.2 The interactive tcl shell `tclServ`

`tclServ` is a server which connects to a list of modules on one side, and accepts tcl clients on the other side. Clients can then send requests to a set of modules, using the `tcl` scripting language. This can be done either interactively, or by the mean of scripts.

To be able to use this functionality, you must generate the module with the option `-t`. A separate document is (will be?) available, and describe the usage of this server.

6.3 `Propice` and `transGen`

You must generate the module with the option `-x`. A separate document is (not yet) available.

6.4 Accessing modules' posters from modules

Two different cases must be considered.

- The name of the poster to be accessed is known (*e.g.* the position of a mobile in the module which produces it).
- The poster name is not known *a priori*, and can be dynamically chosen.

6.4.1 The poster name is known

To be able to read such a poster from the codels of a module, the three steps below must be followed:

First step: "poster_client" declaration

Posters names must be declared within the field `poster_client_from` of each execution task which will read those posters (*i.e.* the one that runs the codels that implement those functions).

For instance, the execution task named `MotionTask` can be enabled to read the poster `Mobile` from the module `demo` and the poster `Echoes` from the module `us` by stating:

```
exec_task MotionTask {
     ...
     poster_client_from: demo::demoMobile, us::usEcho;
     ...
};
```

This declaration lets G^en^oM find the necessary libraries and call the adequate initialization functions.

Second step: reading a poster from within its clients' codels

From within the codels, you can call the poster functions of the libraries `usPosterLib` and `demoPosterLib` (in the `auto/` directory of these modules). You just need to include the files `usPosterLib.h` and `demoPosterLib.h` in the codels' file.

The poster library provides read functions (functions `xxxPosterRead`) and display functions (functions `xxxPosterShow`) for the control and execution posters of a module.

In the following example, one first reads the whole poster `demoMobile`, then only a sub-structure `Ref` (see page 43 for the structure definition):

```
#include "demoPosterLib.h"

DEMO_MOBILE_POSTER_STR mobile;
double ref;

demoMobilePosterRead(&mobile);
demoMobileRefPosterRead(&ref);
```

These functions return a `STATUS` (`OK` or `ERROR`). Only the read functions have a parameter, which is the address of the structure in which the read data will be copied.

As shown in the example, the function name is the concatenation of *i.* the name of the module, *ii.* the name of the poster (`Cntrl` for the control poster), *iii.* the sub-structure name (when a subpart of the poster is addressed instead of its whole) and *iv.* the suffix `PosterRead` or `PosterShow`. These functions can be found in the header `demoPosterLibProto.h`.

Third step: compilation

Beware: this part might be subject to important changes

To use libraries correspondig to other modules, you must:

- Define the path of this module in an `Init.make` file. The variable is the name of the module, all in uppercase letters.

  For instance: `DEMO = /usr/local/robots/modules/sara/demo`.

- Add to the `Makefile.unix` and `Makefile.vxworks` a line such as

  `CPPFLAGS += -I${DEMO} -I${DEMO}/auto`.

- Add to the `Makefile.unix` a line like:

  `LIBS += ${DEMO}/auto/${TARGET}/demoClient.a`.

  For VxWorks, you must load the client library `demoPosterLib.o` or `demoClient.o` (which contains the first one).

6.4.2 The poster name is not known

If the name of the poster to read is unknown (*e.g.* if it will be set by a user) you cannot use its library. You must use basic functions of the generic `posterLib` library instead.

When, at run time, you will get the name of the poster, you must first find its id number, which is done thanks to the functions `posterFind`, as shown in the example below:

```
static POSTER_ID distantPosterId;
char *name;
...
if (posterFind(name, &distantPosterId) == ERROR) {
   *report = errnoGet();
}
...
```

We have already seen how to write into posters. The read function works in the same way:

The `posterRead` function has the same prototype as the `posterWrite` function:

```
int posterRead(POSTER_ID posterId, int offset, char *buf, int nbytes)
```

`posterId` is the poster id (returned by `posterFind`), `offset` is the offset in bytes from the beginning of the structure and `nBytes` is the number of bytes to read. The function returns the number of bytes actually read (normally `nBytes`).

As for `posterWrite`, it is also possible to use the address of the poster and write directly into it, thanks to the `posterAddr` function. Such accesses must be protected with a pair of `posterTake` and `posterGive` (`posterTake` must be called with the flag `POSTER_READ` instead of `POSTER_WRITE`).

```
double speed;

posterTake(posterId, POSTER_READ);
speed = addrPosterMotion->state.speed;
posterGive(posterId);
```

6.5 Accessing modules services from modules

It is possible to send requests to module from any process, and in particular from other modules (*i.e.* from their codels). As for posters, you must follow three steps:

6.5.1 First step: declaring modules

The modules to which you want to send requests are to be declared in the execution task field `cs_client_from`. For instance, if you wish to use the services of the modules `demo` and `us`:

```
exec_task MotionTask {
     ...
     cs_client_from: demo, us;
     ...
};
```

This step lets GenoM find the necessary communication libraries and automatically call the initialization functions (create a mailbox, and get a client id).

6.5.2 Second step: sending requests and receiving replies

You can send requests from codels through the functions of the two libraries `usMsgLib` and `demoMsgLib` (in the case of our example). To do so, you need to include the headers `usMsgLib.h` and `demoMsgLib.h`.

The library `demoMsgLib` defines several functions whose names are concatenation of: *i.* the name of the module, *ii.* the name of a request (`Abort` for the interrupt request), *iii.* a suffix showing its purpose. Four suffixes are available:

suffix	function
`RqstSend`	send a request (non blocking)
`ReplyRcv`	receive a reply (final or intermediate) (blocking or not)
`RqstAndRcv`	send a request *and* receive the *final reply* (blocking)
`RqstAndAck`	send a request *and* receive the *intermediate reply* (blocking)

For a control request, you can use the function `RqstAndRcv` even though it is blocking: indeed, control requests are meant to execute in a very short time, so that the final reply should quickly occur.

However, for an execution request it is strongly advised to use the function `RqstAndAck`, which waits only for the intermediate reply (acknowledgment of the reception of the request). In general, you cannot block your module until the completion of the remote request. The final reply will be read with the non-blocking function `ReplyRcv`, which you will have to call until reception of the reply.

Consider this example:

- To send the control request `SetSpeed` to the `demo` module, you can use the function `demoSetSpeedRqstAndRcv`.
- To send the execution request `Monitor`, you can use:
  - the function `demoMonitorRqstAndAck` and then
  - the function `demoMonitorReplyRcv` in non-blocking mode, until the reply comes. (if there is nothing else to do, the `SLEEP` state is particularly well suited).

The prototypes of these functions are defined in the header `auto/demoMsgLibProto.h`. Generic functions (as for posters) also exists, and are documented in the sections below).

6.5.3 Third step: compilation

To compile and link under Unix, you can use the same procedure as for posters. For VxWorks, the library to load is `demoMsgLib.o`, or `demoClient.o` which includes it.

6.5.4 An example

We present here a few examples, which involve a module `pilo` that must send the `SetSpeed` and `MoveDistance` requests to the module `demo`. These requests are sent by the execution task `CmdTask`. The `cs_client_from: demo` field has been declared in this task.

Sending a control request: `RqstAndRcv`

The functions `RqstAndRcv` can have 2, 3 or 4 arguments, depending on the input and output declarations of the request:

```
int ...RqstAndRcv(CLIENT_ID clientId,
                  [STR_IN *in,] [STR_OUT *out,] int *report);
```

- The first argument is the client number. For instance: `PILO_CMDTASK_DEMO_CLIENT_ID` for the module `pilo`, client of the module `demo`, in the execution task `CmdTask` (see header `piloHeader.h`.
- The second and third arguments (between square brackets) are optional, and defined only if the request defines an input or an output parameter.
- The last argument is the report, returned by the request.

This function returns `FINAL_REPLY_OK` if everything went well, or `ERROR` if not.

The following example sends the `SetSpeed` request from the codel `piloSendRequest` of the module `pilo`:

```
ACTIVITY_EVENT
piloSendRequest(DEMO_SPEED *speed, int *report)
{
   if (demoSetSpeedRqstAndRcv(PILO_CMDTASK_DEMO_CLIENT_ID,
                              speed, report) != FINAL_REPLY_OK) {
      return FAIL;
   }
   return EXEC;
}
```

Sending an execution request: `RqstAndAck`

```
int ...RqstAndAck(CLIENT_ID clientId,
                  int *rqstId, int replyTimeOut,
                  [STR_IN *in,] [STR_OUT *out],
                  int *activity, int *report);
```

In comparison with the `...RqstAndRcv` functions, the `RqstAndAck` functions have three more arguments:

- `rqstId` is filled with the request id. This id will let you read the reply later.
- `replyTimeOut` is the time (in ticks) for which you want to wait for the final reply. The value 0 means "wait forever".
- `activity` is the activity number. This number will let you get information on it (state) or abort it.

Note: if the execution is very fast, you can get the final reply immediately. This is why this function also has the `out` parameter.

This function returns `WAITING_FINAL_REPLY` if the intermediate reply has been received, or `FINAL_REPLY_OK` if the final reply has already been received. `ERROR` is returned in case of a problem.

The following example shows the sending of the `Monitor` request from `piloSendRequest2` in the module `pilo`:

```
/* Global variables */
static int piloDemoMonitoRqstId = -1;     /* Number of the request */
static int demoMonitorActivity;           /* Number of the activity */
static double piloDemoMonitorOut;         /* Output parameter */

ACTIVITY_EVENT
piloSendRequest2(double *posMon, int *report)
{
   switch (demoMonitorRqstAndAck(PILO_CMDTASK_DEMO_CLIENT_ID,
                                 &demoMonitorRqstId,  0,
                                 *posMon,  &piloDemoMonitorOut,
                                 &demoMonitorActivity,  report)) {
      case WAITING_FINAL_REPLY:
         return SLEEP;
      case FINAL_REPLY_OK:
         piloDemoMonitorRqstId = -1;
         return END;
      default:
         return ZOMBIE;
  }  /* switch */
}
```

Receiving replies: `ReplyRcv`

```
int ...ReplyRcv(CLIENT\_ID clientId,
                int rqstId, int block,
                [OUT *out], int *activity, int *bilan);
```

This function takes two new arguments:

- `rqstId`, which is the identification number returned when the request was sent,
- `block`, which tells if we want to block until the reply arrives or not.

The following example shows the reception of the reply of the `Monitor` request from the codel `waitReply`. We use the static variables defined in the previous example.

```
ACTIVITY_EVENT
waitReply(double *posMon, int *report)
{
   switch (demoMonitorReplyRcv(PILO_CMDTASK_DEMO_CLIENT_ID,
                               demoMonitorRqstId, NO_BLOCK,
                               &piloDemoMonitorOut, &demoMonitorActivity,
                               &report)) {
      case WAITING_FINAL_REPLY:
         return SLEEP;
      case FINAL_REPLY_OK:
         piloDemoMonitorRqstId = -1;
         return END;
      default:
         return ZOMBIE;
  }  /* switch */
}
```

Important note: The activity which sends the request and reads its reply can be interrupted. In that case, the codel which reads the final reply might never be executed. It is not advised to let such an activity pending, without reading its reply. To avoid this, you must define an interrupt codel, in which you will interrupt the activity started by the initial sending of your request. This can be done tanks to the `Abort` request:

```
ACTIVITY_EVENT
interActi(double *posMon, int *bilan)
{
   /* No current activity */
   if (piloDemoMonitorRqstId == -1)
         return ETHER;

   /* interrupt the Monitor activity */
   if (demoAbortRqstAndRcv(PILO_CMDTASK_DEMO_CLIENT_ID,
                           &demoMonitorActivity, bilan) != FINAL_REPLY_OK)
         return ZOMBIE;

   /* Wait for the final reply (which should arrive shortly thanks to the
      interruption */
   if (demoMonitorReplyRcv(PILO_CMDTASK_DEMO_CLIENT_ID,
                           demoMonitorRqstId, BLOCK_ON_FINAL_REPLY,
                           &piloDemoMonitorOut, &demoMonitorActivity,
                           &bilan) != FINAL_REPLY_OK)
        return ZOMBIE;

   piloDemoMonitorRqstId = -1;
   return ETHER;
}
```

6.6 Accessing modules services from another process

6.6.1 The library `posterLib`

The library `demoPosterLib` provides an initialization function `demoPosterInit` and a set of read functions (ending with `PosterRead`) and display functions (ending in `PosterShow`) for the control and execution posters of modules.

6.6.2 The library `msgLib`

Before you can use this library, you have to create a mailbox in order to receive the replies of remote servers. This is done with the function `csMboxInit`.

Then, you must initialize connections for individual clients: for instance `demoClientInit` in the libray `demoMsgLib`.

Before you quit, you must free this connection with `demoClientEnd` and close the mailbox with `csMboxEnd`.

Appendix A

Troubleshooting

A.1 Module generation

Avoid conflicts with G^{en}oM keywords: in the `.gen` file, do not use variables nor functions named `type`, `control`, `poster`, ...

Avoid conflicts with structure names which are generated by G^{en}oM: `DEMO_STR`, ...

G^{en}oM should parse every valid `C` file. As of today, a few problems remain, especially for `union`s and recursive typedefs such as `typedef PILO_MOVE_STR PILO_MOVE_STR_2[2]`. You should not use them at this time.

Problems may also arise if you use different names for a structure and a typedef associated to that structure, as in the following example:

```
/* XXX avoid that at this time */
typedef struct DIST_STR {
   double dist;
} dist_str;
```

A.2 Execution under Unix

A.2.1 `csLib` initialization failures

```
rantanplan% h2 init
Initializing csLib devices:
Cslib devices already exist on this machine.
Do you want to delete and recreate them (y/n) ?
```

→ `csLib` is already initialized. You can answer `n` if everything is ok and you don't need to initialize `csLib` again. Answer `y` if you need to reset `csLib`.

A.2.2 Module startup failures

```
rantanplan% ./codels/i386-linux/demo -b
Hilare2 execution environment version 1.0
Copyright (c) 1999 LAAS/CNRS
demo: error creating /home/matthieu/.demo.pid: File exists
```

→ You didn't kill properly an old instance of the module. Use the command "`killmodule <module-name>`".

```
waits[demo] ./codels/sparc-solaris/demo
DEMO :
Spawn control task ... demoCntrlTask/posterCreate:
                                              S_h2devLib_DUPLICATE_DEVICE_NAME
```

→ The control poster already exists: an old module was not killed properly, see above.

A.2.3 Interactive "Essay" program failures

The shell blocks after sending a request: launch another task with another number.

A.3 VxWorks

A.3.1 Loading failures

```
undefined symbol: rdima
```

→ The dynamic link edition failed: the symbol `rdima` (a function, or a variable) is undefined. You must load the library that defines it before the library that uses it.

A.3.2 Module startup failures

A.3.3 Interactive "Essay" program failures

The terminal window does not show up.

- You didn't launch `xes_server` on the remote station.
- The `DISPLAY` variable is unset, or is not properly set
- You do not authorize the remote machine to display on your display server.
- You did not define under VxWorks which Unix machine will host the display: use `xes_set_host <host>` and `xes_get_host` to check.

A.3.4 It was working, but...

Besides problems specific to your codels, some common problems can be the cause of sudden failures:

A stack is too small

You can easily detect this with the command `checkStack`. This command displays the stack usage for all tasks. If a stack size is labeled `OVERFLOW`, the stack is too small. If the corresponding task is an execution task of a module, you just have to increase its size, recompile and run the module again (`stack_size` field). If the task is a control task, you have to turn the control codel into an execution codel (and increase the stack size of the consequent execution task).

Local variables in your functions are stored on the stack. Avoid big arrays and other huge structures. Be careful with the `stdio` functions such as `fopen` which are great stack consumers!

Not enough memory

You can detect this with the command `memShow`. Check that you do not use huge data structures and that you free memory as often as possible.

Common bugs in codels

Writes beyond arrays' bounds and writes in memory pages that do not belong to you are very common mistakes. Use common Unix tools such as `purify` or `workshop` to debug.

Appendix B

Communication libraries

Communication between modules is based on two libraries: `posterLib` for posters and `csLib` for requests. This appendix presents a short overview of the functions of composing these libraries.

B.1 Posters and `posterLib`

Posters are shared memory segments, that can be written by their owners and read by any process. The basic poster handling functions and their prototypes are:

name	description
`posterCreate`	poster creation
`posterFind`	look for a poster id given its name
`posterWrite`	write into a poster
`posterRead`	read a poster
`posterAddr`	get the poster address
`posterTake`	take the poster semaphore
`posterGive`	give the poster semaphore
`posterIoctl`	query information about the poster (date, ...)
`posterDelete`	delete a poster
`posterShow`	display the state of every poster
`posterMemCreate`	create a poster at the given address

```
STATUS   posterCreate      (char *name, int size, POSTER_ID *id);
STATUS   posterFind        (char *name, POSTER_ID *id);
int      posterWrite       (POSTER_ID id, int offset, void *buf, int size);
int      posterRead        (POSTER_ID id, int offset, void *buf, int size);
void *   posterAddr        (POSTER_ID id);
STATUS   posterTake        (POSTER_ID id, POSTER_WRITE);
         or                (POSTER_ID id, POSTER_READ);
STATUS   posterGive        (POSTER_ID id);
STATUS   posterIoctl       (POSTER_ID id, int code, char *parg);
STATUS   posterDelete      (POSTER_ID id);
STATUS   posterShow        (void);
STATUS   posterMemCreate   (char *name, int busSpace, void *pPool, int size,
                           POSTER_ID *id);
                           (busSpace: POSTER_LOCAL_MEM, POSTER_SM_MEM,
                           POSTER_VME_A32, POSTER_VME_A24)
```

B.2 Requests and `csLib`

Communication between modules is based on the client/server library `csLib`. Messages (requests and replies) are held in mailboxes (ring buffers).

`csLib` has the following properties:

- Both the sending of requests and the reception of replies can be done in non-blocking mode.
- The reception of a request is associated to the execution of a `C` function.
- A request generates at most two replies.
- There is no dynamic memory allocation.
- It runs under Unix and VxWorks.

The client side provides the following functions:

name	description
`csClientInit`	
`csClientRqstSend`	Send a request
`csClientReplyRcv`	Receive a reply

B.3 Execution

VxWorks

There is normally nothing to do

Unix

Launch `h2 init`.

To access remote posters (on another machine), define the `POSTER_HOST` environment variable with the name of the *server* machine (there can be only one, *i.e.* all the posters must be hosted by the same machine).

The poster server `posterServ` is launched by `h2 init`, except if `POSTER_HOST` is defined.

Between Unix and VxWorks

This is not possible anymore with this version of G$^{\text{en}}_{\text{o}}$M.

Appendix C

The files generated by G^{en}oM

This appendix presents a description of the files generated by G^{en}oM (source files and compiled libraries).

C.1 Source files in `auto/`

Server side:	
`demoCntrlTask.c`	control task
`demoMotionTask.c`	execution task `MotionTask`
`demoHeader.h`	constants and macros for IDSs (*must included in the codels*).
`demoType.h`	C structures of the module (included by `demoHeader.h`).
`demoModuleInit.c`	module startup.
`demoInit.c`	initialization request.

Client side:	
`demoMsgLib.{c,h,Proto.h}`	Requests libraries.
`demoConnectLib.c`	Smaller version of the library `demoMsgLib.c`: use this if you use `csLib` directly.
`demoPosterLib.{c,h,Proto.h}`	Posters libraries.
`demoError.h`	Error codes (reports).

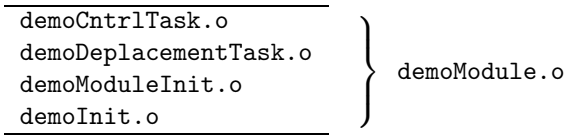
Test program:	
`demoEssay.c`	interactive test program
`demoScan.{c,h}`	interactive scan of input structures
`demoPrint.{c,h}`	display the C structures of the module

Tcl :	
`demoClient.tcl`	tcl scripts for request definition
`demoTcl.c`	`tclServ` server functions

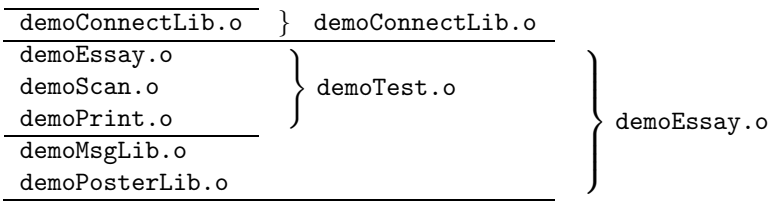
Propice:	
`propice/`	

C.2 Binary files in `${TARGET}`

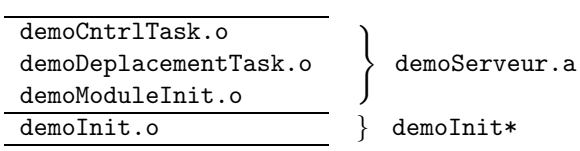
- Libraries for VxWorks (beside individual `.o` object files):
  - Server side:



  - Client side:



- Libraries for Unix:
  - Server side:



  - Client side: