# GenoM

## User's Guide

Sara Fleury, Matthieu Herrb, Anthony Mallet
CNRS ; LAAS
7, Av. du Colonel Roche
F-31077 Toulouse, France

Université de Toulouse; UPS; INSA; INP; ISAE; LAAS; F-31077 Toulouse France

{sara.fleury,matthieu.herrb,anthony.mallet}@laas.fr

September 2013

# 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

# Installation and configuration

## 1.1 Quick start using robotpkg

The robotpkg[1] tool is the quickest way to install GenoM and its dependencies.

### 1.1.1 Setting up robotpkg in five minutes

After downloading the sources, you need to choose where the packages will be installed and bootstrap your installation. For that, you need the GNU make software (version 3.81 or later required), as well as a working C compiler chain.

By default, robotpkg uses the `/opt/openrobots` installation prefix. If this location is convenient for you, installing the bootstrap kit should be as simple as:

```
% cd robotpkg/bootstrap
% ./bootstrap
```

Notes:

- The bootstrap script will try to create the initial /opt/openrobots directory using your user id. You are usually not allowed to do so, so you should create the /opt/openrobots manually before calling bootstrap (using sudo or a similar command). You can also install to a different location as explained below.

- If you need to install to a different prefix, pass the –prefix option to bootstrap. For instance, installing the packages in your home directory can be configured like this:

  ```
  % ./bootstrap --prefix=${HOME}/openrobots
  ```

Make sure to read carefuly the instructions printed at the end of the bootstrap for configuring your shell environment.

[1] http://robotpkg.openrobots.org/

### 1.1.2 The five minutes guide to installing Genom

Once bootstraping is done, installing a package can be done by changing to the directory (within the robotpkg hierarchy) of the package to be installed and doing make update. For GenoM, this means:

```
% cd robotpkg/architecture/genom
% make update
```

Notes:

- The make program must be GNUmake utility, a.k.a gmake. Version >= 3.81 is required.

A more extensive and largely incomplete (!) documentation is also available online[2], or in the `doc/robotpkg` directory of robotpkg.

If you successfully installed GenoM with robotpkg, you can skip to the next chapter. Otherwise read more detailled instructions below.

## 1.2 Requirements

### 1.2.1 Operating systems

GenoM is known to work on most current Unix-like operating systems: Linux (many distributions including Debian, Fedora, Gentoo and Ubuntu have been tested), NetBSD, Mac OS X, Solaris.

GenoM does not work on Microsoft Windows operating system, nor on mobile phones / tablets operating systems like Apple's iOS or Google's Android.

### 1.2.2 External tools

GenoM needs the following tools. They are generally available on most systems. If not, download them from their web sites.

- `autoconf` version 2.59 or later
- `automake` version 1.8 or later
- `GNU make` version 3.79 or later (`<http://www.gnu.org/software/make/make.html>`)
- `pkgconfig` version 0.15 or later (usually part of Gnome development packages).
- `groff` 1.10 or later (usually part of system packages).
- `Tcl/Tk` 8.0 or later development files (for eltclsh) (`<http://tcl.tk/>`)

To edit GenoM modules it is also recommended to use `xemacs` to take advantage of `genom-mode`.

[2] http://softs.laas.fr/openrobots/robotpkg/README.html

## 1.3 What to download ?

### 1.3.1 OpenRobots tools and libraries

GenoM is part of a suite of open-sources tools.

In order to install and use GenoM you will have to download and install the following libraries and tools from `http://softs.laas.fr/openrobots/` :

- `mkdep`: LAAS tool to determine dependencies
- `pocolibs`: system communication and real-time primitives
- `libedit` (editline): generic line editing and history functions (for eltclsh)
- `eltclsh`: interactive TCL shell linked with editline (optional but very practical)
- `GenoM`: the generator of modules

Later on you can also be interested in the following opensource softwares that are part of the OpenRobots suite and work nicely with GenoM:

- `GDHE`: a tool to design 3D interface
- `OpenPRS`: a tool to design complexe supervisors

#### mkdep

`Mkdep` is a tool to manage dependencies for make(1) automatically on Unix-like systems. The original feature of this version is to be able to handle incremental updates of the dependencies.

It is recommended to install it.

#### pocolibs

`Pocolib` is a system communication and real-time primitive layers used by GenoM modules.

These libraries run on systems with POSIX.1 threads and basic real-time extensions. It has been tested successfully on Solaris (7 and above), Linux (with glibc2), and NetBSD (2.0 and later).

There is also some code to support RTAI and LXRT 3.0, but it is currently untested and unsupported (this may change again in the future)

#### tclserv

Starting with G$^{en}_{o}$M version 2.7, `tclserv` is provided as an external package. It will need to be installed separately to be able to use it as described in section 7.2.

#### libedit / editline

The `editline` library provides generic line editing and history functions, similar to those found in tcsh(1). This package contains a Makefile for compiling the current version of NetBSD's library which provides the same consistent installation and compilation environments as for the other tools found in this repository.

Editline is used by eltclsh.

**eltclsh**

## 1.4 Configuration

You will need to specify a root path (for instance the environment variable `OPENROBOTS`) where the binaries, the libraries, the include files, and so on will be installed. Typically you can set this environment variable to `/usr/local/openrobots`.

Then modify (or set if not previously defined) the following environment variables :
`PATH` : add `$OPENROBOTS/bin`
`PKG_CONFIG_PATH` : add `$OPENROBOTS/lib/pkgconfig`
`LD_LIBRARY_PATH` : add `$OPENROBOTS/lib:$OPENROBOTS/lib/openprs`

## 1.5 Installation

Most of the time, it is a simple sequence of untar, autogen, configure, build, and install. Configure options may vary, and some packages don't require the autogen step. It is recommended that you build in a separate directory.

The `-prefix` option (default: /usr/local) defines the base directory of the installation. We recommend a dedicated path (eg, OPENROBOTS=/usr/local/openrobots).

If you want to install binaries for different type of machine on the same files tree you can also use `-exec-prefix` option. It allows to specify where to install the binaries like for instance: `-exec-prefix=$OPENROBOTS/$MACHTYPE-$OSTYPE` (eg, `-exec-prefix=$OPENROBOTS/i386-linux`, for a PC running linux). In such a case, be careful to adapt the environment variables `PATH, LD_LIBRARY_PATH and PKG_CONFIG_PATH` defined above to this `$exec_prefix` path. However, the default value (equal to `-prefix`) is usualy great.

Here is an example with `pocolibs`.

```
tar xfvz pocolibs-XYZ.tar.gz
cd pocolibs-XYZ
[./autogen.sh]
mkdir build
cd build
../configure --prefix=\$OPENROBOTS  %$
make
make install
```

The command `./autogen.sh` is not always necessary. If there is already a `configure` script you can try it directly.

On some systems, `-with-tcl=DIR` and `-with-tk=DIR` will be required to specify where to find `tclConfig.sh` (resp. `tkConfig.h`).

The `configure` command has many options. You can see them with the option `-help`.

# Chapter 2

# A first example

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.

## 2.1 Principle of the module generation

One can distinguish two main parts in a module:

- A *server*, which encapsulates 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.

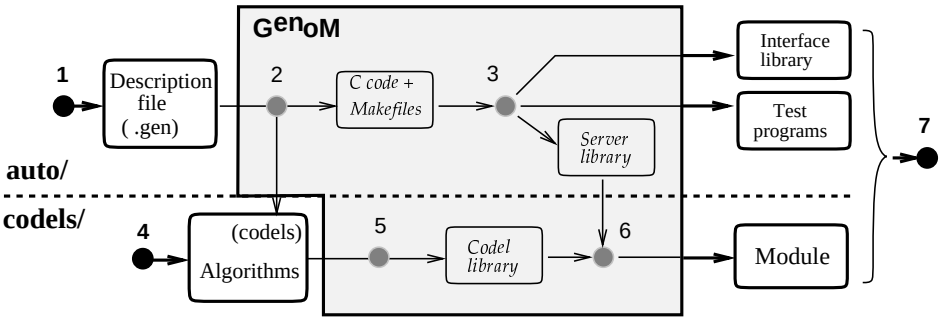


Figure 2.1: Development cycle and separation between server and codels.

To help you distinguish between the server and the codels, the corresponding files are placed in two separate directories. The first one is called `server/` 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 2.1 shows a synthetic view of the separation between `server/` and `codels/` and also represent a typical development cycle :

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

## 2.2 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) */
   version:           "0.1";
   email:             "openrobots@laas.fr";
};


/* ------------- 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 */
   codel_control:       demoSetSpeedCntrl;    /* codel for validity checks */
   fail_reports:        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 */
   interrupt_activity:  Goto;                 /* request to interrupt */
};
```

continued on next page...

... continuation of previous page

```
/* Execution request: translate of a given distance */
request Goto {
   type:                exec;              /* request's type */
   input:               distance::distRef; /* input: distance */
   codel_control:       demoGotoCntrl;     /* codel for validity checks */
   fail_reports:        TOO_FAR_AWAY;      /* possible error messages */
   codel_start:         demoGotoStart;     /* initialization codel */
   codel_main:          demoGotoMain;      /* main codel */
   codel_end:           demoGotoEnd;       /* termination codel */
   codel_inter:         demoGotoInter;     /* interruption codel */
   interrupt_activity:  Goto;              /* 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. */
   codel_main:          demoMonitorExec;       /* main codel */
   fail_reports:        TOO_FAR_AWAY;          /* possible error messages */
   interrupt_activity:  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 4):

| | | |
|---|---|---|
| 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 */
```

## 2.3 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:

```
blues% genom  -i demo
genom demo.gen: info: array MonitorInput added in SDI for request Monitor
genom demo.gen: info: array MonitorOutput added in SDI for request Monitor
perl -w ./demo.pl

Updating top directory
  creating acinclude.user.m4
  creating local.mk.in
  creating configure.ac.user
  creating autogen
  creating Makefile.in

Updating codels
  demoMotionTaskCodels.c changed, skipping
  demoCntrlTaskCodels.c changed, skipping
  Makefile.in changed, skipping

Updating autoconf
  creating genom.mk
  creating install-sh
  creating mkinstalldirs
  creating config.sub
  creating config.guess
  creating ltmain.sh
  creating robots.m4
  creating libtool.m4
  creating config.posix.mk
  creating config.rtai.mk

Updating server
  creating Makefile.in
  creating demoType.h
  creating demoError.h
  creating demoError.c
  creating demoMsgLib.c
  creating demoConnectLib.c
  creating demoMsgLib.h
...
Creating build environment ...
  * Running aclocal
  * Running autoconf


If you already have a build of this module, do not forget to
reconfigure (for instance by running ./config.status --recheck)

Done.
```

Some comments on this run:

1. The `.gen` extension needs not to be explicitely given. G$^{en}$oM appends it automatically.

2. 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 D).

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

```
blues% ls -F
CVS/               autoconf/  configure*         demoConst.h   server/
Makefile.in        autogen*   configure.ac.user  demoStruct.h  xaff/
acinclude.user.m4  codels/    demo.gen           local.mk.in
```

GenoM created two new directories `server/` and `codels/`, and several new files `configure`, `Makefile.in` and several other autoconf related files.

- The `server/` directory is entirely dedicated to GenoM (and that is the only such directory). It contains all the *server* code (see figure 2.1), 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 GenoM 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 GenoM never goes into it again (except if you regenerate the module with the `-i` option).
- The `Makefile.in` and `configure.ac.user` files are also under your control. These are the main files which are used for the compilation of the module.

## 2.4 Module compilation

There are two steps in compilation of the module:

- *configuration* of the module, which is done using the GNU autotools framework, by running the `configure` script.
- *compilation* itself, controlled by the `Makefile` generated by the previous step.

### 2.4.1 Build directory

In order to keep objects files separated from the sources, for instance when you want to generate the module for several different machine architectures, or just in order to have a simple mean to clean up everything that was produced during the building phase, it is strongly recommended to create a separate *build* directory and run every command from there.

```
blues% mkdir obj
blues% cd obj
```

### 2.4.2 Configuration

All the OpenRobots software and tools are generally installed in a specific directory (for instance `${HOME}/openrobots` or `/usr/local/openrobots`. This is the main information that needs to be specifed to the `configure` script. Don't forget to run `configure` from the build directory.

```
blues% cd obj
blues% ../configure --prefix=${HOME}/openrobots
checking build system type... i686-pc-linux-gnu
checking host system type... i686-pc-linux-gnu
checking for gcc... gcc
checking for C compiler default output file name... a.out
checking whether the C compiler works... yes
checking whether we are cross compiling... no
checking for suffix of executables...
checking for suffix of object files... o
...
configure: creating ./config.status
config.status: creating config.mk
config.status: creating Makefile
config.status: creating codels/Makefile
config.status: creating server/Makefile
config.status: creating demo.pc
config.status: creating local.mk
```

### 2.4.3 Compilation itself

To compile the module, just run `make` from the build directory. The GNU make utility is required, but this is the standard make on Linux systems. On different Unixes, it may be installed under a different name, usually `gmake` or `gnumake`.

```
blues% make
make all-posix
make[1]: Entering directory `/home/matthieu/openrobots/modules/demo/obj'
make[2]: Entering directory `/home/matthieu/openrobots/modules/demo/obj/server'
mkdir -p posix-obj
/home/matthieu/openrobots/i386-linux/bin/mkdep -c"gcc" -oposix-obj/dependencies -dposix-obj -t.l
make dependencies for ../../server/demoCntrlTask.c...
make dependencies for ../../server/demoModuleInit.c...
make dependencies for ../../server/demoMotionTask.c...
...
creating posix-obj/demo
make[2]: Leaving directory `/home/matthieu/openrobots/modules/demo/obj/codels'
make[1]: Leaving directory `/home/matthieu/openrobots/modules/demo/obj'
blues%
```

### 2.4.4 Installation

The built binaries and libraries (and some associated files) need to be copied to their final locations. This is achieved by executing `make install`. Depending on your environment, this may require *root* priviledges.

```
blues% su
password:
blues# make install
blues% make install
make install-posix
make[1]: Entering directory '/home/matthieu/openrobots/modules/demo/build'
make[2]: Entering directory '/home/matthieu/openrobots/modules/demo/build/server'
../../autoconf/mkinstalldirs /usr/local/openrobots/lib
...
ranlib /usr/local/openrobots/lib/libdemoClient.a
chmod 644 /usr/local/openrobots/lib/libdemoClient.a
PATH="$PATH:/sbin" ldconfig -n /usr/local/openrobots/lib
----------------------------------------------------------------------
Libraries have been installed in:
   /usr/local/openrobots/lib
...
../autoconf/mkinstalldirs /usr/local/openrobots/lib/pkgconfig
/usr/bin/install -c demo.pc  /usr/local/openrobots/lib/pkgconfig
blues#
```

## 2.5 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 `bin` subdirectory of the directory specified as prefix in the configuration step. (see figure 2.1). This is an executable whose name is the name of the module (i.e., `demo`).

The next two sections present a step-by-step tutorial on how to run modules.

### 2.5.1 Execution under Unix

**Module startup:**

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

```
blues% h2 init
Initializing pocolibs devices: OK
pocolibs execution environment version 2.11
Copyright (c) 1999-2011 CNRS-LAAS
blues%
```

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

```
blues% h2 init
Initializing pocolibs devices:
pocolibs devices already exist on this machine.
Do you want to delete and recreate them (y/n) ? y
OK
pocolibs execution environment version 2.11
Copyright (c) 1999-2011 CNRS-LAAS
```

2. Launch the module.

```
blues% demo -b
pocolibs execution environment version 2.11
Copyright (c) 1999-2011 CNRS-LAAS
demo: spawning control task.
demoCntrlInitTask: created mailbox
demoCntrlInitTask: initialized mailbox as a server
demoCntrlInitTask: installed requests
demoCntrlInitTask: installed abort request
demoCntrlInitTask: created control poster
demo: spawning task demoMotionTask.
demoMotionTaskInitTaskFunc: timer allocated
demoMotionTaskInitTaskFunc: timer started
demoMotionTaskInitTaskFunc: posters created
demoMotionTaskInitTaskFunc: client posters initialized
demoMotionTaskInitTaskFunc: ok
demo: all tasks are spawned
blues%
```

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 `demoTest`.

**Client startup: the interactive test program `demoTest`**

```
demoTest 1
```

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

**Killing the module:**

```
blues% killmodule demo
```

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

**Cleaning everything:**

```
blues% h2 end
```

# Chapter 3

# Modules description

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

## 3.1 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 will be sent by the operator using various interfaces (`Test` program generated by G$^{en}$oM or `tcl` shell - see paragraph 7.2 on page 60) or by a program, usually named a *supervisor*, that will supervise the robot (for instance using `OpenPRS` - see paragraph 7.3 on page 61).

Each request can have 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 request. Each reply is associated with an *execution report*: it reports on the execution of the service and lets the client know about problems that might have occurred.

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 mainly allow to set parameters or interrupt services. 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 (even indefinitely). Thus they send an *intermediate* reply, as soon as the service starts. The final reply is sent when the service is over.

A running service is called an *activity*. Some functions, such as monitoring services, can start *several* activities and execute them simultaneously. Other kind of function, such as servoing functions, cannot handle parallelism and can only start *one* activity at a time. This constraint has to be indicated by the developer.

Activities can control a physical device (sensors, actuators), read data produced by other modules (by the mean of posters) 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` and `XML` 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, ...) at any time.

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).

## 3.2 Structure and functioning of modules

As shown in the figure 3.1, 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 *control task* that controls the module.
- One (or several) *execution tasks* that run your services and the algorithms they are made of.

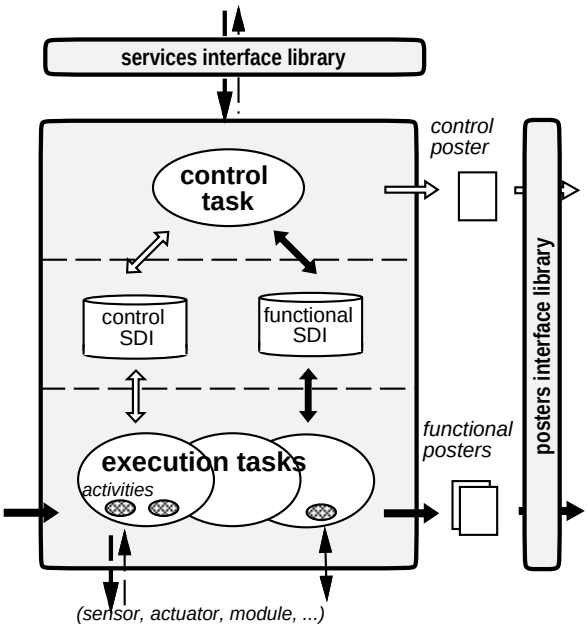


Figure 3.1: Structure of modules.

**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. it receives the requests for the module,
2. it checks the validity of the input parameters and stores them in the fIDS,

3. it checks if the module can start a given service (handles conflicts between requests),
4. it 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 possibly 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 according to the following rule: if incompatible services are to be run at the same time, **the last request has the priority**. Activities that happen to be incompatible with that new 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 (supported under real-time operating systems only), 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 in various programming languages (for now: `C`, `tcl`, `XML`, `OpenPRS`).

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

## 3.3 Integration of the algorithms: the codels

In order to associate your code to the requests, you have to tell G$^{\text{en}}$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$^{\text{en}}$oM has generated and the codels libraries.

# Chapter 4

# 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.

## 4.1 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$^{en}_{o}$M structures (requests, posters, ...). It also includes on-line help.

Note that for now `genom-mode` works only with `xemacs` and not anymore with emacs...

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 structures from another module |
| `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 **b**uffer |
| `C-c C-v` | **v**erify 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$^{en}_{o}$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.

Note that if this function has not been written yet (new codel or even request), then `genom-mode` prompts you to insert the empty template of the codel (or all the codels of the request if you click on the request head-line). For this to work, the the module must be first regenerated once.

## 4.2 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.*

### 4.2.1 Module declaration

A module declaration looks like this:

```
module <<module-name>> {
    number:            <<module-number>>;
    internal_data:     <<SDI-type>>;
    version:           <"version-string">;
    email:             <"address">;
    requires:          <package-dependency> , ...;
    codels_requires:   <package-dependency> , ...;
    lang:              <c or c++>;
};
```

This part is mandatory and lets you choose a name for your module. Fill-in the field `<<module-name>>`.

The parameters are the following:

- `number:` is the identification number of the module. It must be greater that 1000 (small numbers are reserved for system) and smaller than $2^{15} = 32768$. It should be unique and no other module should use the same one. It is used to generate unique error report numbers.

- `internal_data:` is the name of a valid `C` type. This will be the module internal database (IDS). `genom-mode` provides you with a default value for this field.

- `version` is a string enclosed in double-quotes that defines the external version number of your module. This is used by the GNU build tools in various places. You should incremement it before releasing a new version of a module.

- `iface_version:` is similar but it is the `libtool` version of the module, not the required package. The strategy relatively to the module versioning is still not well defined and for now this field is rarely used.

- `email` is the contact address for the module. This will be copied in some files generates by the GNU build tools. Make sure you use an address that will be valid in the future if you plan to release and distribute your module.

- `requires:` allows mainly to list the other /GenoM/ modules that are imply within this `.gen` file (referenced structures). More generally, it allows to declare the list of packages (in the sens of `pkg-config`) require to generate this `.gen` file.

- `codels_requires:` it allows to declare the list of packages (in the sens of `pkg-config`) require to compile and link the codels (and only the codels).

- `lang` sets the programming language used by the codels. It defaults to “C”. “C++” can be used to set the language of the codels to C++.

### 4.2.2 `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 will be defined progressively by adding the requests parameters each time a new request is added: do not try to write it *a priori*.

#### 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 or programs. Thus, it is *strongly* advised you prefix their names with the name of your module to avoid conflict and determine its origin (see the example `demoStruct.h` in the section 2.2).

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$^{en}_{o}$M:

1. **Allowed C types**: G$^{en}_{o}$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 must be protected 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.

It is strongly recommended that you do not hard-code the path to the external headers.

The field `requires:` within the module declaration structure, will allow GenoM to automatically find out the corresponding header and library files using `pkg-config`.

It is also possibly to specify path by hand using GenoM options `-I<path>`, `-J<VAR>=<path>` or `-P<package>` (see chapter 5 on page 37).

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

```
module pilo {
     ...
     requires:    loco;
};

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

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

#### 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 (but only within the codels file).

### 4.2.3 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 2.

### 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>;
     codel_control:          <codel-name>;
     fail_reports:           <report-name>, ... ;
     interrupt_activity:     <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.

- `codel_control` is a codel (`C` function) which is executed by the control task and which controls the validity of the input parameter.

- `fail_reports` is a list of possible reports returned by the control codel (the special report "OK" is always implicitly defined).

- `interrupt_activity` 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>;
     codel_control:        <codel-name>;
     codel_start:          <codel-name>;
     codel_main:           <codel-name>;
     codel_end:            <codel-name>;
     codel_inter:          <codel-name>;
     codel_fail:           <codel-name>;
     fail_reports:         <report-name>, ... ;
     interrupt_activity:   <exec-rqst-name>, ... ;
};
```

- `exec_task` is the name of the execution task in charge of the codels execution.
- `posters_input:` structure types of the posters that the execution codels of this request will read (see § 8.3 on page 74).
- `codel_activity_start`, `codel_activity_main`, `codel_activity_end`, `codel_activity_inter` and `codel_activity_fail` are the codels of this service. All fields are optional, but at least one of `codel_activity_start`, `codel_activity_main` or `codel_activity_end` must be defined. Be careful that all activities can be interrupted: do not forget to fill-in `codel_activity_inter` if something must be done in such a case. Note that `codel_activity_fail` is rarely used. Codels are further described in chapter 6.
- 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$^{\text{en}}_{\text{o}}$M builds an executable called `<module>Init` (where `<module>` is the name of the module). 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.

### 4.2.4 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$^{\text{en}}$oM in the file `server/<module>Poster.h` and its name is `<MODULE>_<POSTER>_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 2) 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:                 <<name>>::<<type-name>>, ...;
     exec_task:            <<exec-task-name>>;
     codel_poster_create:  <codel>;
};
```

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

- `codel_poster_create` 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.

### 4.2.5 Execution tasks declaration

```
exec_task <<exec-task-name>> {
     period:             <number>;
     delay:              <number>;
     priority:           <<number>>;
     stack_size:         <<number>>;
     codel_task_start:   <codel>;
     codel_task_end:     <codel>;
     codel_task_main:    <codel>;
     codel_task_main2:   <codel>;
     posters_input:      <poster-type>, ...;
     error_reports:      <report-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). *Remark:* previously the period was to be *a divisor or a multiple* of 20. (e.g. 1, 2, 4, 5, 10, 20, 40, 60, ... . This constraint has been raised.

- `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. Pertinent only on real-time OS.

- `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. Pertinent only on real-time OS.

- `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. Note that under Unix, stack size are not used at this time (the stack is grown dynamically). Pertinent only on real-time OS.

- `codel_task_start` 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).

- `codel_task_end` is the symmetric of the `codel_task_start`. It is called once, just before the module exits.

- `codel_task_main` is a *permanent* codel. It is executed each time the execution tasks wakes up (*before* all the other activites). Thus, for a periodical task, it is also periodical.

- `codel_task_main2` is a *permanent* codel. It is executed each time the execution tasks wakes up (*after* all the other activites). Thus, for a periodical task, it is also periodical.
- `posters_input:` structure types of the posters that the execution codels of this task will read (see § 8.3 on page 74).
- `error_reports` is a list of reports that can be reported by the permanent activity `codel_task_main`. Since this activity does not belong to a request, its reports are stored in the *control poster* of the module.

# Chapter 5

# Module generation

## 5.1 The `genom` command

### Synopsis

```
genom [-Ocdinxyto] [-Ipath] [-Dmacro] <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:

`-O` *Accept **O**bsolete syntax:* the syntax of `.gen` files has changed a bit over time. This option tells genom to still accept input syntax that is now considered as obsolete. Warning: this option can produce output code that is incompatible with what is generated without it. Use at your own risk.

`-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 7.2).

`-o` *OpenPRS libraries generation:* Generates OpenPRS interface libraries. They are mandatory if you wish to control this module from an OpenPRS program.

`-x` *Tclserv C Client generation:* Generates Tclserv C interface libraries. They can be used to control this module in C from a remote host, using Tclserv protocol.

`-y` *Python client generation:* Generate Python interface (using ctypes) to manipulate the different GeNoM structure. They can be used further to create Python controller, or to directly access to posters.

`-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 `.genom/codels/`. 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 4).

## 5.2 Product of the generation

The module generation produces files in the two directories `codels/` and `server/`. The files located in the `codels/` directory are described in the next chapter. This section describes the files located in the `server/` 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 D for information on the file-system hierarchy.

### 5.2.1 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`. 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, function prototypes, ...).

To use this library, you must link your program with `-ldemoClient`.

#### 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. `demoPosterLib.h` defines the name of the posters along with their data structures and prototypes of access functions. The *cIDS* structure is also defined there[1].

To use this library, you must link with `-ldemoClient`.

### 5.2.2 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_reports` field) of this module.

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).

### 5.2.3 Tcl library

The files `demoTcl.c` and `demo.tcl` are used by Tcl to control the module. See section 7.2 in this document.

### 5.2.4 Openprs library

The files located in the `openprs/` directory are used by OpenPRS to control the module. See section 7.3 in this document.

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

### 5.2.5 TclservClient Library

The files located in the `tclservClient/` directory can be used by a C client to remotely control the module. To use this lib, you need to link with `-ldemoTclservClient`. See section 7.5.5 in this document for more details.

### 5.2.6 Executables: server, test program, initialization request

The executable `demoTest` is an interactive test program. To use it, you just have to run `${TARGET}/demoTest`.

The files `demoCntrlTask.c`, `demoMotionTask.c` and `demoModuleInit.c` contain the module's execution tasks source code. Once compiled, they produce the files `libdemoServer.a`. 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` (the executable is in `${prefix}/bin`.

### 5.2.7 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 `demoTest`.

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

# Chapter 6

# 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.

## 6.1 Different kinds of codels

### 6.1.1 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 `codel_control`) 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 `codel_main*`) 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.

### 6.1.2 Codels associated to execution tasks

Three codels can be optionally defined for each execution task:

- **Initialization codel** (`codel_task_start`). 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** (`codel_task_end`). This codel is executed by the execution task upon destruction of the module, just after it has stopped serving requests.

- **Permanent codel** (`codel_task_main`). This codel is executed each time the task wakes up (therefore periodically if the task is periodic). This codel creates a permanent activity.

## 6.2 Simple examples of codels

### 6.2.1 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 `demoSetSpeedCntrl` 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 2).

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

### 6.2.2 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) */
   codel_main:            demoHypotExec;      /* execution codel */
   exec_task:             MotionTask;         /* execution task */
   interrupt_activity:    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 G$^{en}$oM 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)/server -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 6.7 page 48 and 6.8 page 51 will present more complex examples.

## 6.3 Codel files and compilation

### 6.3.1 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 `.genom/codels/` directory, so that you can fuse parts together by yourself. The `genom-mode` can help you to do this: see chapter 4.

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.

### 6.3.2 Makefiles

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

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

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`.

## 6.4 Accessing the IDS

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

### 6.4.1 The fIDS

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

*Warning:* You should not read parameters of a request directly from the fIDS, but only from the codel function parameters.

Indeed, 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$^{\text{en}}$oM generates arrays for that case. The member you have defined in the *fIDS* still exists, but is not used.

### 6.4.2 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 `server/<module>Header.h`.

This cIDS is also exported in the control poster of the module.

Warning: the control poster belongs to the control task of the module. It updates this poster every time it wakes up, ie, when it receives a request or when an activity is over. However, the activity state transitions, executed by the execution tasks, are usualy much more frequent. Thus, the control poster is not updated in real time.

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`.

## 6.5 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_reports` of every request. These are strings, which 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.

### 6.5.1 Numerical values of reports

Reports which are declared in the `fail_reports` field are mapped into 32 bits integers using the VxWorks convention:

The name of this integer is build as follow: `S_<source>_<report>`, where `<source>` is the name of the module or the library which defines the `<report>` string. For instance: `S_demo_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 $N$.
- the 16 lowest significative bits encode the id of the report inside the module, and is computed by G$^{\text{en}}_{\circ}$M.

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

## 6.6 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$^{\text{en}}_{\circ}$M (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$^{\text{en}}_{\text{o}}$M 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);
```

## 6.7 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.

### 6.7.1 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 (`codel_main`). 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 monitors the position of the mobile and throw an alert when it has the requested value, say, `myPos`. The execution codel `demoMonitorExec` (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
demoMonitorExec(double *monPos, double *pos, int *report)
{
   /* 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 (`codel_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.

Similarly, a termination phase would be defined thanks to the *termination codel* (the `codel_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`.

### 6.7.2 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.

### 6.7.3 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 `codel_inter` codel is executed. By default, there is no such codel, and the activity goes immediately 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 being interrupted. In that case, define a `codel_inter` codel which will handle the necessary operations.

### 6.7.4 States and transitions of activities

The different states an activity can go through are shown on figure 6.1. The external ring corresponds 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.

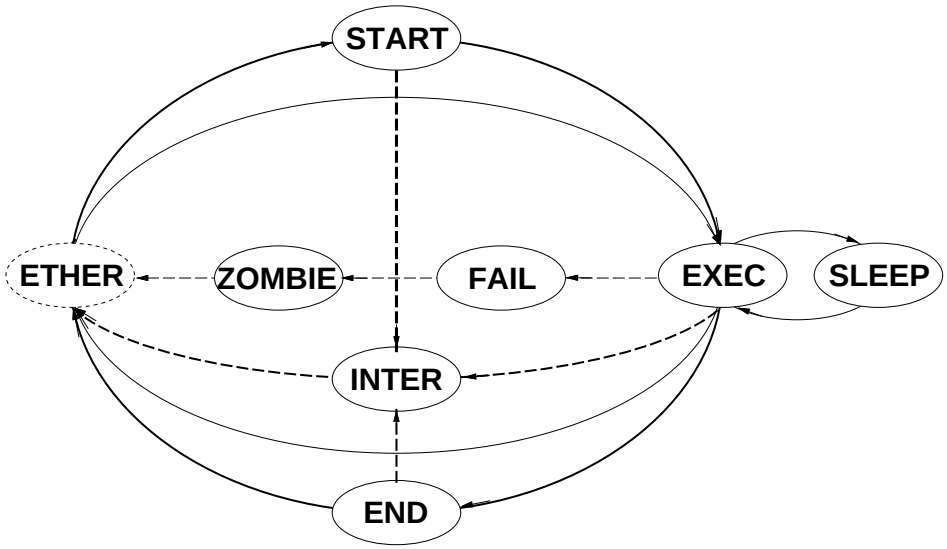


Figure 6.1: States and transitions of activities.

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 | `codel_start` |
| `EXEC` | main execution state | `codel_main` |
| `END` | termination state | `codel_end` |
| `FAIL` | failure (and module freeze) | `codel_fail` |
| `INTER` | interruption state | `codel_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 described 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 | (codel_start) | X | X | X | X | X | X |
| EXEC | (codel_main) | | X | X | X | X | X |
| END | (codel_end) | | | X | X | X | X |
| INTER | (codel_inter) | | | | X | X | X |
| FAIL | (codel_fail) | | | | | X | X |

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

## 6.8 Writing the codels

### 6.8.1 Control codels `codel_control`

See example in section 6.2.1, page 42.

### 6.8.2 Execution codels `codel_*`

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 (`codel_main`) of the `Goto` 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 *report)
{
   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) {
        *report = S_demo_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) {
            *report = S_demo_MOBILE_OUT_OF_ORDER;
            return ETHER;
        }
        return END;
   }

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

Notes:

The `SDI_F` macro let you access 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.

### 6.8.3 Initialization codel `codel_task_start`

One initialization codel can be associated to any execution task, by the mean of the field `codel_task_start`. It is usually used to initialiaze 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 initializes 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 *report)
{
  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;
}
```

### 6.8.4 Permanent activity codel `codel_task_main`

A permanent activity can be defined for any execution task, by the mean of the field `codel_task_main`. 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.

## 6.9 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 activity is identified by 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 `codel_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 `codel_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 codel_start of the "Motion" activity */
ACTIVITY_EVENT
demoMotionStart(MOTION_STR *params, int *report)
{
    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 codel_end of the "Motion" activity */
ACTIVITY_EVENT
demoMotionEnd(MOTION_STR *params, int *report)
{
    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.

## 6.10 Coding advice

### 6.10.1 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, the VxWorks programmers manual recommends the use of 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.

### 6.10.2 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 necessarily bounded in time. Modules generated by G$^{\text{en}}_{\text{o}}$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 7 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 7

# Using modules

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

## 7.1 The interactive test program `Test`

The interactive test program `<module>Test` is a client of a module. One can launch several instances of it, provided they are given different *numbers* (`<module>Test 5`, `<module>Test 2`, etc.).

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.

### 7.1.1 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. Initially default values are set to 0. Then they keep the previous entered value.

Note: if you have defined another request (or a poster) that outputs the same IDS data as this input parameter, then the `<module>Test` suggests to initialize automatically this input parameter using the other request (or poster). That means that the input parameter will be initialized with the current module 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).

### 7.1.2 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.*

## 7.2 The interactive tcl shell `tclserv`

`tclserv` allows to control a set of modules (requests and posters) from a `tcl` shell. Thus, you can take advantage of a full and interactive programming language (tcl) to manipulate requests and data. Moreover, you can control modules running on distinct CPUs.

`tclserv` is a server which connects to a list of modules on one side, and accepts tcl clients on the other side. `tclserv` must run on the machine that runs the modules. If the modules run on several machines then you must run one `tclserv` on each machine.

The `tcl` client can run on any machine. It is a standard `tcl` shell, loading the `tcl` package "`genom`". However, we provide such a shell

Here is a typical sequence to control modules with tcl:

1. generate the modules with the option `-t`.
2. compile and install them
3. start the modules
4. start a server `tclserv` on the same machine
5. on any machine, start a tcl shell with the package genom: `eltclsh -package genom`
6. from ths shell, connect to the machine running `tclserv`:
   `connect <tclserv-host>`
7. load the functions to communicate with the modules:
   `lm <module1>`
8. send the requests to the modules:
   `module1::Move 1.0`

`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.

## 7.3 OpenPRS and transGen

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

## 7.4 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.

### 7.4.1 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 substructure `Ref` (see page 47 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 `demoPosterLib.h`.

#### Third step: compilation

Compilation is handled by the GNU autoconf framework. It's sufficient to specify in the `requires` list of the new module, the list of modules names it is reading posters from.

The `configure` script will check that the necessary libraries are present in your openrobots directory and generate the appropriate options in the generated Makefiles.

### 7.4.2 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);
```

## 7.5 Accessing modules services from another process

### 7.5.1 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.

### 7.5.2 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`.

### 7.5.3 Sending requests and receiving replies

You can send requests 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/demoMsgLib.h`. Generic functions (as for posters) also exists, and are documented in the sections below).

### 7.5.4 An example

We present here a few examples, which involve a task `pilo` that must send the `SetSpeed` and `Goto` requests to the module `demo`.

#### 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.
- 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:

```
if (demoSetSpeedRqstAndRcv(PILO_CMDTASK_DEMO_CLIENT_ID,
                           speed, report) != FINAL_REPLY_OK) {
   /* FAIL */
}
```

**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:

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

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

**Receiving replies: `ReplyRcv`**

```
int ...ReplyRcv(CLIENT_ID clientId,
                int rqstId, int block,
                [OUT *out], int *activity, int *report);
```

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. We use the static variables defined in the previous example.

```
   switch (demoMonitorReplyRcv(PILO_CMDTASK_DEMO_CLIENT_ID,
                               demoMonitorRqstId, NO_BLOCK,
                               &piloDemoMonitorOut, &demoMonitorActivity,
                               &report)) {
      case WAITING_FINAL_REPLY:
/* SLEEP */
      case FINAL_REPLY_OK:
         piloDemoMonitorRqstId = -1;
/* DONE */
      default:
         /* ZOMBIE */
  }  /* switch */
```

### 7.5.5 The library `tclservClientMsgLib`

To generate this library, you need to call G$^{en}$oM with the -x option.

Before you can use this library, you need to create a connection with a TclServ. This is done with the function `tclserv_client_init` .

Then, you must initialize connections for individual module : for instance `demoTclservClientInit` in the library `demoTclservClientMsgLib`. One connection with a TclServ, identified by a `TCLSERV_CLIENT_ID` can be shared by multiples modules.

Before you quit, you must free this connection with `demoTclservClientEnd` and close the connection with the Tclserv with `tclserv_client_destroy`.

### 7.5.6 Sending requests and receiving replies witch tclservClientMsgLib

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

| suffix | function |
|---|---|
| `RqstSend` | send a request (non blocking) |
| `ReplyRcv` | wait for a final reply (blocking ) |
| `RqstAndRcv` | send a request *and* receive the *final reply* (blocking) |
| `RqstAndAck` | send a request *and* receive the *intermediate reply* (blocking) |

The library design is the same than `MsgLib`, so usage described in 7.5.3 are still valid.

### 7.5.7 API description

**Sending an request: `RqstSend`**

```
  int ...RqstSend(TCLSERV_CLIENT_ID clientId,
                  ssize_t *rqstId,
                  [STR_IN *in,]
)
```

- The first argument is the client identifier
- `rqstId` is filled with the request id. This id will let you read the reply later or to
- the third argument is optional, and defined only if the request defines an input.

The function returns `0` in success case, and `-1` in failure case.

**Sending an request and wait for acknowledgement: `RqstAndAck`**

```
  int ...RqstAndAck(TCLSERV_CLIENT_ID clientId,
                  ssize_t *rqstId,
                  [STR_IN *in,],
                  int* report
)
```

The argument are the same than the previous request. It only get one more argument, to store the report value.

**Sending and wait for a request : `RqstAndRcv`**

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

- The first argument is the client identifier.
- 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 `0` if everything went well, `-1` if the request has failed (in this case, report has some valid values), or `-2` in the case of internal failure.

**Getting final reply: `ReplyRcv`**

```
int ...ReplyRcv(TCLSERV_CLIENT_ID clientId,
                ssize_t rqstId,
                [STR_OUT *out,]
                int* report
)
```

- The first argument is the client identifier.
- `rqstId` is the request descriptor.
- The third argument is optional, and defined only if the request defines an output.
- The last argument stores the error value, if any.

The function returns `0` in success case (and out is filled, if it exists), `-1` in failure case (report is filled, and must be freed by the user), or `-2` in the case of internal failure.

### 7.5.8 Checking if a request is terminated

You can use the generic function

```
int tclserv_client_has_terminated(TCLSERV_CLIENT_ID clientId, ssize_t rqstId)
```

- the first argument is the client identifier
- `rqstId` is the request descriptor

The function returns `-2` in case of internal failure, `-1` if the request is not terminated and `0` otherwise.

## 7.6 Sharing modules between different unix users

By default, modules and clients can only communicate between processes owned by the same unix user id. This is so because communication libraries use special files located in the `${HOME}` directory of the user running the processes.

In most situations, this is the desired behaviour since this allows different users to work on the same machine without unwanted interaction. However, in some situations, you might want to share the modules and clients between different users. To do so, you must define the `H2DEV_DIR` environment variable and make it point to the same location for all users that must interact together. For instance, you can set it to `/tmp` with `export H2DEV_DIR=/tmp`

(sh, bash, ...) or `setenv H2DEV_DIR /tmp` (csh, ...). This must be done before running any G$^{en}$oM related process (server or client) as well as `h2 init`. Note that only the first user (the "owner") of the shared `H2DEV_DIR` needs to execute the `h2 init` command and execute the `h2 end` command after all modules and clients are shut down.

For information, the special files created under `${H2DEV_DIR}` are the following:

**`.h2dev-<hostname>`** Created by `h2 init` and used by the communication libraries to retrieve the shared communication objects.

**`.<module>.pid-<hostname>`** Created when starting a G$^{en}$oM module. This file contains the `pid` of the module, and is used by some scripts such as `killmodule`.

# Chapter 8

# GenoM syntax

This chapter gives a description of the different commands defining a G$^{en}$oM module. These commands are organized into sections corresponding to the different parts of a GenoM module.

## 8.1 Module Declaration

`module moduleName { ... };` declaration of module with name `moduleName`.

The sub-commands of `module` are the following:

`number: n;` non-negative integer uniquely identifying the module in a given control architecture.

`version: "major.minor";` defines the version number of the module

`email: email;` defines the e-mail address of the person to contact for information about this module.

`requires: packageOrModule1 , ..., packageOrModuleN;` specifies the dependency of the module. When generating the module, G$^{en}$oM will look for `pkgconfig` input files `packageOrModule1.pc`,...,`packageOrModuleN.pc` and set the necessary compilation flags accordingly.

`codels_requires: packageOrModule1 , ..., packageOrModuleN;` specifies the dependency of the codels. When configuring the codels, G$^{en}$oM will look for `pkgconfig` input files `packageOrModule1.pc`,...,`packageOrModuleN.pc` and set the necessary compilation flags accordingly.

`lang: "c++"` or `"c"`: specifies the programming language with which the codels are developed.

`internal_data`: specifies the name of the internal database. The internal database is a standard C data-structure that should include no pointer.

`clock_rate`: specifies the rate of the internal clock producing the system tick, on which period tasks are synchronized. This declaration is optional. The default value, if not specified is 100 ticks per second.

## 8.2 Requests

`Request requestName { ... };` declaration of a request with name `requestName`.

The sub-commands of `Request` are the following.

`doc: "comment";` inline help of the request.

`type: control`, `exec` or `init` defines the type of the request:

- `control`: for a control request; control requests do not trigger any specific action. They are mostly used for accessing the internal database (reading or changing parameters),
- `exec`: for an execution request; execution requests trigger (periodic or not) actions defined into execution codels,
- `init`: for the initialization request of the module. No execution request can be treated until this request has been called and has returned success.

`exec_task: execTaskName;` defines which execution task executes the request. Applies only for *execution* and *initialization* tasks.

`input: name::idbRef;` defines the input of the request. `name` is the name of the variable that will be defined in the request codels. `idbRef` is the field of the internal database defining the type of the codels input. For instance:

```
module Test {
    internal_data:          TEST_STR;
    number:                 100;
};

typedef struct MY_STR {
  double a1;
  double a2;
} MY_STR;

typedef struct TEST_STR {
  MY_STR myStr;
} TEST_STR;

request Req {
  ...
  input:              a1::mStr.a1;
  ...
}
```

defines a request with as input a `double`

`input_info: defaultValue1::"doc1" , ..., defaultValueN::"docN";`

`posters_input: STRUCT_NAME_1 , ..., STRUCT_NAME_N;` specifies that the codels of the request need to read information on posters exporting data-structures of types `STRUCT_NAME_1`, ..., `STRUCT_NAME_N`. GenoM server part then defines functions to read the poster with the following prototypes:

```
STATUS moduleNameSTRUCT_NAME_1PosterRead(POSTER_ID posterId, STRUCT_NAME_1* x);
STATUS moduleNameSTRUCT_NAME_1PosterFind(char *posterName, POSTER_ID *posterId);
...
STATUS moduleNameSTRUCT_NAME_NPosterRead(POSTER_ID posterId, STRUCT_NAME_1* x);
STATUS moduleNameSTRUCT_NAME_NPosterFind(char *posterName, POSTER_ID *posterId);
```

`output: name::idbRef;` output of the request. The type of the output is defined by field `idbRef` of internal database similarly as input.

`codel_control: codelName;` specifies the name of the control codel of the request. The control codel is called before any other codels when request is triggered. If the control codel returns `ERROR`, the request ends. The prototype of the control codels is defined by the input and output of the request.

`codel_start: codelName;` (only for initialization and execution tasks.) specifies the name of the start codel of the request. Called after the control codel. The start codel can return:

- `EXEC`: the function defined by `codel_main` is called at the next execution period,
- `END`: the function defined by `codel_end` is called at the next execution period,
- `ETHER`: the request terminates.

`codel_main: codelName;` (only for initialization and execution tasks) specifies the name of the main codel of the request. The main codel can return:

- `EXEC`: the main codel is called again at the next execution period,
- `END`: the function defined by `codel_end` is called at the next execution period,
- `ETHER` the request terminates.

`codel_end: codelName;` (only for initialization and execution tasks) terminates request execution and returns `ETHER`.

`codel_inter: codelName;` (only for initialization and execution tasks) specifies the name of the interruption codel. The interruption codel is called when the request is interrupted by another request defined by field `interrupt_activity`

`fail_reports: reportName1 , ..., reportNameN;` possible error messages generated by the request. Error messages are transformed into unique integer values stored into C macros with names:

`S_moduleName_reportName1, ..., S_moduleName_reportNameN`

where `moduleName` is the name of the module.

`interrupt_activity: RequestName1, ..., RequestNameN;` defines the names of the requests of this module that interrupt this request when triggered by a client. `RequestName1, ..., RequestNameN` can be replaced by `all` to denote all the requests of the module.

## 8.3 Posters

`poster posterName { ... };` declaration of poster with name `posterName`.

The sub-commands of `poster` are the following.

`update: user` or `auto;` defines how the poster is updated, either automatically (`auto`) or by the user in the codels (`user`)

`type: structName1, ..., structNameN;` defines the types of the sub-fields of the poster. Can be used only with `update: user;`.

`data: name1::idbRef1 , ..., nameN::idbRefN;` defines the types of the sub-fields of the poster by internal database fields.

`exec_task: taskName;` defines the name of the task that updates the poster.

`codel_poster_create: codelName;` defines the codel that creates the poster. Can be used only with `update: user;`.

## 8.4 Execution Tasks

`exec_task taskName { ... };` defines a new execution task. The sub-commands of an execution task declaration are the following.

`period: number;` defines the period of the task in ticks. The number of ticks per second is defined by the global `clock_rate` definition of the module.

The special value **`none`** can be used to create asynchronous execution tasks.

`delay: number;` defines the offset of the task with respect to the beginning of the period.

`priority: number;` defines the priority of the task.

`stack_size: size;` defines the size of the task stack.

`posters_input: STRUCT_NAME_1 , ..., STRUCT_NAME_N;` specifies that the codels of the task need to read information on posters exporting data-structures of types `STRUCT_NAME_1`, ..., `STRUCT_NAME_N`. GenoM server part then defines functions to read the poster with the following prototypes:

```
STATUS moduleNameSTRUCT_NAME_1PosterRead(POSTER_ID posterId, STRUCT_NAME_1* x);
STATUS moduleNameSTRUCT_NAME_1PosterFind(char *posterName, POSTER_ID *posterId);
...
```

```
STATUS moduleNameSTRUCT_NAME_NPosterRead(POSTER_ID posterId, STRUCT_NAME_1* x);
STATUS moduleNameSTRUCT_NAME_NPosterFind(char *posterName, POSTER_ID *posterId);
```

`codel_task_start: codelName;` defines the codel called at initialization of the task when the module is launched.

`codel_task_end: codelName;` defines the codel called when the task ends.

`codel_task_main: codelName;` defines the codel systematically called by the task (before all the other activities).

`codel_task_main2: codelName;` defines the codel systematically called by the task (after all the other activities).

`codel_task_wait: codelName`: defines a "*waiting*" codel for asynchronous execution tasks, that is called before `codel_task_main`. When this codel is called, it doesn't have the lock on the internal data structure of the module. Its goal is to block the task, waiting on availability of data on a file descriptor or a semaphore.

It's not permitted to declare a `codel_task_wait` codel for a periodic task.

`fail_reports: reportName1 , ..., reportNameN;` possible error messages generated by the task. Error messages are transformed into unique integer values.

# 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

```
blues% 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

```
blues% ./codels/i386-linux/demo -b
Hilare2 execution environment version 2.11
Copyright (c) 1999-2011 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 "Test" program failures

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

**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 `valgrind` or `purify` to help debugging these issues.

# 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, ...) |
| `posterForget` | forget about previously found poster |
| `posterDelete` | delete a poster |
| `posterShow` | display the state of every poster |
| `posterMemCreate` | create a poster at the given address |

| | | |
|---|---|---|
| STATUS | posterCreate | (const char *name, int size, POSTER_ID *id); |
| STATUS | posterFind | (const 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, void *parg); |
| STATUS | posterForget | (POSTER_ID id); |
| STATUS | posterDelete | (POSTER_ID id); |
| STATUS | posterShow | (void); |
| STATUS | posterMemCreate | (const 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

### 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.

# Appendix C

# Genom module instances

As of Genom version 2.8.2, it is possible to have several instances of the same module running on one machine.

Genom module instances are supported in the C/C++ and Tcl bindings for Genom.

## C.1 Identifying instances

A genom module instance is identified by a string suffix added to the module name in the mailbox name and all poster names (plus a few internal pocolibs identifiers that are normally not used directly by Genom modules).

In this documentation an **instance name** of a module is the name of the module plus the instance suffix. For example for module `demo`, if the instance "`0`" is created, the instance name is `demo0`.

Note that the suffix should be a short string choosen to keep the instance name as a valid identifier in the languages used by Genom. Including a word separator, such as whitespace or punctuation characters, is going to create various, undefined behaviours.

The **default instance** of a module, is a module launched whitout specifying any instance. It uses the bare module name as identifier base for its mailbox and posters.

## C.2 Starting a module instance

To launch a module with an instance name, the `GENOM_INSTANCE_`*module* environment variable should be to the specified instance suffix.

For examle, for module demo, instance `demo0` is created by:

```
GENOM_INSTANCE_demo=0
```

## C.3 Accessing data in a module instance

## C.4 C language

Instance aware poster access functions are generated by Genom:

`modulePosterNameInstancePosterRead(instance_name, data)` reads the poster of the specified `instance_name`.

## C.5 tclserv

Load module instances as aliases :

`lm demo as demo0`

loads the instance `demo0` of the demo module. It creates the `demo0` name space which will allow to interact with that instance of the module.

More instances of the same module can be loaded too:

`lm demo as demo1`

will load the `demo1` instance of the module and create the corresponding namespace.

# Appendix D

# 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).

**XXX still not fully up-to-date**

## D.1 Source files in `server/`

| 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.[ch]` | Requests libraries. |
| `demoConnectLib.c` | Smaller version of the library `demoMsgLib.c`: use this if you use `csLib` directly. |
| `demoPosterLib.[ch]` | Posters libraries. |
| `demoError.h` | Error codes (reports). |

| Test program: | |
|---|---|
| `demoTest.c` | interactive test program |
| `demoScan.[ch]` | interactive scan of input structures |
| `demoPrint.[ch]` | display the C structures of the module |

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

| Python : | |
|---|---|
| `struct.py` | ctypes python interface to manipulate structure |

| OpenPRS: | |
|---|---|
| `openprs/` | |

| Tclserv Client: | |
|---|---|
| `tclservClient/demoTclservClientDecode.c` | Transform string returned by `tclserv` into a C structure |
| `tclservClient/demoTclservClientEncode.c` | Transform a C structure into a string for `tclserv` |
| `tclservClient/demoTclservClientMsgLib.c` | Use it to control modules using `tclserv` protocol |
| `tclservClient/demoError.c` | Encode and decode module error encoded by `tclserv` protocol |

## D.2 Binary files

- Libraries:
  - Server side:

| | |
|---|---|
| `demoCntrlTask.o` | `demoServeur.a` |
| `demoDeplacementTask.o` | |
| `demoModuleInit.o` | |
| `demoInit.o` | `demoInit*` |

  - Client side:

| | |
|---|---|
| `demoConnectLib.o` | |
| `demoTest.o` | `demoTest*` |
| `demoScan.o` | `demoClient.a` |
| `demoPrint.o` | |
| `demoMsgLib.o` | |
| `demoPosterLib.o` | |