The gPROMS Server Guide v2.3

Release 3.5
April 2012

Process Systems Enterprise Limited

6th Floor East

26-28 Hammersmith Grove

London W6 7HA

e: info@psenterprise.com

w: www.psenterprise.com

t: +44 20 8563 0888

f: +44 20 8563 0999

The gPROMS Server Guide v2.3

Release 3.5

April 2012

Trademarks

Disclaimer

Table of Contents

1 Introduction

The gPROMS Server (gSERVER) makes gPROMS' powerful capabilities available to other application programs. The gSERVER is precisely the same modelling and solution engine as used by gPROMS itself, and has the complete set of features and capabilities.

1.1 gPROMS-based Applications (gBA)

A software program making use of the gSERVER is called a *gPROMS-based Application* (gBA). The gSERVER allows the gBA to:

- construct complex process models expressing them in the gPROMS language;
- perform sophisticated *model-based activities* including
  - steady-state and dynamic simulation;
  - steady-state and dynamic optimisation;
  - parameter estimation from steady-state and dynamic experiments

  or combinations thereof;
- have full access to the mathematical descriptions of the above models and activities, e.g.
  - the set of differential-algebraic equations that describe a process model;
  - the mathematical statement of a dynamic optimisation problem.

Typical examples of gBAs that may be constructed using the gSERVER include:

- specialist modelling tools with domain-specific user interfaces;
- process operations decision support tools;
- operator training systems;
- model-predictive control;

and so on. Most of these gBAs will normally present their users with their own user interface. Thus, users are not expected to interact with gPROMS directly; indeed, they may even be completely unaware of the existence or use of gPROMS.

1.2 This document

This document describes version 2.3 of the gSERVER. This is the modelling and simulation engine used within gPROMS v2.3.

The document is aimed at system integrators involved in the development of new gBAs and the maintenance of existing ones. It is structured as follows:

- Part I provides an overview of the operation and use of the gSERVER. It is important to study and understand this material thoroughly in order to make the best use of the powerful facilities provided.

- Part II is a reference guide to the functions provided by the gSERVER.
- Part III contains additional information that may be of relevance to FORTRAN-based gBAs, and also provides some examples of the usage of the gSERVER.

1.3 Prerequisites

The reader is assumed to have a good degree of familiarity with gPROMS concepts and operations, as described in the “*gPROMS Introductory User Guide*” and “*gPROMS Advanced User Guide*”.

For more sophisticated gBAs, some of the material described in the “*gPROMS System Programmer Guide*” may also be required, as indicated by the references in the text.

Part I: gSERVER Overview

2 Overview of gSERVER for the gBA Developer

2.1 The gSERVER directory structure

The gSERVER software is provided as part of the standard gPROMS installation, the directory structure of which is shown in Figure 1. The files that are relevant to the gSERVER are also shown in this diagram[1].

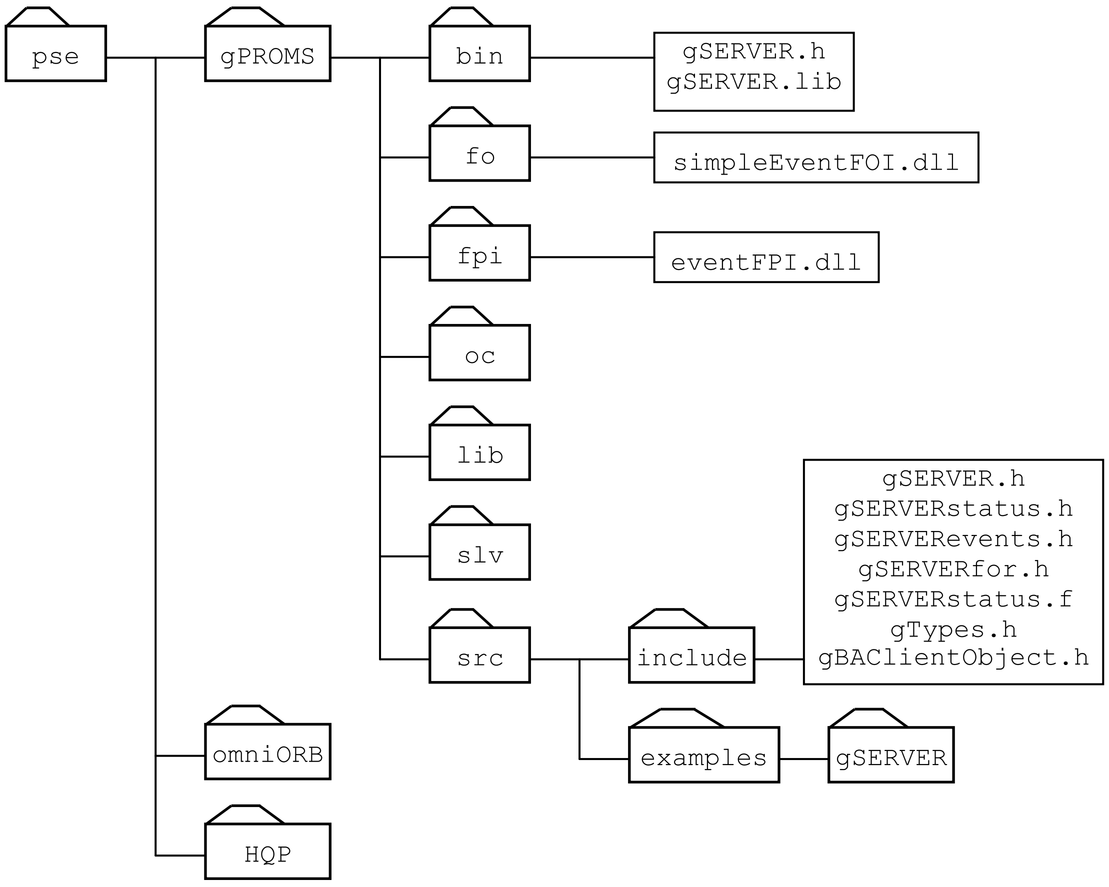

Figure 1: Directory structure of typical gPROMS installation.

2.2 Typical gBA installation directory structure

A gBA will typically consist of three parts, namely:

- the actual gBA code (usually in object code and/or executable form) and other associated files (e.g. databanks) that are used by it but are unrelated to gPROMS;
- the gSERVER software;

[1] Some of the files shown in Figure 1 are shown with extensions that are specific for Microsoft Windows platforms; on most UNIX systems, the extension `.dll` would be replaced by `.so` (or `.a` for UNIX AIX-based systems).

- other files that are used by gPROMS including:
  - implementations of foreign objects and output channels, and any auxiliary files used by these (e.g. property databanks for physical property foreign objects);
  - pre-configured gPROMS input files or parts thereof (see section 3.3);
  - saved gPROMS results (`.gSTORE`) files used for initialising the gPROMS calculations.

It is recommended to:

- keep the gSERVER software in a separate directory structure in the form described in section 2.1;
- place any foreign object and output channel implementations in the corresponding `fo` and `oc` sub-directories (see Figure 1);
- place any files used by the foreign object implementations anywhere in the gBA installation (e.g. in a separate directory, or in the `fo` sub-directory sub-directory) while ensuring that the foreign object code refers to them by their *absolute* pathnames;
- place any other files that are needed for the execution of gPROMS activities (e.g. pre-configured gPROMS input files and/or saved results files) anywhere within the gBA installation, making sure that the gBA copies them to the user's `input` or `save` directories *before* initiating the gPROMS execution (see section 3.3.2).

The overall gBA installation will then be as shown in Figure 2.

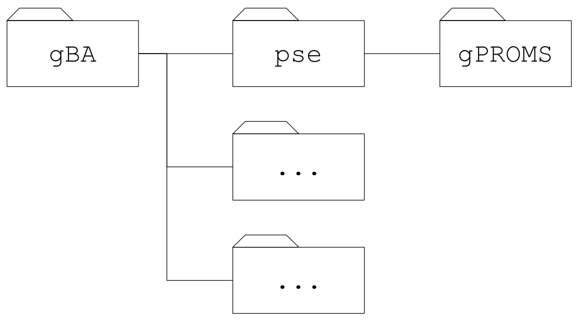

Figure 2: Suggested gBA directory structure.

It is worth noting that:

- The files in the `include` sub-directory are necessary only for gBA development. They do not need to be delivered to the gBA users as part of the final gBA installation.
- All other directories contain files that are potentially necessary both for developing and for running gSERVER.

- The `fo` sub-directory must contain the shared objects or dynamic link libraries that implement any foreign objects (e.g. external physical property packages) being used by the gBA.
- The `oc` sub-directory must contain the shared objects or dynamic link libraries that implement any output channels being used by the gBA.
- The contents of the `examples` sub-directory are explained in more detail in section 14.

The directories `HQP` and `omniORB` have to be distributed together with all gBAs for copyright reasons.

3 Developing a New gBA

3.1 Program structure for a simple gBA that uses a gPROMS activity

At the simplest level, a gBA that makes use of a gPROMS simulation, optimisation or parameter estimation activity will typically involve a minimal number of steps:

1. Start the gSERVER by invoking function `gproms_start` (see section 6.1).
2. Load into the gSERVER the model of interest in the form of one or more gPROMS input files by invoking function `gproms_select` (see section 7.1).
3. Execute a simulation, optimisation or parameter estimation activity by invoking functions `gproms_simulate`, `gproms_optimise` or `gproms_estimate` respectively (see sections 9.1-9.3 respectively).
4. Stop the gSERVER by invoking function `gproms_end` (see section 6.2).

As indicated by step 2 above, the gSERVER operates on problem descriptions in the form of gPROMS input files. The interactions between the gBA, the gSERVER and these files are discussed in more detail in section 3.3.

Of course, in addition to the steps mentioned above, a gBA will also normally wish to obtain some results from the activity executed at step 3. Mechanisms for achieving this are discussed in section 3.3.3.

3.2 Synchronous vs. asynchronous execution of activities

The simple sequence of steps described in section 3.1 assumes a *synchronous* type of interaction between the gBA and the gSERVER. This means that each gSERVER function invoked by the gBA completes its operation before returning control of program execution to the gBA itself.

However, albeit quite simple, the synchronous mode of interaction is not appropriate for all applications. For example:

- The execution of an activity (e.g. a complex dynamic optimisation calculation) may require significant amounts of computation. In such cases, it may be desirable to allow the gBA user to continue interacting with the gBA during this time (e.g. for the purposes of defining another problem to be solved or for viewing the results of an earlier run).
- In many applications, it is necessary to allow the user to interact with the activity while the latter is executing. This is typically the case for dynamic simulations using the gPROMS Foreign Process Interface (FPI).

In order to address the above requirements, gSERVER also supports an *asynchronous* mode of operation. A typical gBA program structure making use of this mode is as follows:

1. Start the gSERVER by invoking function `gproms_start` (see section 6.1).

2. Load into the gSERVER the model of interest in the form of a gPROMS input file by invoking function `gproms_select` (see section 7.1).

3. Initiate a simulation, optimisation or parameter estimation activity in asynchronous mode by invoking functions `gproms_simulate_mt`, `gproms_optimise_mt` or `gproms_estimate_mt` respectively (see sections 10.1-10.3 respectively).

4. Optionally:

   - interact with the events produced by the executing activity using the functions described in section 11;
   - monitor the status of the executing activity by invoking `gproms_get_activity_status` (see section 12.1);
   - if necessary, terminate the executing activity by `gproms_kill_activity` (see section 12.2).

5. Stop the gSERVER by invoking function `gproms_end` (see section 6.2).

The main difference between the above steps and the simpler version presented in section 3.1 is that the functions invoked at step 3 are now asynchronous, i.e. they simply initiate the requested activity and immediately return program execution control to the gBA.

The optional step 4 then allows the gBA to interact with the executing activity.

Typical synchronous and asynchronous gBA program structures are shown schematically in Figure 3. It will be noted that, in addition to the various gSERVER functions that have already been mentioned, these diagrams also refer to the function `gproms_redirect_output`, the invocation of which may be necessary for gBAs that have a graphical user interface; on the MS Windows platform it is strictly necessary to use this function in a GUI-based application. This function is discussed in more detail in section 6.3.

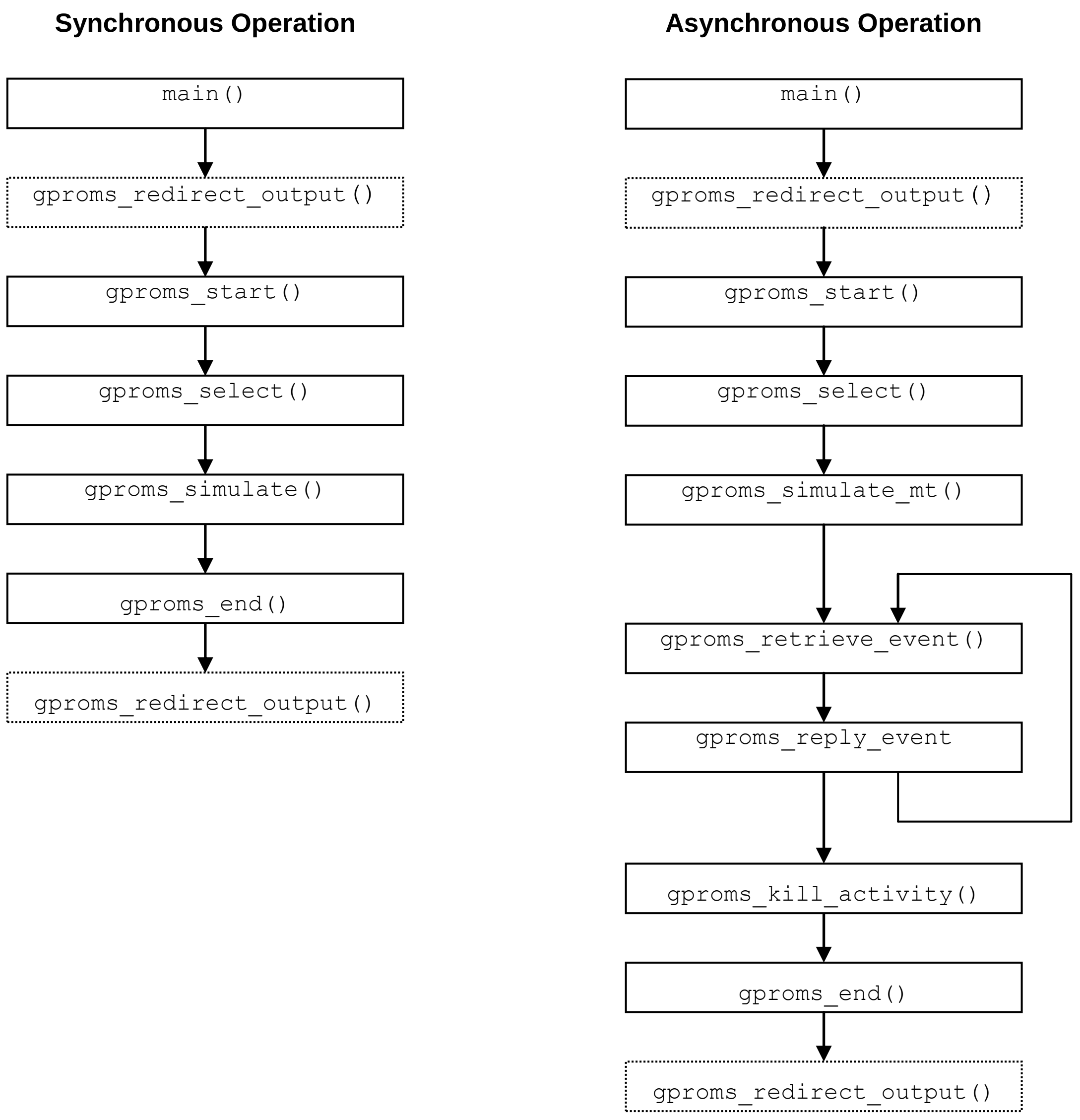

Figure 3: Synchronous and asynchronous operation of gSERVER.

3.3 gPROMS input files for gBA execution

At run-time, gSERVER operates on a set of gPROMS input files which describe the particular problem to be solved. These files depend on the type of activity for which the gBA wishes to use gPROMS. For example:

- A simulation activity requires a gPROMS input file (extension `.gPROMS`).
- An optimisation activity requires a gPROMS input file and an optimisation file (extension `.gOPT`).
- A parameter estimation activity requires a gPROMS input file, an estimation file (extension `.gEST`) and one or more experimental results files (extension `.RUN`).
- If the gBA makes use of a physical properties package, further files (e.g. databanks, or files configuring a particular type of material) may be required.

3.3.1 Using pre-existing gPROMS input files vs. creating them "on-the-fly"

For many gBAs, the input files described above may pre-exist, having been constructed "once-and-for-all" by the gBA developer using standard gPROMS facilities (e.g. the gPROMS ModelBuilder). In such cases, the files will be provided as part of the gBA installation directory structure (cf. section 2.2).

The use of fixed, pre-existing gPROMS input files is appropriate in cases where:

- The gBA involves always running exactly the same model with exactly the same types of specifications.
- There are only relatively minor differences, if any, from one gPROMS run initiated by the gBA to another. For example, the gBA users may be allowed to change the values to which certain variables are `ASSIGN`ed and/or `INITIAL`ised; such changes can be effected within a fixed gPROMS input file if the corresponding values are obtained via the invocation of appropriate gPROMS Foreign Object methods instead of being hardcoded inside the gPROMS input file. Alternatively, the gBA users may be allowed to interact with an executing simulation activity (cf. step 4 of the sample program sequence in section 3.2).
- The fixed gPROMS input file contains all Foreign Object Interface and/or Foreign Process Interface commands that are necessary to support the above interactions.

However, the use of fixed gPROMS input files constructed a priori may not be appropriate in all situations. For example, an advanced gBA may need to determine the precise form of the equations in a gPROMS `MODEL` entity and/or the schedule in a gPROMS `TASK` entity based on the information provided by its user via its own user interface. In such cases, all or some of the gPROMS input files may be constructed "on-the-fly" by the gBA itself prior to initiating its interaction with the gSERVER.

Finally, it is common to have hybrid situations in which an input file is assembled by the gBA partly from pre-existing file(s) and partly from files written by the gBA itself during its execution. A typical example of such an approach is one in which the `DECLARE`, `MODEL` and `TASK` parts of a gPROMS problem description pre-exist while the `PROCESS` entity is written on-the-fly by the gBA taking account of information specified by the user . It is the responsibility of the gBA to assemble all these in a single file in the right order prior to passing this file to the gSERVER.

3.3.2 The gBA user's working directory

Whatever the source for the gPROMS input files used by a gBA, at run-time gSERVER will expect them at certain locations. Usually, this is a sub-directory called `input` of the directory in which the user is executing the gBA (the "user's working directory"). The `input` sub-directory must either exist or be created on-the-fly by the gBA itself.

Similarly, any .`gSTORE` files that are provided with the gBA for the purpose of initialising the gPROMS model variables via by gPROMS `RESTORE` elementary tasks (cf. "*gPROMS Introductory User Guide*") need to be places in the `save` sub-directory of the user's working directory. In such cases, the `save` sub-directory must either exist or be created on-the-fly by the gBA itself.

It is the gBA's responsibility to place the above files in the directories mentioned. This may involve some copying of files from the gBA installation directory (cf. section 2.2) into the appropriate sub-directories of the user's working directory.

Any result files created by the gSERVER's operation will reside in a sub-directory called `output` of the user's working directory. Similarly, the execution of SAVE elementary tasks during a gPROMS simulation will result in the newly created `.gSTORE` file be placed in a sub-directory called `save` of the user's working directory. Both the `output` and `save` sub-directories will be created automatically by the gSERVER if they do not already exist, so there is no need for any action on behalf by the gBA.

The structure of the gBA user's working directory described above is illustrated schematically in Figure 4. Here, it is supposed that this working directory is called `MyDir`.

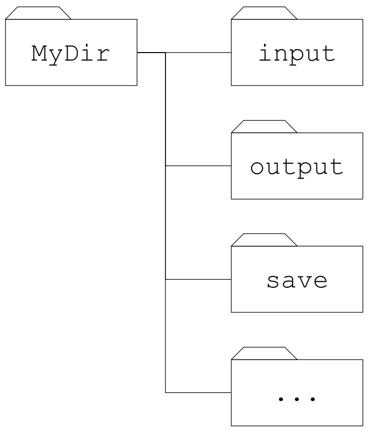

Figure 4: Structure of gBA user's working directory.

3.3.3 gPROMS input file encryption

It is envisaged that the gPROMS input files used by some gBAs may contain significant amounts of intellectual property in terms of, for instance:

- the mathematical modelling of certain processes;
- the values of various parameters and other quantities appearing in these models;
- the manner in which certain problems have been formulated to allow their solution in the form of a gPROMS-based activity.

The gBA developer may wish to protect this intellectual property by encrypting the gPROMS input file(s) being used by the gBA. This is possible using standard gPROMS encryption facilities.

In the case of the gBA using pre-existing gPROMS input files (cf. section 3.3.1),

- The `.gPROMS` file[2] should be provided in encrypted format (and thus will have the `.gENCRYPT` file extension).
- The gBA developer can effect this encryption using the gPROMS `encrypt` command and specifying an appropriate encryption password. In the interests of security, it is recommended that the *short* form of this command (see section 9.3.1 of the "*gPROMS Advanced User Guide*") is used.

[2] Note that the `.gPROMS` files are the only ones that can be encrypted in gPROMS.

- The encrypted gPROMS input file must be loaded into the gSERVER using the `gproms_select` function with the encryption password specified in the argument list (cf. section 7.1).

In the case of the gBA using gPROMS input files created "on-the-fly" (cf. section 3.3.1), the gBA itself must encrypt the input file using the `gproms_encrypt` function provided by the gSERVER (see section 7.2). In the interests of security, this encryption must be performed as soon as the file itself is created and this must be followed immediately by the deletion of the original (unencrypted) file from the disk.

IMPORTANT NOTE

While every effort has been made to ensure the security of the gPROMS encryption facility, PSE Ltd offers no guarantee either explicitly or implicitly regarding its actual performance and security.

3.4 Obtaining results of gPROMS model-based activities

gBAs will normally need to obtain the results of any model-based activities that they execute via the gSERVER. The manner in which this is achieved will depend on the type of activity.

3.4.1 Results of simulation activities

There are two ways in which gBAs may obtain results from simulation activities.

3.4.1.1 Obtaining simulation results via a gPROMS Output Channel

gPROMS uses Output Channels to transmit results of dynamic simulation computations at a fixed frequency (e.g. every *x* simulation time units where *x* is a specified constant), as well as at other important points during the simulation (e.g. after the initialisation of the simulation, and before and after each discontinuity).

The frequency of variable transmission to all Output Channels that are active during a simulation is controlled via the `ReportingInterval` parameter in the `SOLUTIONPARAMETERS` part of the `PROCESS` entity (see section 10.2 of the "*gPROMS Introductory User Guide*").

All transmissions of results normally take place automatically; there is no need for any explicit action to be inserted in the gPROMS input file. However, if desired, results transmission can be switched off or on at will during a simulation using the `MONITOR` elementary task (see section 7.3 of the "*gPROMS Introductory User Guide*").

By default, gPROMS transmits to the Output Channels the values of all the variables in the model being simulated. However, a subset of the variables may also be transmitted by specifying them in the `MONITOR` part of the `PROCESS` entity.

To use this mechanism for obtaining results, the gBA must provide an Output Channel implementation, preferably placing the corresponding `.dll` (or `.so`) file in the `oc` sub-directory of the gPROMS installation directory (cf. section 2.1). Details on how to implement an Output Channel are given in the "*gPROMS System Programmer Guide*".

Once the gBA Output Channel implementation is prepared and installed as described above, the gPROMS input file must specify that this Output Channel be used by setting the value of the `gUSerOutput` parameter in the `SOLUTIONPARAMETERS` part of the `PROCESS` entity (see section 10.2 of the "*gPROMS Introductory User Guide*").

As an example, suppose that:

- The gBA wishes to monitor all model variables whose name contains the string `temp`.
- Monitoring is to be done every 5 simulation units (and at extra-ordinary events such as before and after each discontinuity).
- The gBA Output Channel implementation has been placed in a file called `pse/gPROMS/oc/mygBA_OC.dll.`

Then the PROCESS entity gPROMS input file must be of the form shown in Figure 5.

```
PROCESS xxx
UNIT
    . . . . . .
MONITOR
    *.*temp* ;
ASSIGN
    . . . . . .
INITIAL
    . . . . . .
SOLUTIONPARAMETERS
    gUserOutput        := "mygBA_OC" ;
    ReportingInterval := 5 ;
SCHEDULE
    . . . . . .
```

Figure 5: Example of PROCESS entity making use of Output Channel.

3.4.1.2 Obtaining simulation results via a gPROMS Foreign Process

The use of Foreign Processes allows finer control over both the variables that are obtained by the gBA and the time points at which these variables are obtained. This is achieved by the explicit introduction of `SEND` elementary tasks within the schedule executed by the dynamic simulation.

In general, the gBA must include an implementation of a Foreign Process constructed in the manner described in the "*gPROMS System Programmer Guide*" and placed in the directory `/pse/gPROMS/fpi`. However, a sophisticated implementation of a Foreign Process, `eventFPI.dll`, is already provided as part of the gSERVER's support for asynchronous execution of activities (cf. section 3.2). It is envisaged that the needs of most gBAs will be served by this implementation.

In any case, the Foreign Process to be used must be specified in the gPROMS input file by setting the `FPI` parameter in the SOLUTIONPARAMETERS section of the `PROCESS` entity (see section 5.3 of the "*gPROMS Advanced User Guide*"). This specification takes the form:

```
SOLUTIONPARAMETERS
   FPI := "eventFPI.dll" ;
```

An example of using the FPI for obtaining results from dynamic gPROMS simulations is given in section 14 of this document.

3.4.1.3 Output Channels *vs.* Foreign Processes

Some gBAs will certainly require the finer control on results transmission afforded by the Foreign Process mechanism over the Output Channel one. Moreover, the use of a Foreign Process requires little programming effort if the gBA relies on the sophisticated `eventFPI` already provided as part of the gSERVER. However, for other gBAs, the decision on which mechanism to use for obtaining results from a simulation activity is not an entirely straightforward one.

As a general rule of thumb, if the gBA requires results at a regular frequency, then the use of an Output Channel may well be more efficient than that of a Foreign Process. This is because gPROMS will never take a time integration step that goes past a `SEND` task[3] in a schedule. Hence, the maximum length of time step that will be taken during a simulation cannot possibly exceed the reporting interval. For example, consider the following extract from a gPROMS schedule designed to transmit the value of a variable xxxx at a regular frequency[4]:

```
WHILE TRUE DO
    SEQUENCE
       CONTINUE FOR 10
       SEND
            xxxx
       END
    END
END
```

In this case, the integration time step used internally by gPROMS will never exceed 10 simulation time units. By contrast, no such limitation would exist if the gBA employed instead an Output Channel with a `ReportingInterval` of 10.

[3] Or, indeed, any other elementary task whose timing gPROMS can determine *a priori*.

[4] In a realistic situation, this would probably be executing in parallel with other tasks.

The above consideration may or may not be important depending on the details of the gBA under consideration. For instance, if, in the above example, a `GET` elementary task also had to be executed at a frequency of 10 time units, then every execution of this task would interrupt the simulation anyway; therefore, there would be no additional penalty in using the `SEND` task.

3.4.2 Results of optimisation activities

In order to obtain results of a dynamic or steady-state optimisation activity, a gBA has to read and interpret the various files generated by gPROMS at the end of the execution of the activity. The contents of these files are described in section 2.8 of the "*gPROMS Advanced User Guide*", and their format is illustrated in Appendices A.8-A.11 of the same guide.

All optimisation output files are placed by gPROMS in the output sub-directory of the user's working directory (see section 3.3.2).

3.4.3 Results of parameter estimation activities

In order to obtain results of a parameter estimation activity, a gBA has to read and interpret the various files generated by gPROMS at the end of the execution of the activity. The contents of these files are described in section 3.6 of the "*gPROMS Advanced User Guide*", and their format is illustrated in Appendices C.8-C.9 of the same guide.

All parameter estimation output files are placed by gPROMS in the output sub-directory of the user's working directory (see section 3.3.2).

3.5 gBAs making use of mathematical descriptions of gPROMS activities

An implicit assumption in sections 3.1-3.3.3 has been that the gBAs under consideration use the gSERVER in order to execute a model-based simulation, optimisation or parameter estimation activity. While this will be true for most cases, some gBAs may wish to employ the gSERVER only as a means for describing a process model using the powerful gPROMS modelling language.

Once this is done, the gBAs may wish to extract the underlying mathematical description of the model or of an activity based on it in order to operate on it independently of the gSERVER. This situation may arise when the gBA involves a model-based activity that requires mathematical manipulations beyond those supported directly by gPROMS.

In order to support this type of operation, the gSERVER provides two additional functions:

- Function `gproms_eso` returns an interface to an Equation Set Object (ESO), i.e. a formal software description of the set of differential and algebraic equations that describes a gPROMS `PROCESS` entity. A second ESO describing the set of initial conditions in the `PROCESS` is also returned.
- Function `gproms_gdoo` returns an interface to gPROMS Dynamic Optimisation Object (gDOO), i.e. a formal software description of a dynamic optimisation activity in gPROMS.

A typical usage scenario for a gBA making use of these functions is as follows:

1. Start the gSERVER by invoking function `gproms_start` (see section 6.1).
2. Load into the gSERVER the model of interest in the form of one or more gPROMS input files by invoking function `gproms_select` (see section 7.1).
3. Obtain an interface to an ESO or a gDOO by invoking functions `gproms_eso` (cf. section 8.1) or `gproms_gdoo` (cf. section 8.2) respectively.
4. Perform other (non-gSERVER based) operations using the interfaces obtained at step 3.
5. Stop the gSERVER by invoking function `gproms_end` (see section 6.2).

The complete sets of methods supported by the ESO and gDOO interfaces are described in the "*gPROMS System Programmer Guide*".

3.6 Operating system and compiler requirements for gBA development

System and compiler compatibility is listed in the gPROMS Release Notes.It is worth noting that:

- the operating system requirements are exactly the same as for gPROMS;
- only the compiler version listed explicitly in the gPROMS Release Notes are supported by PSE; earlier or later compiler versions *may* work but are not supported.

3.7 Constructing the gBA executable

It is recommended that the gBA executable employ a dynamic link to the gSERVER software residing in `pse/gPROMS/bin/gSERVER.dll`. Sample makefiles demonstrating how such a link may be created are provided in the directory `pse/gPROMS/src/examples/gSERVER`.

A number of environment variables must be set to appropriate values for the gBA executable to be linked properly. The settings of these environment variables partly depend on the operating system, as shown in Table 1.

Platform	Variable	Value[5]
Windows	`GPROMSHOME`	`pse\gPROMS`
	`PATH`	`pse\gPROMS\bin`
Linux	`GPROMSHOME`	`pse/gPROMS`

Table 1: gSERVER environment variables.

[5] Must be preceded by full pathname of `pse` directory (cf. Figure 1).

4 gBA Installation and Usage

4.1 gSERVER licensing for gBA use

As a minimum a gBA's end-user will need to have a *base* licence of one of the following types:

- A gBA-specific *gSRV_nnnn_Smmmm* licence.

  This licence permits the use of the gSERVER exclusively in the context of a specific gBA with id=*nnnn* for a problem containing ≤ *mmmm* equations.

  Commercial terms for such licences will have to be agreed a priori between the gBA developer and Process Systems Enterprise Ltd.

- A gBA-specific *gSRE_nnnn_Smmmm* licence.

  This licence only allows the selection of encrypted gPROMS input files.

The type of base licence to use must be specified in the gBA's invocation of `gproms_start` (c.f. section 6.1).

N.B. The end-user will also require the appropriate 'model-based activity', 'model-based activity object' and 'component' licenses for the activities performed by the gBA and any licensed components (such as Multiflash) that it uses.

4.2 Installation and set-up of the gBA

For a gBA to be usable, certain environment variables must be set to appropriate values so that shared objects or dynamic link libraries can be located correctly at run-time. These are the environment variables listed in Table 1 in section 3.7.

Part II: Detailed description of gSERVER functions

5 Calling Conventions and Diagnostics for gSERVER Functions

5.1 Calling conventions

The gSERVER is designed to be used primarily by gBAs written in C, C++, or similar procedural languages. The peculiarities that need to be taken into account when using the gSERVER from within FORTRAN-based gBAs are discussed in section 13.

Variables which are specified on exit from the gSERVER functions have to be passed as pointers.

All character strings passed as inputs to the gSERVER functions always have to be terminated with `\0`.

5.2 The `gBAclient` argument

All the gSERVER functions described in the following chapters have a first parameter called `gBAclient`. This is reserved for use of the gSERVER with gBA-specific licences (cf. section 4.1). A `NULL` pointer should be passed in all other cases.

5.3 Diagnostics

All of the gSERVER functions have a final parameter called `status`. On return, this variable contains the execution status of the function. A value of `GPSTAT_OK` indicates that the call has been succesful; all other values denote various types of failure, as shown in Table 2 below.

Status value (declared in `gSERVERstatus.h`)	Description
`GPSTAT_OK`	gSERVER function completed successfully.
`GPSTAT_UNCLASSIFIED_ERROR`	General error. See standard or error output files for more information (see section 6.3).
`GPSTAT_GSERVER_NOT_STARTED`	The function `gproms_start` has not been called prior to a function call or a licence is not available.
`GPSTAT_WRONG_LICENCE_TYPE`	Invalid licence type requested in `gproms_start`. Or attempt to run a gSERVER function for which the licence requested in `gproms_start` is not valid.
`GPSTAT_FILE_ACCESS_ERROR`	An error occurred accessing the specified gPROMS input file in `gproms_select`.
`GPSTAT_TRANSLATION_ERROR`	A syntactical error was detected in a gPROMS input file during `gproms_select`.
`GPSTAT_NO_SUCH_PROCESS`	A `PROCESS` entity with the specified name was not found by `gproms_simulate, gproms_optimise` or `gproms_estimate` (or their asynchronous equivalents).
`GPSTAT_INSTANTIATION_FAILURE`	An error occurred while trying to instantiate a `PROCESS` entity. This is usually caused by mistakes in the gPROMS input file that are not caught during translation (e.g. missing `PARAMETER` specifications).
`GPSTAT_SOLVERCONFIG_FAILURE`	Either a numerical solver (specified in the `PROCESS SOLUTIONPARAMETERS` section) could not be loaded or the configuration specification for a numerical solver is wrong.
`GPSTAT_OPTESTCONFIG_FAILURE`	An optimisation or estimation activity could not start due to missing or invalid `.gOPT`, `.gEST` or `.RUN` files.
`GPSTAT_ACTIVITY_FAILURE`	A failure (usually in a numerical solver) occurred while executing an activity.
`GPSTAT_ESO_BLOCKING`	Illegal attempt to execute any gSERVER action (apart from `gproms_end`) following a call to `gproms_eso`.
`GPSTAT_SERVER_BUSY`	Attempt to execute gSERVER function while gSERVER is busy executing an asynchronous model-based activity.
`GPSTAT_ENCRYPTED_ACCESS`	Attempt to execute a function that is not allowed when working with encrypted gPROMS input.
`GPSTAT_NO_QUEUE`	Attempt to access event when no there is no gSERVER event queue (see section 11.1).

`GPSTAT_NO_EVENT`	No event available in gSERVER event queue within specified timeout period (see section 11.1).
`GPSTAT_INVALID_PRIORITY`	An invalid priority has been specified in call to `gproms_set_activity_priority` (see section 12.3)
`GPSTAT_INVALID_QUEUE_CAPACITY`	An invalid queue capacity been specified in call to `gproms_xxx_mt`.
`GPSTAT_WRONG_EVENT_SEQUENCE`	Unexpected sequence of events detected in call to `gproms_evaluate` (see section 11.4)
`GPSTAT_WRONG_INPUT_SIZE`	Number of inputs requested by gPROMS `GET` task different to value of dimU argument in call to `gproms_evaluate` (see section 11.4)
`GPSTAT_WRONG_OUTPUT_SIZE`	Number of outputs provided by gPROMS `SEND` task different to value of dimY argument in call to `gproms_evaluate` (see section 11.4)
`GPSTAT_NO_STATUS`	The status of the last activity cannot be determined in call to `gproms_get_activity_status`.
`GPSTAT_INVALID_LICENSE`	The licence type requested in `gproms_start` is invalid.
`GPSTAT_INVALID_LICENSE_SUBTYPE`	the licence sub-type requested in `gproms_start` is invalid.
`GPSTAT_SERVER_ALREADY_RUNNING`	Attempt to call `gproms_start` or `gproms_set_mode` after gSERVER has already been started.
`GPSTAT_INTERRUPT_CALLED`	gPROMS stopped due to user-requested interrupt.

Table 2: Status return codes from gSERVER functions.

5.4 gproms_print_error

Functionality: Print a diagnostic message corresponding to a gSERVER status code to the standard output.

Synopsis: void gproms_print_error(errorStatus)

Name of argument	Type	Description	Specified on	
			Entry	Exit
errorStatus	INTEGER	This function prints the diagnostic message for this status code.	yes	no

Header: Declared in `gSERVER.h`

5.5 gproms_get_error

Functionality: Get a diagnostic message corresponding to a gSERVER status code.

Synopsis: void gproms_get_error(errorStatus, error)

Name of argument	Type	Description	Specified on	
			Entry	Exit
errorStatus	INTEGER	This function gets the diagnostic message for this status code.	yes	no
error	CHAR*	The diagnostic message is copied into this string. The gBA should allocate a string at least 256 characters long.	no	yes

Header: Declared in `gSERVER.h`

6 Initialisation and Termination of the Interaction with gSERVER

Two functions are provided for the gBA to start and terminate its interaction with the gSERVER:

- `gproms_start` initialises the gSERVER and secures a licence (see section 6.1).
- `gproms_end` terminates the interaction with the gSERVER and returns a licence (see section 6.2).

In addition to these, function `gproms_redirect_output` (see section 6.3) allows the gBA to redirect any gPROMS output that would normally appear on the console to a set of files. This is necessary for MS Windows-based applications with their own graphical user interfaces; GUI-based applications not using this functionality will experience crashes or other unpredictable behaviour.

Note that if you use `gproms_redirect_output`, it should be called **before** `gproms_start` and **after** `gproms_end`.

6.1 gproms_start

Functionality: Start gPROMS and secure a licence.

Synopsis: void gproms_start(gBAClient, licenceType, licenceMaxSize, status)

Name of argument	Type	Description	Specified on	
			Entry	Exit
gBAClient	void*	gBA client identification interface (see section 5.2)	yes	no
licenceType	CHAR*	Specifies the licence type. All new gBAs should use either “gSRV” or “gSRE”. The `gSRE` license type permits the selection of encrypted models only, i.e. files which have been encrypted using the `gproms_encrypt()` method and containing the extension `.gENCRYPT`.	yes	no
licenceMaxSize	INTEGER	Used in conjunction with gBA specific licences (see section 4.1) to specify the maximum equation limit for the licence to	yes	no

		be used. Valid values range between -1 (meaning an unlimited number of equations) and the theoretical[6] maximum of 2^{31}. The value –1 should be used if not using a gBA specific licence.		
status	INTEGER*	Return status. See section 5.3.	no	yes

Remarks

- This function must always be called *before* any of the other gSERVER functions (with the exception of `gproms_redirect_output`, see section 6.3) can be called. The gBA should check the *status* variable on exit. If it is set to `GPSTAT_OK`, the initialisation was successful; otherwise there was an error.
- You can call the function gproms_set_mode (see section 6.4) before this function to provide additional start-up flags.

Header: Declared in `gSERVER.h`

6.2 gproms_end

Functionality: Finish a gPROMS session and free the licence.

Synopsis: void gproms_end(gBAClient, status)

Name of argument	Type	Description	Specified on	
			Entry	Exit
gBAClient	void*	gBA client identification interface (see section 5.2)	yes	no
status	INTEGER*	Return status. See section 5.3.	no	yes

Remarks

[6] Theoretical because no operating system on which gPROMS is available is capable of supporting the amount of memory required for a 2^{31} equation problem.

- This function terminates a gPROMS session. It is usually called just before the gBA is about to terminate.

Header: Declared in `gSERVER.h`

6.3 gproms_redirect_output

Functionality: Redirects the output from gPROMS to a set of files.

Synopsis: void gproms_redirect_output(filename, status)

Name of argument	Type	Description	Specified on	
			Entry	Exit
filename	CHAR*	Basename for the output files or a NULL pointer for closing the files.	yes	no
status	INTEGER*	Return status. See section 5.3.	no	yes

Remarks

- For console based applications on MS Windows or UNIX platforms, a call to this function is not necessary. For GUI based applications on Windows (e.g. programmed with MFC), it is strictly necessary to use this function because otherwise the application may crash or show other unpredictable behaviour.
- If the argument `filename` is specified on entry, this function will create 3 files containing respectively:
    - standard output (`filename.out`),
    - error output (`filename.err`)
    - output generated by FORTRAN code (`filename.for`).
- This function should be called with a valid filename **before** `gproms_start` is used and with a `NULL` pointer **after** `gproms_end`.

Header: Declared in `gSERVER.h`

6.4 gproms_set_mode

Functionality: Provide additional start-up flags for gSERVER.

Synopsis: void gproms_set_mode(flags, status)

Name of argument	Type	Description	Specified on	
			Entry	Exit

flags	INTEGER	One or more of the following flags:
		
		`GPSTART_PARALLEL`
		Cause gSERVER to request special "_Parallel" licenses.
		
		`GPSTART_QUEUE_FOR_LICENSES`
		Cause gSERVER to queue for licenses if there are none currently available. By default gSERVER will halt and report an error if it cannot get a license.
		
		`GPSTART_NO_CREATE_DIRECTORIES`
		Cause gSERVER to not create the "input", "output", "save" and "code" directories.
			yes	no

status	INTEGER*	Return status. See section 5.3.	no	yes

Remarks

- The `flags` argument is a bitwise OR of the desired flags. e.g. To request "Parallel_" licenses and queue for licenses specify `GPSTART_PARALLEL | GPSTART_QUEUE_FOR_LICENSES`.
- This method must be called before `gproms_start`. If called afterwards it returns `*status == GPSTAT_SERVER_ALREADY_RUNNING`.

Header: Declared in `gSERVER.h`

7 Loading of gPROMS Problem Descriptions

This category includes three distinct functions:

- `gproms_select` causes the translation of a (potentially encrypted) `.gPROMS` input file (see section 7.1).
- `gproms_encrypt` encrypts a `.gPROMS` input file (see section 3.3.3).
- `gproms_decrypt` decrypts an already encrypted input file.

7.1 gproms_select

Functionality: Select a gPROMS input file.

Synopsis: void gproms_select(gBAClient, fileName, password, flag, status)

Name of argument	Type	Description	Specified on	
			Entry	Exit
gBAClient	void*	gBA client identification interface (see section 5.2)	yes	no
filename	CHAR*	Name of the gPROMS input file	yes	no
password	CHAR*	File encryption password. Use a NULL pointer or empty string if the input file is not encrypted.	yes	no
flag	INTEGER	Level of output during input file loading: 0 for silent and 1 for loud	yes	no
status	INTEGER*	Return status. See section 5.3.	no	yes

Remarks

- A gPROMS session must have been already initiated using `gproms_start` for this function to work properly.
- This function is equivalent to the `select` command issued at the gPROMS command line to load a model file. After this step is successfully completed, all other gSERVER functions become available.
- You may omit the "`.gPROMS`" extension from the input *fileName*.
- The password is required when encrypted files are used. Use a `NULL` pointer if you do not want to specify a password.

Header: Declared in `gSERVER.h`

7.2 gproms_encrypt

Functionality: Encrypts a gPROMS input file

Synopsis: gproms_encrypt(gBAClient, filename, password1, password2, status)

Name of argument	Type	Description	Specified on	
			Entry	Exit
gBAClient	void*	gBA client identification interface (see section 5.2)	yes	no
filename	CHAR*	The name of the file to decrypt, without extension	yes	no
password1	CHAR*	The first password for the file	yes	no
password2	CHAR*	The second password for the file. If NULL or the empty string then the created file can **only** be decrypted for internal use by gPROMS; the plain text version cannot be retrieved using the gPROMS decrypt command (or the gproms_decrypt function).	yes	no
status	INTEGER*	Return status. See section 5.3.	no	yes

7.3 gproms_decrypt

Functionality: Decrypts a gPROMS input file

Synopsis: gproms_decrypt(gBAClient, filename, password1, password2, status)

Name of argument	Type	Description	Specified on	
			Entry	Exit
gBAClient	void*	gBA client identification interface (see section 5.2)	yes	no
filename	CHAR*	The name of the file to decrypt, without extension	yes	no
password1	CHAR*	The first password for the file	yes	no

password2	CHAR*	The second password for the file	yes	no
status	INTEGER*	Return status. See section 5.3.	no	yes

8 Obtaining Mathematical Descriptions of gPROMS-based Activities

The two functions in this category return software interfaces to mathematical descriptions of gPROMS-based model s and model-based activities. The interfaces can be accessed by the gBA operating in-process or via CORBA.

Detailed specifications of these interfaces and the functions supported by them are given in the "*gPROMS System Programmer Guide*".

8.1 gproms_eso

Functionality: Returns the mathematical description of a gPROMS PROCESS entity

Synopsis: gproms_eso(gBAClient, process, flag, eso, esoInitialConditions, status)

Name of argument	Type	Description	Specified on	
			Entry	Exit
gBAClient	void*	gBA client identification interface (see section 5.2)	yes	no
process	CHAR*	The name of the `PROCESS` entity for which the ESO is to be built.	yes	no
flag	INTEGER	0 – default; 1 – create separate ESO for the initial conditions, `/S`; 2 – numeric, `/N`; 4 – ignore the ASSIGN section, `/I`.	yes	no
eso	INTEGER*	A pointer to the main ESO	no	yes
icEso	INTEGER*	A pointer to the ESO describing the initial conditions if flag=1 (but also see 3rd remark below); otherwise undefined.	no	yes
status	INTEGER*	Return status. See section 5.3.	no	yes

Remarks

- This function uses a *gESO* licence whilst running.
- A call to this function will block the execution of all other commands except for calls to:

`gproms_end`, `gproms_print_copyright`, `gproms_print_version`, `gproms_get_activity_status`, `gproms_set_activity_priority`, `gproms_retrieve_event`, `gproms_reply_event`, `gproms_kill_activity`, `gproms_get_error`, `gproms_print_error`.

Any attempt to invoke any other gSERVER function will result in a return `status` of `GPSTAT_ESO_BLOCKING`.

- In order to pass combinations of switches in the argument `flag`, a *sum* of the values listed above must be provided. For example, a `flag` with a value of 5 (=1+4) will result in two separate ESOs (one for the main model and one for the initial conditions) in which the ASSIGN section is ignored.
- Two valid pointers (`eso` and `esoInitialConditions`) must always be passed to this function even if a separate ESO for the initial conditions is not requested.

Header: Declared in `gSERVER.h`

8.2 gproms_gdoo

Functionality: Returns a mathematical description of a gPROMS dynamic optimisation activity.

Synopsis: gproms_gdoo(gBAClient, process, gdoo, status)

Name of argument	Type	Description	Specified on	
			Entry	Exit
gBAClient	void*	gBA client identification interface (see section 5.2)	yes	no
process	CHAR*	The name of the process for which the gDOO is to be built.	yes	no
gdoo	INTEGER*	Stores a pointer to the gDOO	no	yes
status	INTEGER*	Return status. See section 5.3.	no	yes

Remarks

- This function uses a *gDOO* licence whilst running.
- A call to this function will block the execution of all other commands except for calls to `gproms_end`. Any attempt to invoke any other gSERVER function will result in a return `status` of `GPSTAT_ESO_BLOCKING`.

- A valid pointer must always be passed for argument `gdoo`. A `NULL` value is not allowed.

Header: Declared in `gSERVER.h`

9 Synchronous Execution of gPROMS Model-based Activities

This category includes three major functions:

- `gproms_simulate` executes a dynamic simulation of a specified `PROCESS` entity, the name of which is specified as a string input argument in the call to the procedure (see section 9.1).
- `gproms_optimise` executes a dynamic optimisation activity based on a specified `PROCESS` entity, the name of which is specified as a string input argument in the call to the procedure; a `.gOPT` input file with the same name must also exist for this to be successful (see section 9.2).
- `gproms_estimate` executes a parameter estimation activity based on a specified `PROCESS` entity, the name of which is specified as a string input argument in the call to the procedure; a `.gEST` input file with the same name must also exist for this to be successful (see section 9.3).

The above functions execute in a synchronous manner, i.e. control is returned to the calling program only after their execution has been completed (e.g. in the case of `gproms_simulate`, after the end of the dynamic simulation). If the application program desires to have some interaction with the gPROMS calculation *while* the latter is still executing, then the procedures described in section 10 should be called instead. These issues are discussed in detail in section 3.2 of this document.

9.1 gproms_simulate

Functionality: Perform a simulation based on a specified gPROMS `PROCESS` entity.

Synopsis: void gproms_simulate(gBAClient, processName, outputLevel, status)

Name of argument	Type	Description	Specified on	
			Entry	Exit
gBAClient	void*	gBA client identification interface (see section 5.2)	yes	no
processName	CHAR*	Name of the `PROCESS` entity to simulate	yes	no
outputLevel	INTEGER	0 for silent (/S), 1 for quiet (/Q), 2 for the default level, 3 for loud (/L) and 9 for diagnose (/D)	yes	no
status	INTEGER*	Return status. See section 5.3.	no	yes

Remarks

- This function uses a *gSIM* licence whilst running.

- This function is equivalent to the `simulate` command issued in a gPROMS command line to simulate a process. A successful call to `gproms_select` function is a pre-requisite of calling this function.
- If there is any problem with the simulation, the function returns immediately with a *status* indicating the error (e.g. `PROCESS` entity does not exist due to a failure of `gproms_select` caused by syntax errors in the input file).
- Once a simulation has begun successfully, this function does not return until the simulation is over. Application programs should use the gproms_simulate_mt function to start a non-blocking simulation asynchronously.

Header: Declared in `gSERVER.h`

9.2 gproms_optimise

Functionality: Perform an optimisation based on a specified gPROMS `PROCESS` entity.

Synopsis: void gproms_optimise(gBAClient, processName, status)

Name of argument	Type	Description	Specified on	
			Entry	Exit
gBAClient	void*	gBA client identification interface (see section 5.2)	yes	no
processName	CHAR*	Name of the process to optimise	yes	no
status	INTEGER*	Return status. See section 5.3.	no	yes

Remarks

- This function uses a *gOPT* licence whilst running.
- This function is equivalent to the `optimise` command issued in a gPROMS command line to simulate a process. A successful call to `gproms_select` function is a pre-requisite of calling this function.
- A file called `processName.gOPT` must exist in the input sub-directory of the gBA user's working directory (cf. section 3.3.2) for this function to be successful.
- Once an optimisation has begun successfully, this function does not return until the optimisation is over. Application programs should use the `gproms_optimise_mt` function to start a non-blocking optimisation asynchronously.

Header: Declared in `gSERVER.h`

9.3 gproms_estimate

Functionality: Estimate model parameters based on a specified gPROMS `PROCESS` entity.

Synopsis: void gproms_estimate(gBAClient, processName, status)

Name of argument	Type	Description	Specified on	
			Entry	Exit
gBAClient	void*	gBA client identification interface (see section 5.2)	yes	no
processName	CHAR*	Name of the process to estimate	yes	no
status	INTEGER*	Return status. See section 5.3.	no	yes

Remarks

- This function uses a *gEST* licence whilst running.
- This function is equivalent to the `estimate` command issued in a gPROMS command line to perform a process estimation. A successful call to `gproms_select` function is a pre-requisite of calling this function.
- A file called `processName.gEST` must exist in the input sub-directory of the gBA user's working directory (cf. section 3.3.2) for this function to be successful. Any experimental results (.`RUN`) files referenced by the `processName.gEST` file must also exist in the same sub-directory.
- Once an estimation has begun successfully, this function does not return until the estimation is over. Clients should use the `gproms_estimate_mt` function to start a non-blocking estimation asynchronously.

Header: Declared in `gSERVER.h`

10 Asynchronous Execution of gPROMS Model-based Computations

This category includes three major functions:

- `gproms_simulate_mt` executes a dynamic simulation of a specified `PROCESS` entity, the name of which is specified as a string input argument in the call to the procedure (see section 10.1).
- `gproms_optimise_mt` executes a dynamic optimisation activity based on a specified `PROCESS` entity, the name of which is specified as a string input argument in the call to the procedure; a `.gOPT` input file with the same name must also exist for this to be successful (see section 10.2).
- `gproms_estimate_mt` executes a parameter estimation activity based on a specified `PROCESS` entity, the name of which is specified as a string input argument in the call to the procedure; a `.gEST` input file with the same name must also exist for this to be successful (see section 10.3).

The above functions are the exact equivalents to those described in section 9 except that they execute in an asynchronous manner, thereby affording the gBA significantly more flexibility (cf. section 3.2 of this document).

10.1 gproms_simulate_mt

Functionality: Perform a simulation based on a specified gPROMS `PROCESS` entity in an asynchronous mode.

Synopsis: void gproms_simulate_mt(gBAClient, processName, outputLevel, queueCapacity, status)

Name of argument	Type	Description	Specified on	
			Entry	Exit
gBAClient	void*	gBA client identification interface (see section 5.2)	yes	no
processName	CHAR*	Name of the process to simulate	yes	no
outputLevel	INTEGER	Level of output: 0 for silent (/S), 1 for quiet (/Q), 2 for the default level, 3 for loud (/L) and 9 for diagnose (/D)	yes	no
queueCapacity	INTEGER	Size of event queue to use (see section 11.1). 0 means no queue, < 0 means use the default queue size.	yes	no
status	INTEGER*	Return status. See section 5.3.	no	yes

Remarks

- A *gSIM* license is used whilst the simulation is executing.
- This function is similar to `gproms_simulate` (cf. section 9.1), simulating the process *processName*. However, `gproms_simulate_mt` returns immediately to the client (caller) program, with the simulation executing asynchronously, leaving the main program free to perform other tasks, e.g. keep a Graphical User Interface (GUI) responsive.
- Once a background simulation is initiated, the main application can interact with it using the functions described in sections 11 and 12.
- While the gSERVER is busy performing a background simulation, it will not accept requests for most other operations, e.g. another parallel simulation of another process, an optimisation etc. Use `gproms_get_activity_status` to query the status of the gSERVER.

Header: Declared in `gSERVER.h`

10.2 gproms_optimise_mt

Functionality: Perform an optimisation based on a specified gPROMS `PROCESS` entity in an asynchronous mode.

Synopsis: void gproms_optimise_mt(gBAClient, processName, queueCapacity, status)

Name of argument			Specified on	
	Type	Description	Entry	Exit
gBAClient	void*	gBA client identification interface (see section 5.2)	yes	no
processName	CHAR*	Name of the process to optimise	yes	no
queueCapacity	INTEGER	Size of event queue to use (see section 11.1). 0 means no queue, < 0 means use the default queue size.	yes	no
status	INTEGER*	Return status. See section 5.3.	no	yes

Remarks

- A *gOPT* license is used whilst the optimisation is executing.
- This function is similar to `gproms_optimise` (cf. section 9.2), performing an optimisation of the process *processName*. However, `gproms_optimise_mt` returns immediately to the client (caller) program. gPROMS then proceeds to perform the optimisation asynchronously, leaving the main program free to perform other tasks, e.g. keep a Graphical User Interface (GUI) responsive.

- Once a background optimisation is initiated, the main application can interact with it using the functions described in sections 11 and 12.
- While the gSERVER is busy performing a background optimisation, it will not accept requests for most other operations, e.g. another parallel simulation of another process, an optimisation etc. Use `gproms_get_activity_status` to query the status of the gSERVER.

Header: Declared in `gSERVER.h`

10.3 gproms_estimate_mt

Functionality: Estimate model parameters based on a specified gPROMS `PROCESS` entity in an asynchronous mode.

Synopsis: void gproms_estimate_mt(gBAClient, processName, queueCapacity, status)

Name of argument	Type	Description	Specified on	
			Entry	Exit
gBAClient	void*	gBA client identification interface (see section 5.2)	yes	no
processName	CHAR*	Name of the process to estimate	yes	no
queueCapacity	INTEGER	Size of event queue to use (see section 11.1). 0 means no queue, < 0 means use the default queue size.	yes	no
status	INTEGER*	Return status. See section 5.3.	no	yes

Remarks

- A *gEST* license is used whilst the estimation is executing.
- This function is similar to `gproms_estimate` (cf. section 9.3), performing a parameter estimation for the process *processName*. However, `gproms_estimate_mt` returns immediately to the client (caller) program. gPROMS then proceeds to perform the estimation asynchronously, leaving the main program free to perform other tasks, e.g. keep a Graphical User Interface (GUI) responsive.
- Once a background estimation is initiated, the main application can interact with it using the functions described in sections 11 and 12.
- While the gSERVER is busy performing a background parameter estimation, it will not accept requests for most other operations, e.g. another parallel simulation of another process, an optimisation etc. Use `gproms_get_activity_status` to query the status of the gSERVER.

Header: Declared in `gSERVER.h`

11 Exchanging Information with gPROMS Activities Executing Asynchronously

The section describes the event-based mechanism that allows a gBA to communicate with a gPROMS activity (simulation, optimisation or parameter estimation) run while the latter is executing asynchronously.

Asynchronously executing activities will have been initiated by calling the corresponding `gproms_xxx_mt` function (where *xxx* is `simulate`, `optimise` or `estimate`, see section 10). It will be recalled that this call immediately returns to the gBA, while gPROMS proceeds to execute the requested activity. The gBA is then free to perform other tasks, and/or to monitor the activity progress at the same time. It can also interact with the activity using one or both of two different mechanisms:

- The `eventFPI` mechanism (see section 11.5)

  This is based on a full implementation of a gPROMS Foreign Process Interface (see chapter 5 of the "*gPROMS Advanced User Guide*"). This kind of interaction is currently possible only for gPROMS simulation activities.

- The `simpleEventFOI` mechanism (see section 11.6)

  This is based on a restricted implementation of a gPROMS Foreign Object Interface (see chapter 4 of the "*gPROMS Advanced User Guide*"). This kind of interaction is possible for all types of model-based activities.

Both of the above are based on events that are being notified by the executing gPROMS activity. The basic concepts underlying these event-based mechanisms are described in section 11.1. Sections 11.2 and 11.3 then describe the functions that are provided by the gSERVER for gBAs to, respectively, retrieve and respond to these events. Finally, section 11.4 describes `gproms_evaluate`, a function that is designed for the handling of the common GET-SEND sequence of events.

11.1 Events and the gSERVER event queue

The communication between the gBA and the executing gPROMS activity is effected though the use of *events*. Basically, gPROMS posts events that the gBA, first retrieves and then replies to.

11.1.1 Management of the event queue

Events posted by gPROMS are stored in the gSERVER's *event queue*. A gBA can configure the size of this queue using the `queueCapacity` parameter to the `gproms_xxx_mt` functions (see sections 10.1, 10.2 and 10.3). For gSERVER 2.2, we recommend you use either a value of –1 (default queue size) or 0 (no queue). You should use `queueCapacity == 0` if you do not wish to handle events.

A gBA can retrieve the event at the front of the queue by calling `gproms_retrieve_event`; note that **this does not remove the event from the queue**.

A gBA can reply to the event at the front of the queue by calling `gproms_reply_event`; this removes the event from the front of the queue.

If gPROMS tries to post an event when the queue is full, then the asynchronous activity stops and waits until the gBA replies to the event at the front of the queue (and thus causes it to be removed from the queue).

11.1.2 Blocking events

Some events are described as *blocking*; these events cause the asynchronous activity to stop and wait for a reply even if the queue is not full. Blocking events are typically those whose response may affect the future path of the gPROMS activity (e.g. via the provision of the value of a variable that forms an input to a gPROMS simulation activity, or by setting an appropriate status flag).

If your model produces blocking events, then it will not execute if you use `queueCapacity == 0`.

11.1.3 Event types generated by the gSERVER

A complete list of the different types of event that are generated by the gSERVER is given in Table 3. There are essentially three distinct categories of such events:

- Events providing general information as to the current status of the gSERVER and the progress of an activity.

  These are all non-blocking (cf. section 11.1.2) and include `GPE_START`, `GPE_SIMULATION_START`, `GPE_OPTIMISATION_START`, `GPE_ESTIMATION_START`, `GPE_ACTIVITY_FINISH` and `GPE_ACTIVITY_INTERRUPT` events.

- Events generated by the execution of FPI tasks in the `SCHEDULE` of a gPROMS simulation activity (see section 11.5).

  These blocking events include `GPE_FPI_PAUSE`, `GPE_FPI_GET`, `GPE_FPI_SEND`, `GPE_SENDMATHINFO` and `GPE_LINEARISE`.

- Events generated by Foreign Object methods (see section 11.6).

  At present, this includes only the blocking event `GPE_FOI_GET`.

Event constant (defined in `gSERVERevents.h`)	Description	Blocking	Type of `m_Data` (see section 11.1.4)
`GPE_START`	First event posted to the queue after it has been initialised	No	`NULL`
`GPE_SIMULATION_START`	Simulation activity starts	No	`NULL`
`GPE_OPTIMISATION_START`	Optimisation activity starts	No	`NULL`
`GPE_ESTIMATION_START`	Estimation activity starts	No	`NULL`

`GPE_ACTIVITY_FINISH`	Activity ends naturally	No	`GPFinishData*` (see section 11.1.5)
`GPE_ACTIVITY_INTERRUPT`	Activity ends prematurely due to an interrupt (such as Ctrl-C)	No	`GPFinishData*` (see section 11.1.5)
`GPE_FPI_PAUSE`	`eventFPI PAUSE` task performed	Yes	`GPFPIPauseData*` (see section 11.5.1)
`GPE_FPI_GET`	`eventFPI GET` task performed	Yes	`GPFPIGetSendData*` (see section 11.5.2)
`GPE_FPI_SEND`	`eventFPI SEND` task performed	Yes	`GPFPIGetSendData*` (see section 11.5.2)
`GPE_FPI_SENDMATHINFO`	`eventFPI SENDMATHINFO` task performed	Yes	`GPFPISendMathInfoData*` (see section 11.5.3)
`GPE_FPI_LINEARISE`	`eventFPI LINEARISE` task performed	Yes	`GPFPILineariseData*` (see section 11.5.4)
`GPE_FOI_GET`	`simpleEventFOI` method called	Yes	`GPFPIGetSendData*` (see sections 11.6 and 11.5.2)

Table 3: Event types generated by gSERVER.

11.1.4 Data associated with a gSERVER event

The data for an event is contained within a `GPEvent` structure (header `gSERVERevents.h`):

Member variable	Description
INTEGER *m_EventId*	Identifies the type of the event.
void* *m_Data*	Pointer to additional data for the event. The form and content of these data will differ depending on the type of event, as indicated in the last column of Table 3.

11.1.5 The `GPE_ACTIVITY_FINISH` and `GPE_ACTIVITY_INTERRUPT` events

A `GPE_ACTIVITY_FINISH` event is posted when an asynchronous gPROMS activity ends naturally.

A `GPE_ACTIVITY_INTERRUPT` event is posted when an asynchronous gPROMS activity ends due to an interrupt; for example the user presses Ctrl-C in a console gBA.

Having retrieved a `GPE_ACTIVITY_FINISH` or `GPE_ACTIVITY_INTERRUPT` event via a call to the `gproms_retrieve_event` function (cf. section 11.2), the gBA can cast the `m_Data` pointer in the `GPEvent` structure (cf. section 11.1.4) to a `GPFinishData *` pointer.

The `GPFinishData` structure (declared in header `gSERVERevents.h`) has the following format:

Member variable	Description
INTEGER *m_ExitStatus*	The gSERVER status when the activity finished or was terminated. (see section 5.3)

11.2 gproms_retrieve_event

Functionality: Retrieve the event at the front of the event queue.

Synopsis: void gproms_retrieve_event(gBAClient, timeout, ppEvent, status)

Name of argument	Type	Description	Specified on	
			Entry	Exit
gBAClient	void*	gBA client identification interface (see section 5.2)	yes	no
timeout	INTEGER	If the event queue is empty when this function is called, then this is the time (in milliseconds) that it should wait for an event before returning. • timeout = 0 means that it will return immediately if there is no event • timeout < 0 means that it will wait indefinitely for an event.	yes	no
ppEvent	GPEvent**	On entry, this must be the address of a pointer to a GPEvent structure. On exit, the pointer will point to a GPEvent structure associated with the retrieved event. The data is managed by gPROMS and should only be modified as specified for particular events. You should **never** call `free` or `delete` on it.	no	yes
status	INTEGER*	Return status. See section 5.3.	no	yes

Remarks

- This function is valid only for activities started asynchronously via `gproms_xxx_mt`, with a non-zero value specification for argument `queueCapacity`. Otherwise it returns with `*status == GPSTAT_NO_QUEUE`.
- `gproms_retrieve_event` **does not remove the event from the front of the event queue**[7]. You need to call `gproms_reply_event` to do this.

Header: Declared in `gSERVER.h`

11.3 gproms_reply_event

Functionality: "Reply" to the event at the front of the event queue.

Synopsis: void gproms_reply_event(gBAClient, status)

Name of argument	Type	Description	Specified on	
			Entry	Exit
gBAClient	void*	gBA client identification interface (see section 5.2)	yes	no
status	INTEGER*	Return status. See section 5.3.	no	yes

Remarks

- This function is valid only for activities started asynchronously via `gproms_xxx_mt`, with a non-zero value specification for argument `queueCapacity`. Otherwise it returns with `*status == GPSTAT_NO_QUEUE`.
- gBAs should call this function to tell gPROMS that they are finished with the event at the front of the event queue.
- After a call to `gproms_reply_event` the `GPEvent` structure returned by the previous call to `gproms_retrieve_event` is **no longer valid** and gPROMS will free up the associated memory. If you need to record this data you should make a separate copy.
- The correct method to call `gproms_retrieve_event` and `gproms_reply_event` is illustrated below:

```
INTEGER status;
GPEvent* pEvent;

/*
 * For this example, call gproms_retrieve_event() with a NULL gBAClient and
 * 0 timeout.
```

[7] For example, two consecutive calls to `gproms_retrieve_event` will return the *same* event.

```
         */
        gproms_retrieve_event(NULL, 0, &pEvent, &status);

        /*
         * status should now be GPSTAT_OK, GPSTAT_NO_EVENT or GPSTAT_NO_QUEUE.
         */
        if (status == GPSTAT_OK)
        {
            /*
             * Process the event.
             *
             * pEvent->m_EventId is GPE_START, GPE_FINISH, GPE_FPI_GET, ...
             * pEvent->m_Data    is NULL or a pointer to additional event specific
             *                   data.
             *
             * N.B. The data pointed at by pEvent and pEvent->m_Data is internal
             *      gPROMS data. You should not 'free' or 'delete' it and should
             *      only modify it as specified for particular events.
             */

            gproms_reply_event(NULL, &status);

            /*
             * After the call to gproms_reply_event(), the GPEvent pointed at by
             * pEvent is no longer valid!
             */
        }
```

Header: Declared in `gSERVER.h`

11.4 gproms_evaluate

Functionality: Treat an entire gPROMS simulation as a function evaluation of the form *y=f(u)*.

Synopsis: void gproms_evaluate(gBAClient, u, dimU, y, dimY, status)

Name of argument	Type	Description	Specified on	
			Entry	Exit
gBAClient	void*	gBA client identification interface (see section 5.2)	yes	no
u	DOUBLE*	Array of input variables	yes	no
dimU	INTEGER	Dimensionality of input vector *u*. If < 0 then a failure `STATUS` will be passed to gPROMS at the next `GPE_FPI_GET` event. If 0 then no `GPE_FPI_GET` event will be expected.	yes	no
Y	DOUBLE*	Array of output variables, calculated by gPROMS.	no	yes

dimY	INTEGER	Dimensionality of output vector *y*. If < 0 then a failure `STATUS` will be passed to gPROMS at the next `GPE_FPI_SEND` event. If 0 then no `GPE_FPI_SEND` event will be expected.	yes	no
status	INTEGER*	Return status. See section 5.3.	no	yes

Remarks

- This function is valid only for gPROMS `PROCESS` entities that fulfil all three of the following requirements:
    - are executed asynchronously via `gproms_simulate_mt`, **and**
    - specify the `eventFPI` in their `SOLUTIONPARAMETERS` section, **and**
    - contain a sequence of
        - a `GET` task to receive the required inputs *u* from the gBA
        - a `SEND` task to transmit the required outputs *y* to the gBA.
- The invocation of the `gproms_evaluate` function by a gBA handles two FPI events, a `GPE_FPI_GET` followed by a `GPE_FPI_SEND`, occurring consecutively in the simulation schedule, possibly separated by a certain period of undisturbed integration and/or other non-FPI tasks. An example of such a schedule is shown in Figure 6.

```
StatFlag := TRUE;
WHILE StatFlag DO
   SEQUENCE
      GET
         SIGNALID "GETInfo"
         STATUS StatFlag
         simInterval ;
        . . . . . . other input variables u
      END

      . . . . . . perform any other valid tasks
      CONTINUE FOR simInterval

      SEND
         SIGNALID "SENDInfo"
         STATUS StatFlag
        . . . . . . transmit output variables y
      END
   END
```

Figure 6: Example of gPROMS SCHEDULE for use with the `gproms_evaluate` function.

Note that, in this example:

- The entire sequence is enclosed within a `WHILE` loop; this allows gPROMS to act as a function evaluator for repeated calls to `gproms_evaluate` by the gBA.
- The boolean variable `StatFlag` is actually the status of the `GET` and `SEND` elementary tasks. The gBA will set this to `FALSE` if the *dimU* and/or the *dimY* arguments is < 0 on entry to `gproms_evaluate`.
- The length of the simulation interval, `simInterval`, separating the `GET` and `SEND` events happens to be also obtained from the gBA via the FPI. This is a typical occurrence in problems where the evaluation of the function *y*=*f*(*u*) actually involves a dynamic simulation. However, `simInterval` could just as easily be a hardcoded constant.

- The dimensionalities of the *u* and *y* vectors (given by arguments *dimU* and *dimY*, respectively) should be consistent with the number of variables in the `GET` and `SEND` tasks, otherwise errors `GPSTAT_WRONG_INPUT_SIZE` or `GPSTAT_WRONG_OUTPUT_SIZE` are returned by `gproms_evaluate` (cf. section 5.3).
- The following example illustrates the correspondence of gPROMS tasks and the arguments to `gproms_evaluate`:

gPROMS task	Associated input/output vectors

`GET`
`  SIGNALID "GET"`
`  STATUS StatFlag`
`  SimInterval ;`
`  Tank.Fin(1) ;`
`END`
`CONTINUE FOR simInterval`
`GET`
`  SIGNALID "SEND"`
`  STATUS StatFlag`
`  Tank.V ;`
`  Tank.C(1) ;`
`END`
	`DOUBLE u[2] = {1.0, 0.5}; /* input */`
	`DOUBLE y[2]; /* output */`
	`INTEGER dimU = 2, dimY = 2;`
	`INTEGER status;`
	
	`gproms_evaluate(u, dimU, y, dimY, &status);`
	
	`// correspondence of variables`
	`// u[0] -> SimInterval`
	`// u[1] -> Tank.Fin(1)`
	`// y[0] -> Tank.V`
	`// y[1] -> Tank.C(1)`

- `gproms_evaluate` internally utilises functions `gproms_retrieve_event` and `gproms_reply_event` to retrieve the two consecutive `GET/SEND` events. If these events are not encountered in the expected order, `gproms_evaluate` returns a `GPSTAT_WRONG_EVENT_SEQUENCE` error (cf. section 5.3).

- gproms_evaluate interprets dimU < 0 and/or dimY < 0 to have a special meaning. If dimU < 0 then a failure STATUS (0) is passed to gPROMS on the next GET event. If dimY < 0 then a failure STATUS (0) is passed to gPROMS on the next SEND event.

  Note that neither of these is guaranteed to stop a simulation, they require the gPROMS model to correctly handle the STATUS of the GET and SEND tasks and stop when it equals 0. Even then the simulation may not stop **immediately** and you may find if you call another gSERVER function without waiting it reports GPSTAT_SERVER_BUSY. To determine when the simulation really has stopped use gproms_get_activity_status to monitor the status of gSERVER until it reports GPRUN_STOPPED.

- gproms_evaluate interprets dimU = 0 and/or dimY = 0 to have a special meaning. If dimU = 0 (no inputs) then this function just expects a SEND event. If dimY = 0 (no outputs) then this function just expects a GET event. If dimU = 0 and dimY = 0 then this function does nothing!

Header: Declared in gSERVER.h

11.5 Using the **eventFPI**

The eventFPI is an implementation of the gPROMS Foreign Process Interface (FPI) provided as standard with the gSERVER software. If a gPROMS PROCESS entity uses the eventFPI, each invocation of an elementary FPI task (PAUSE, GET, SEND, SENDMATHINFO and LINEARISE) posts a corresponding blocking event into the event queue.

To use the eventFPI, the PROCESS entity must specify it in its SOLUTIONPARAMETERS section:

```
SOLUTIONPARAMETERS
    FPI := “eventFPI”;
```

It should be noted that:

- The eventFPI can only be used for asynchronously executing simulation activities initiated by a call to the gproms_simulate_mt function (cf. section 10.1).
- Because all events posted by the eventFPI are blocking events (cf. section 11.1.2), gproms_simulate_mt must be called with a non-zero value for its queueCapacity argument.
- The eventFPI cannot be used for optimisation or parameter estimation activities.

11.5.1 The **GPE_FPI_PAUSE** event

A GPE_FPI_PAUSE event is posted by the eventFPI in response to a PAUSE elementary task (see section 5.2 of the “*gPROMS Advanced User Guide*”).

Having retrieved a GPE_FPI_PAUSE event via a call to the gproms_retrieve_event function (cf. section 11.2), the gBA can cast the m_Data pointer in the GPEvent structure (cf. section 11.1.4) to a GPFPIPauseData* pointer.

The `GPFPIPauseData` structure (declared in header `gSERVERevents.h`) has the following format:

Member variable	Description
CHAR* *m_Signal*	The `SIGNALID` of the `PAUSE` task.
DOUBLE *m_Time*	The simulation time at which the `PAUSE` task was performed.
INTEGER* *m_Status*	Pointer to an INTEGER corresponding to the `STATUS` of the `PAUSE` task; this value can be changed by the gBA before its next call to `gproms_reply_event` as follows: 0 = Failure, 1 = Success (this is the value the event has when it is retrieved).

11.5.2 The `GPE_FPI_GET` and `GPE_FPI_SEND` events

A `GPE_FPI_GET` event is posted by the `eventFPI` in response to a `GET` elementary task (see section 5.2 of the "*gPROMS Advanced User Guide*").

A `GPE_FPI_SEND` event is posted by the `eventFPI` in response to a `SEND` elementary task (see section 5.2.3 of the "*gPROMS Advanced User Guide*").

Having retrieved either a `GPE_FPI_GET` or a `GPE_FPI_SEND` event via a call to the `gproms_retrieve_event` function (cf. section 11.2), the gBA can cast the `m_Data` pointer in the `GPEvent` structure (cf. section 11.1.4) to a `GPFPIGetSendData*` pointer.

The `GPFPIGetSendData` structure (declared in header `gSERVERevents.h`) has the following format:

Member variable	Description
CHAR* *m_Signal*	The `SIGNALID` of the `GET`/`SEND` task.
DOUBLE *m_Time*	The simulation time at which the `GET`/`SEND` task was performed.
INTEGER* *m_Status*	Pointer to an INTEGER corresponding to the `STATUS` of the `GET`/`SEND` task; this value can be changed by the gBA before its next call to `gproms_reply_event` as follows: 0 = Failure, 1 = Success (this is the value the event has when it is retrieved).
INTEGER *m_NumVars*	The number of variables being got or sent.
GSTRING* *m_VarNames*	Array of *m_NumVars* GSTRINGs (strings up to 256 characters long) containing to the names of the quantities involved in the `GET`/ `SEND` task as specified by their *ForeignProcessIDs* (tag-

	name).
INTEGER* *m_VarTypes*	Array of *m_NumVars* INTEGERs corresponding to the types of the variables: 1 = REAL 2 = INTEGER 3 = LOGICAL.
DOUBLE* *m_VarValues*	Array of *m_NumVars* DOUBLEs corresponding to the current value of the variables. In the case of a `GPE_FPI_GET` event, these values can be changed by the gBA before its next call to `gproms_reply_event`; this mechanism allows the gBA to pass back to gPROMS the information requested by it via the `GET` task. In the case of `GPE_FPI_SEND` event, any such changes will be ignored by gPROMS.

11.5.3 The `GPE_FPI_SENDMATHINFO` event

A `GPE_FPI_SENDMATHINFO` event is posted by the `eventFPI` in response to a `SENDMATHINFO` elementary task (see section 5.2 of the "*gPROMS Advanced User Guide*").

Having retrieved a `GPE_FPI_SENDMATHINFO` event via a call to the `gproms_retrieve_event` function (cf. section 11.2), the gBA can cast the `m_Data` pointer in the `GPEvent` structure (cf. section 11.1.4) to a `GPFPISendMathInfoData *` pointer.

The `GPFPISendMathInfoData` structure (declared in header `gSERVERevents.h`) has the following format:

Member variable	Description
CHAR* *m_Signal*	The `SIGNALID` of the `SENDMATHINFO` task.
DOUBLE *m_Time*	The simulation time at which the `SENDMATHINFO` task was performed.
INTEGER* *m_Status*	Pointer to an INTEGER corresponding to the `STATUS` of the `SENDMATHINFO` task; this value can be changed by the gBA before its next call to `gproms_reply_event` as follows: 0 = Failure, 1 = Success (this is the value the event has when it is retrieved).
INTEGER *m_NumVars*	Total number of variables in the mathematical model.
INTEGER *m_NumEqs*	Total number of equations in the mathematical model.

INTEGER *m_NumDVars*	Number of differential variables in the mathematical model.
INTEGER *m_NumNonZeroes*	Number of non-zero Jacobian elements in the mathematical model.
DOUBLE* *m_VarValues*	Array of *m_NumVars* values (DOUBLEs) corresponding to the current value of all the variables in the mathematical model.
DOUBLE* *m_VarDerivValues*	Array of *m_NumVars* values (DOUBLEs) corresponding to the current value of the time derivatives of all the variables in the mathematical model.
INTEGER* *m_VarTypes*	Array of *m_NumVars* INTEGERs corresponding to the type of all the variables in the current mathematical model: 0 = Input. 1 = Algebraic. 2 = Differential.
GSTRING* *m_VarNames*	Array of *m_NumVars* GSTRINGs (strings up to 256 characters long) corresponding to the gPROMS pathnames of all the variables in the mathematical model.
INTEGER* *m_JacRows*	Array of *m_NumNonZeroes* INTEGERs corresponding to the row numbers of the non-zero elements in the current Jacobian matrix.
INTEGER* *m_JacCols*	Array of *m_NumNonZeroes* INTEGERs corresponding to the column numbers of the non-zero elements in the current Jacobian matrix.
DOUBLE* *m_JacValues*	Array of *m_NumNonZeroes* DOUBLEs corresponding to the values of the non-zero elements in the current Jacobian matrix.

11.5.4 The `GPE_FPI_LINEARISE` event

A `GPE_FPI_LINEARISE` event is posted by the `eventFPI` in response to a `LINEARISE` elementary task (see section 5.2 of the "*gPROMS Advanced User Guide*").

Having retrieved an `GPE_FPI_LINEARISE` event via a call to the `gproms_retrieve_event` function (cf. section 11.2), the gBA can cast the `m_Data` pointer in the `GPEvent` structure (cf. section 11.1.4) to a `GPFPILineariseData *` pointer.

The `GPFPILineariseData` structure (declared in header `gSERVERevents.h`) has the following format:

Member variable	Description
CHAR* *m_Signal*	The `SIGNALID` of the `LINEARISE` task.
DOUBLE *m_Time*	The simulation time at which the `LINEARISE` task was performed.
INTEGER* *m_Status*	Pointer to an INTEGER corresponding to the `STATUS` of the `LINEARISE` task: 0 = Non-existent Jacobian element. 1 = Success. -1 = Structurally singular DAE system. -2 = Failure of linear system solver. This value can be changed by the gBA before its next call to `gproms_reply_event` as follows: 0 = Failure, 1 = Success.
INTEGER *m_NumU*	Number of independent input variables specified in the LINEARISE task.
INTEGER* *m_UIndices*	Array of *m_NumU* INTEGERs corresponding to the indexes of the independent input variables in the global variable array.
GSTRING* *m_UNames*	Array of *m_NumU* GSTRINGs (strings up to 256 characters long) corresponding to the names of the independent input variables as specified by their *ForeignProcessID* (tag-name) in the LINEARISE task.
INTEGER *m_NumY*	Number of output variables specified in the LINEARISE task.
INTEGER* *m_YIndices*	Array of *m_NumY* INTEGERs corresponding to the indexes of the output variables in the global variable array.
GSTRING* *m_YNames*	Array of *m_NumY* GSTRINGs (strings up to 256 characters long) corresponding to the names of the output variables as specified by their *ForeignProcessID* (tag-name) in the LINEARISE task.
INTEGER *m_NumX*	Number of the minimal subset of state variables.
INTEGER* *m_XIndices*	Array of *NumX* INTEGERs corresponding to the indexes of the state variables in the global variable array.
GSTRING* *m_XNames*	Array of *NumX* GSTRINGs (strings up to 256 characters long) corresponding to the gPROMS pathnames of the state

	variables.
DOUBLE* *m_AMatrix*	Array of DOUBLEs corresponding to Matrix A (*m_NumX* * *m_NumY* elements sorted by rows).
DOUBLE* *m_Bmatrix*	Array of DOUBLEs corresponding to Matrix A (*m_NumX* * *m_NumU* elements sorted by rows).
DOUBLE* *m_Cmatrix*	Array of DOUBLEs corresponding to Matrix A (*m_NumY* * *m_NumX* elements sorted by rows).
DOUBLE* *m_Dmatrix*	Array of DOUBLEs corresponding to Matrix A (*m_NumY* * *m_NumU* elements sorted by rows).

11.6 Using the `simpleEventFOI`

The `simpleEventFOI` is an implementation of the Foreign Object Interface (see chapter 4 of the "*gPROMS Advanced User Guide*") provided as standard with the gSERVER software. It provides support for a special case of Foreign Object methods that

- take no inputs
- return a single (scalar) REAL output.

When gPROMS calls such a method, it posts a corresponding `GPE_FOI_GET` blocking event (cf. section 11.1.3).

11.6.1 Typical applications of the `simpleEventFOI` interaction mechanism

The main purpose of the `simpleEventFOI` is to allow a gBA to interact with a gPROMS `PROCESS` entity before the computations associated with a simulation, optimisation or parameter estimation activity is actually started. This interaction may aim to:

- `SET` the values of one or more `PARAMETER`s;
- `ASSIGN` values to one or more input variables;
- provide `INITIAL` values for one or more variables.

The interaction between the gBA and the `simpleEventFOI` are unlike those involving the eventFPI mechanism (cf. section 11.5) which take place after the start of the computations associated with a simulation activity. In fact, some of the information provided by the `simpleEventFOI` (e.g. the value of a `PARAMETER` describing the number of trays in a distillation column) may affect the size and/or the form of the mathematical model that will actually be used for the activity.

11.6.2 Example of `simpleEventFOI` usage

An example of the use of `simpleEventFOI` is shown below:

```
MODEL Tank
```

```
        PARAMETER
            Diameter AS REAL

        ...
    END

    PROCESS DoTestA
        PARAMETER
            theFO AS FOREIGN_OBJECT "simpleEventFOI"
        UNIT
            SmallTank AS Tank
            LargeTank AS Tank
        SET
            theFO := "Dummy";  # The instance name used to set the FO
    PARAMETER is
                               # irrelevant
            SmallTank.Diameter := theFO.SmallTankDiameter ;
            LargeTank.Diameter := theFO.LargeTankDiameter ;
        ...
    END
```

Now consider a gBA applying the gSERVER's `gproms_simulate_mt`,`gproms_optimise_mt`, or `gproms_estimate_mt` function to the `DoTestA PROCESS` entity[8]. This will result in gPROMS posting two `GPE_FOI_GET` events, each requesting the value of a real quantity.

Having retrieved a `GPE_FOI_GET` event via a call to the `gproms_retrieve_event` function (cf. section 11.2), the gBA can cast the `m_Data` pointer in the `GPEvent` structure (cf. section 11.1.4) to a `GPFPIGetSendData*` pointer. The precise form of the `GPFPIGetSendData` structure is described in section 11.5.2. In the above example, the members of the `GPFPIGetSendData` structure corresponding to these events would have the values:

```
    m_Signal        == "SIMPLEFOGET"
    m_Time          == 0.0
    *m_Status       == 1 /* SUCCESS */
    m_NumVars       == 1
    m_VarNames[0]   == "SMALLTANKDIAMETER" /* or "LARGETANKDIAMETER". */
    m_VarTypes[0]   == 1 /* REAL */
    m_VarValues[0] == 0.0
```

A gBA is normally expected to change `m_VarValues[0]`, placing in it the requested value, and then call `gproms_reply_event`. The new value of `m_VarValues[0]` will then be returned to gPROMS as the value of the corresponding method.

If the gBA does not recognise `m_VarNames[0]` as the name of a quantity that it can provide, it should set `*m_Status = 0` to indicate failure.

[8] Because of the blocking nature of the `GPE_FOI_GET` events (cf. section 111.1.2), the `queueCapacity` argument to these functions must be set to a non-zero value.

12 Managing Asynchronously Executing Activities

This category includes three auxiliary functions that can be used by a gBA to manage activities executing asynchronously with it:

- `gproms_get_activity_status` allows a gBA to determine the status of a simulation, optimisation or parameter estimation activity launched asynchronously using the functions of section 10.
- `gproms_kill_activity` allows a gBA to immediately terminate the execution of any activity executing asynchronously (see section 12.1).
- `gproms_set_activity_priority` allows a gBA to assign a priority to an asynchronously executing activity that is different to the priority of the gBA itself.

12.1 gproms_get_activity_status

Functionality: Query the status of a gPROMS activity executing asynchronously.

Synopsis: void gproms_get_activity_status (gBAClient, backgroundActivity, lastStatus)

Name of argument	Type	Description	Specified on	
			Entry	Exit
gBAClient	void*	gBA client identification interface (see section 5.2)	yes	no

background Activity	INTEGER*	The currently executing asynchronous activity:
		`GPRUN_STOPPED` (idle)
		`GPRUN_SIMULATING`
		`GPRUN_OPTIMIZING`
		`GPRUN_DESIGNING_EXPERIMENT`
		`GPRUN_ESTIMATING`
			no	yes

lastStatus	INTEGER*	`GPSTAT_xxx` code of the most recently completed activity (cf. section 5.3). This is only valid once `backgroundActivity = GPRUN_STOPPED`.	no	yes

Remarks

- This function may be used to “poll” or query the status of a gPROMS simulation, optimisation or parameter estimation activity executing asynchronously.

Header: Declared in `gSERVER.h`

12.2 gproms_kill_activity

Functionality: Stop a gPROMS simulation, optimisation or parameter estimation executing asynchronously.

Synopsis: void gproms_kill_activity(gBAClient, status)

Name of argument	Type	Description	Specified on	
			Entry	Exit
gBAClient	void*	gBA client identification interface (see section 5.2)	yes	No
status	INTEGER*	Return status is always `GPSTAT_OK`. See section 5.3.		

Remarks

- This function can immediately interrupt any gSERVER operation executed asynchronously. Upon return, the asynchronous execution will have been terminated; the gBA is then free to invoke other gSERVER functions.
- Calling this function destroys the existing event queue so until a new activity is started with `gproms_xxx_mt` any pending or future calls to `gproms_receive_event` or `gproms_reply_event` will return the `GPSTAT_NO_QUEUE` status code.

Header: Declared in `gSERVER.h`

12.3 gproms_set_activity_priority

Functionality: Sets the priority of a simulation, optimisation or parameter estimation activity that is executing asynchronously.

Synopsis: void gproms_set_activity_priority(gBAclient, newPriority, status);

Name of argument	Type	Description	Specified on	
			Entry	Exit
gBAClient	void*	gBA client identification interface (see section 5.2)	yes	no
newPriority	INTEGER	This must be set to one of the following values: `GPEXPRI_LOW` `GPEXPRI_NORMAL` `GPEXPRI_HIGH`	yes	no
status	INTEGER*	Return status. See section 5.3.	no	yes

Remarks

- This function is concerned with the management of shared CPU resources between a gPROMS activity that is executing asynchronously and the gBA that has launched it. By default, both the activity and the gBA have the same priority. However, in some cases, this may prevent other functions of the gBA from receiving enough CPU time. A typical situation is when the GUI of a gBA appears not to be sufficiently responsive to user actions while executing an activity. In such cases, the gBA may try reducing the proirity of the gPROMS activity by invoking this function. Of course, a trade-off may have to be reached if the actual time spent completing the activity rises to an unacceptable level.

Header: Declared in `gSERVER.h`

Part III: Appendices

13 gSERVER Functions for FORTRAN Clients

The previous chapters have concentrated on the C/C++ interface to gSERVER. However most of the gSERVER functionality is also available to FORTRAN clients.

In order to accommodate FORTRAN's naming and parameter passing conventions gSERVER exports a set of FORTRAN compatible functions with names starting with the `gpfor_` prefix instead of `gproms_`. For example, the FORTRAN equivalent for `gproms_start` is `gpfor_start`, the equivalent of `gproms_simulate_mt` is `gpfor_simulate_mt`, and so on.

The number and order of the parameters for the `gpfor_` functions are exactly the same as those of the corresponding `gproms_` ones. The table below indicates which FORTRAN types replace the C/C++ types used when describing the functions in the previous chapters.

C/C++ parameter types	FORTRAN parameter types
`CHAR*`	`CHARACTER*256`
`INTEGER`	`INTEGER`
`INTEGER*`	`INTEGER`
`DOUBLE`	`DOUBLE PRECISION`

There are no FORTRAN equivalents for the `gproms_set_mode`, `gproms_gdoo`, `gproms_eso`, `gproms_retrieve_event` and `gproms_reply_event` functions.

N.B. The C/C++ declarations of the `gpfor_` functions can be found in the `gSERVERfor.h` header file. C/C++ programmers will notice that all parameters are pass by pointer (needed for FORTRAN pass by reference semantics) and that functions taking string arguments have additional trailing parameters which FORTRAN uses to pass the string length.

14 Example Applications

A number of simple client example applications are presented in this Appendix. They primarily aim to illustrate the asynchronous features of gSERVER. More particularly:

- simpleClient.cxx: A C++ client that uses `gproms_evaluate` to single-step a gPROMS model.
- advancedClient.cxx: A variation of the above, using the low-level `gproms_retrieve_event` and `gproms_reply_event` to achieve the same single-stepping.
- fortranClient.f: A sample FORTRAN client using `gpfor_evaluate` to achieve the same effect.

14.1 gPROMS input file for example applications

All three examples considered in this appendix use the same model file, called `eventFPI.gPROMS`, and simulate the `FPIEVAL PROCESS` entity defined in this file. The most important part of the simulation schedule is contained in a TASK entity called `TestFPI`. This is reproduced below:

```
TASK TestFPI
  PARAMETER
    Tank     AS MODEL MixingTank
    Interval AS REAL
  VARIABLE
    StatFlag AS LOGICAL
    simInterval as REAL

  SCHEDULE
    SEQUENCE
      StatFlag := TRUE;
      simInterval := Interval;

      WHILE StatFlag DO
        SEQUENCE
        GET
          SIGNALID "GET"
          STATUS StatFlag
          simInterval ;
          Tank.Fin(1) ;
          Tank.Fin(Tank.NoInput) ;
        END

        CONTINUE FOR simInterval

        SEND
          SIGNALID "SEND"
          STATUS StatFlag
          Tank.V ;
          Tank.C(1) ;
        END
      END # while
    END # sequence
  END #schedule
END # task
```

The schedule is an “infinite” `WHILE` loop, with a series of `GET`/`SEND` event pairs. The client example programs are tailored to respond to such a sequence of events. A few noteworthy details of this simulation are:

- The `WHILE` loop uses the logical variable `StatFlag` to determine when to terminate the simulation. `StatFlag` takes its value from the `STATUS` of the FPI elementary tasks in the schedule. This combination permits the use of `gproms_evaluate` with `dimU = dimY = -1` to stop the simulation. If this structure is absent, the only way to stop a simulation is via `gproms_kill_activity`.
- The real variable `simInterval` controls the length of the integration between the `GET` and `SEND` events, since it is the controlling argument of the `CONTINUE FOR` task.
- The `GET` event asks for the value of three variables and `SEND` transmits the calculated values of two other variables. The gBA client will receive the variables in arrays (vectors) whose elements are placed exactly in the order in which they appear in the FPI task.

14.2 Example 1: simpleClient.cxx

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

using namespace std;

/**
 * Stop gSERVER and return the licence.
 */
void StopServer()
{
    INTEGER status;
    INTEGER backgroundActivity;

    //
    // If there is still background activity then shut it down with
    // gproms_kill_activity().
    //
    gproms_get_activity_status(NULL, &backgroundActivity, &status);
    if (backgroundActivity != GPRUN_STOPPED)
    {
        gproms_kill_activity(NULL, &status);
    }

    gproms_end(NULL, &status);
    if (status != GPSTAT_OK)
    {
        cout << "Error ending gSERVER (" << status << ")" << endl;
        gproms_print_error(status);
    }
}

int main(int argc, char** argv)
{
    //
    // Register a function to stop gSERVER when the program exits.
    //
    atexit(StopServer);

    INTEGER status;

    //
    // Redirect output if necessary. Here we don't need it.
    //

    // ...

    //
    // Start gSERVER.
    //
    gproms_start(NULL, "gSRV", -1, &status);
    if (status != GPSTAT_OK)
    {
        cout << "Error starting gPROMS! (" << status << ")" << endl;
        gproms_print_error(status);
        exit(1);
    }

    //
    // Select the .gPROMS model input file (located in the 'input' directory).
    //
    CHAR* gpFile = "eventFPI";
    gproms_select(NULL, gpFile, 0, 0/*silent*/, &status);
    if (status != GPSTAT_OK)
    {
        cout << "Error selecting model file " << gpFile;
        cout << " (" << status << ")" << endl;
        gproms_print_error(status);
        exit(1);
    }

    //
    // Execute gPROMS simulation in the background.
    //
    CHAR* gpProcess = "FPIEVAL";
    gproms_simulate_mt(NULL, gpProcess, 0/*silent*/, -1, &status);
    if (status != GPSTAT_OK)
    {
        cout << "Error simulating process " << gpProcess;
        cout << " (" << status << ")" << endl;
        gproms_print_error(status);
        exit(1);
    }

    //
    // gPROMS MODEL FILE
    // There is a close relationship between the gPROMS PROCESS and the client
    // program. The loop below corresponds to PROCESS that has a SCHEDULE of
    // the form GET-CONTINUE-SEND, where the following variables are transmitted:
    // GET   (IN)  :  simInterval, Tank.Fin(1), Tank.Fin(Tank.NoInput)
    // SEND  (OUT): Tank.V, Tank.C(1)
    //
    // We want to make evaluations of OUT = gproms(IN). The I/O buffers should
    // be of the proper size and the order of the variables the same as that in
    // the gproms model file statements.
    //

    DOUBLE in[3] = {5.0, 1.0, 1.5}; // Inputs to GET
    DOUBLE out[2]; // Outputs from SEND
    int iteration = 0;
    //
    // The main loop.
    //
    while (iteration < 10000)
    {
        //
        // Evaluate the next GET SEND sequence.
        //
        gproms_evaluate(NULL, in, 3, out, 2, &status);
        if (status != GPSTAT_OK)
        {
            cout << "Evaluation error (" << status << ")" << endl;
            gproms_print_error(status);
            exit(1);
        }

        //
        // Don't print every iteration result.
        //
        if (iteration % 100 == 0)
        {
            cout << "ITERATION = " << iteration;
            cout << ", Tank.V = " << out[0];
            cout << ", Tank.C = " << out[1] << endl;
        }

        //
        // Recalculate the inputs for the next iteration.
        //
        in[1] += 0.1;
        in[2] += 0.1;
        iteration++;
    }

    exit(0); // Normal termination - calls StopServer() function.
    return 0;
}
```

This first C++ client is pretty straightforward. It uses `gproms_start` to initialise a gSERVER session, selects a file with `gproms_select` and performs a background simulation using `gproms_simulate_mt`. If all these steps are successful, it enters a for-loop utilizing `gproms_evaluate` to retrieve a sequence of `GET`/`SEND` events, printing out intermediate results. All execution paths that cause the program to exit call the local `StopServer` function which uses `gproms_kill_activity` to stop the asynchronous execution and `gproms_end` to terminate the session.

Note that the `GET` task has three variables where the first one is an “artificial” (helper) variable denoting the simulation interval. This helper variable is treated like the `MODEL` variables in the `GET` event and is therefore part of the vector `in[]`.

Note that after each call to any gSERVER function, the *status* variable is tested in order to determine the outcome. Any value other than `GPSTAT_OK` indicates an error.

14.3 Example 2: advancedClient.cxx

An alternative approach to hook into the FPI events of our sample model is to use the elementary `gproms_retrieve_event` and `gproms_reply_event` functions to retrieve each `GET` or `SEND` event separately. The initialisation and termination of gSERVER is identical to `simpleClient.cxx`. The difference is the main for-loop where the FPI events are extracted. This block is presented below:

```
    int iteration = 0;
    GPEvent* pEvent;
    GPFPIGetSendData* pData;
    DOUBLE in[3] = {5.0, 1.0, 1.5}; // Inputs to GET.
    DOUBLE out[2]; // Outputs from SEND.
    bool expectingGet = true;
    bool finished = false;

    //
    // The main loop.
    //
    while (!finished && (iteration < 10000))
    {
        //
        // Retrieve the next event.
        //
        gproms_retrieve_event(NULL, -1, &pEvent, &status);
        if (status != GPSTAT_OK)
        {
            cout << "Error retrieving event (" << status << ")" << endl;
            gproms_print_error(status);
            exit(1);
        }

        //
        // Check what type of event we received and procede accordingly.
        //
        switch (pEvent->m_EventId)
        {
        case GPE_START:
        case GPE_SIMULATION_START:
            //
            // Every background simulation begins with a GPE_START
            // followed by a GPE_SIMULATION_START event.
            // We don't have to do anything with them other than reply.
            //
            break;

        case GPE_FPI_GET:
            if (!expectingGet)
            {
                cout << "Events are out of sequence, expected SEND but ";
                cout << "received GET." << endl;
                exit(1);
            }

            //
            // Update the inputs in the GET event.
            //
            pData = (GPFPIGetSendData*)(pEvent->m_Data);
            pData->m_VarValues[0] = in[0];
            pData->m_VarValues[1] = in[1];
            pData->m_VarValues[2] = in[2];
            expectingGet = false;
            break;

        case GPE_FPI_SEND:
            if (expectingGet)
            {
                cout << "Events are out of sequence, expected GET but ";
                cout << "received SEND." << endl;
                exit(1);
            }

            //
            // Extract the outputs from the SEND event.
            //
            pData = (GPFPIGetSendData*)(pEvent->m_Data);
            out[0] = pData->m_VarValues[0];
            out[1] = pData->m_VarValues[1];

            //
            // Don't print every iteration result.
            //
            if (iteration % 100 == 0)
            {
                cout << "ITERATION = " << iteration;
                cout << ", Tank.V = " << out[0];
                cout << ", Tank.C = " << out[1] << endl;
            }

            //
            // Recalculate the inputs for the next iteration.
            //
            in[1] += 0.1;
            in[2] += 0.1;
            iteration++;

            expectingGet = true;
            break;

        case GPE_ACTIVITY_FINISH:
        case GPE_ACTIVITY_INTERRUPT:
            //
            // The activity has finished or been interrupted.
            // Set the 'finished' flag to break out of the loop.
            //
            finished = true;
            break;

        default:
            cout << "Error: Unexpected event id: " << pEvent->m_EventId;
            cout << endl;
            exit(1);
            break;
        }

        //
        // Reply to the event.
        //
        gproms_reply_event(NULL, &status);
        if (status != GPSTAT_OK)
        {
            cout << "Error replying to event (" << status << ")" << endl;
            gproms_print_error(status);
            exit(1);
        }
    }
```

This main loop is identical with the one in `simpleClient.cxx`, albeit more generic. It intercepts the “raw” FPI events as they appear in the gPROMS simulation SCHEDULE (in a way similar to what `gproms_evaluate` performs internally[9]). The main advantage of using the low-level `gproms_retrieve_event` and `gproms_reply_event` is that *any* event sequence can be handled, in contrast to `gproms_evaluate`, which expects a *fixed* sequence `GET-SEND-GET-SEND-GET-SEND`…

[9] `gproms_evaluate` is a sequence of two consecutive `_retrieve_event/_reply_event` call pairs, the first retrieving the GET event, followed by the SEND event.

14.4 Example 3: fortranClient.f

The final example is a FORTRAN client with functionality identical to the first C++ sample, `simpleClient.cxx`. Note that the FORTRAN-friendly `gpfor_xxx` functions are used in place of the various `gproms_xxx` equivalents (cf. Appendix A).

```
      program fortranClient

      double precision in(3), out(2)
      integer istatus, iteration
      character*256 licence, model, password, process

      include 'gSERVERstatus.f'
c
c     Start gSERVER
c
      licence = 'gSRV'
c
c     Redirect output if necessary. Here we don't need it.
c
c     ...

      call gpfor_start(0, licence, -1, istatus)
      if (istatus .ne. GPSTAT_OK) then
          print *, 'Error starting gPROMS!', istatus
          call gpfor_print_error(istatus)
          stop
      endif

c
c     Select the .gPROMS model input file (located in the 'input' directory.
c
      model = 'eventFPI'
      password = ''
      call gpfor_select(0, model, password, 0, istatus)
      if (istatus .ne. GPSTAT_OK) then
          print *, 'Error selecting model file', istatus
          call gpfor_print_error(istatus)
          stop
      endif

c
c     Execute gPROMS simulation in the background.
c
      process = 'FPIEVAL'
      call gpfor_simulate_mt(0, process, 0, -1, istatus)
      if (istatus .ne. GPSTAT_OK) then
          print *, 'Error simulating process', istatus
          call gpfor_print_error(istatus)
          stop
      endif



c
c     gPROMS MODEL FILE
c     There is a close relationship between the gPROMS PROCESS and the client
c     program. The loop below corresponds to PROCESS that has a SCHEDULE of
c     the form GET-CONTINUE-SEND, where the following variables are transmitted:
c     GET  (IN) :  simInterval, Tank.Fin(1), Tank.Fin(Tank.NoInput)
c     SEND (OUT): Tank.V, Tank.C(1)
c
c     We want to make evaluations of OUT = gproms(IN). The I/O buffers should
c     be of the proper size and the order of the variables the same as that in
```

```
c     the gproms model file statements.
c

      in(1) = 5.0
      in(2) = 1.0
      in(3) = 1.5
      iteration = 0

c
c     The main loop.
c
      do while (iteration .lt. 10000)
c
c        Evaluate the next GET SEND sequence.
c
          call gpfor_evaluate(0, in, 3, out, 2, istatus)
         if (istatus .ne. GPSTAT_OK) then
              print *, 'Evaluation error', istatus
              call gpfor_print_error(istatus)
              stop
          endif

c
c         Don't print every iteration result.
c
         if (mod(iteration, 100) .EQ. 0) then
             print *, 'ITERATION=',iteration,' Tank.V=',out(1),
     &        ' Tank.C(1)=',out(2)
          endif

c
c        Recalculate the inputs for the next iteration.
c
          in(2) = in(2) + 0.1
          in(3) = in(3) + 0.1
          iteration = iteration + 1
       end do

c
c     Interrupt simulation and wait for gPROMS to terminate.
c
      call gpfor_kill_activity(0, istatus)
      if (istatus .ne. GPSTAT_OK) then
          print *, "Error stopping gPROMS", istatus
          call gpfor_print_error(istatus)
          stop
      endif

c
c     stop gSERVER and return the licence
c
      call gpfor_end(0, istatus)

      stop
      end
```

All the comments made in the discussion of the first example `simpleClient.cxx` apply here, too. The only slight difference is the *password* argument passed to `gpfor_select`. When no password is required, an empty string needs to be passed — whereas `gproms_select` would require a NULL pointer.

14.5 Building the examples

All the examples are located in the directory `gPROMS/src/examples/gSERVER` of the gPROMS installation (cf. section 2.1). This directory also contains projects for MS Visual C++ or Compaq Visual Fortran for building on Windows, as well as a `Makefile` for building on the supported UNIX platforms. The projects and `Makefile` show the necessary compiler and linker settings for each platform.

The file `readme.txt` in this directory contains a complete file listing and further instructions not covered in this manual.

15 FAQ

15.1 Why does the function gproms_xxxx return `GPSTAT_SERVER_BUSY`?

There are two reasons why a gSERVER function will return status `GPSTAT_SERVER_BUSY`:

1. You have started an asynchronous activity using `gproms_simulate_mt`, `gproms_optimise_mt` or `gproms_estimate_mt` that has not yet completed.

   The only gSERVER functions that can be called whilst there is an uncompleted asynchronous activity are:

   - `gproms_get_activity_status`
   - `gproms_get_error`
   - `gproms_kill_activity`
   - `gproms_print_error`
   - `gproms_reply_event`
   - `gproms_retrieve_event`
   - `gproms_set_activity_priority`

2. You are calling gSERVER functions simultaneously from more than one thread.

   We recommend that all gSERVER functions are called from a single thread.

15.2 How can I be sure an asynchronous activity has completed?

You can be sure that the asynchronous activity has completed when one of the following 3 conditions is met and before another asynchronous activity is started:

1. A call to `gproms_get_activity_status` returns a `backgroundActivity` of `GPRUN_STOPPED`.
2. After you have retrieved (with `gproms_retrieve_event`) a `GPE_ACTIVITY_FINISH` or `GPE_ACTIVITY_INTERRUPT` event (this is only true since gSERVER 2.1.1).
3. After a call to `gproms_kill_activity` (this function forces the asynchronous activity to complete).

16 Summary of Changes: gSERVER 2.1.1 to gSERVER 2.2

16.1 Changes to gproms_start

`gproms_start` now returns `GPSTAT_SERVER_ALREADY_RUNNING` if it is called a second time without `gproms_end` having been called in between.

Licensing for gBAs has changed and all new gBAs should call `gproms_start` with the "`gSRV`" licence type.

Support for the "`RuntimeModel`" licence type has been removed – if you have been using this licence type please contact PSE for advice on an alternative.

Support for the "`RuntimeActivity`" and "`Developer`" licence types has been deprecated and will be removed in a future version of gSERVER – if you have been using this licence type please contact PSE for advice on an alternative.

16.2 Other changes

`gproms_eso` now returns status `GPSTAT_WRONG_LICENSE_TYPE` if it is called whilst using a "`RuntimeActivity`" licence, it previously incorrectly reported `GPSTAT_ENCRYPTED_ACCESS`.

`gproms_select` now returns `GPSTAT_WRONG_LICENSE_TYPE` if you attempt to select an unencrypted .gPROMS model whilst using a "`RuntimeActivity`" licence; it previously incorrectly reported `GPSTAT_ENCRYPTED_ACCESS`.

The `gBAclient` parameter has been removed from the `gproms_redirect_output` and `gpfor_redirect_output` functions.

The `gproms_set_mode` function has been added (see section 6.4).

17 Summary of Changes: gSERVER 2.1 to gSERVER 2.1.1

The `GPE_ACTIVITY_FINISH`/`GPE_ACTIVITY_INTERRUPT` event that ends any asynchronous execution using events is now only posted to the event queue after the activity has been flagged as stopped.

In gSERVER 2.1, these events were posted just before the activity was flagged as stopped; the result was that, if you called another gSERVER function immediately after replying to `GPE_ACTIVITY_FINISH` or `GPE_ACTIVITY_TERMINATE`, the function sometimes returned `GPSTAT_SERVER_BUSY`.

18 Summary of Changes: gSERVER 2.0 to gSERVER 2.1

18.1 Changes for C/C++

- The standard gSERVER headers have been rearranged; `gServer.h, eventServer.h` and `gStatusCodes.h` have been removed; include `gSERVER.h` instead.
- Two functions have been renamed and also have new parameter signatures:
  - `gproms_getevent` is now `gproms_retrieve_event`.
  - `gproms_replyevent` is now `gproms_reply_event`.
- Function `gproms_stop_simulation_mt` has been removed. Use `gproms_kill_activity` instead.
- The parameter signatures of the following functions have changed:
  - `gproms_simulate_mt`
  - `gproms_optimise_mt`
  - `gproms_estimate_mt`
  - `gproms_get_activity_status`
  - `gproms_kill_activity`
  - `gproms_set_activity_priority`
- Two new functions have been added:
  - `gproms_print_error`
  - `gproms_get_error`
- The list of gSERVER status codes has been expanded.
- The number of events produced by gPROMS and handled with `gproms_retrieve_event` and `gproms_reply_event` has been expanded and now includes events for the `SENDMATHINFO` and `LINEARISE` elementary tasks as well as a simple FOI event and diagnostic events when gPROMS starts and stops.

18.2 Changes for FORTRAN

All of the modifications listed in section 18.1 for C/C++ apply. In addition:

- An include file containing FORTRAN compatible declarations of all the gSERVER status codes is now provided (`gSERVERstatus.f`). This should be used in preference to hard-coding the gSERVER status codes as `INTEGER`s in the gBA.
- All functions except `gproms_retrieve_event`, `gproms_reply_event`, `gproms_eso` and `gproms_gdoo` now have FORTRAN compatible `gpfor_` equivalents.