C Interface Generator Application (IG)

version 1.8

Torbjörn Törnquist, Peter Lundell

1997-05-02

Typeset in LaTeX from SGML source using the DOCBUILDER 3.0 Document System.

Contents

Chapter 1

IG User's Guide

The Interface Generator Application (*IG*) makes it easy to communicate between Erlang and C/C++ programs.

IG provides an easy way of making C functions accessible from Erlang and vice versa. The underlying communication can optionally be be based upon sockets or the `open_port/2` BIF mechanism.

> Note:
> The Interface Generator will *not* be supported from next release of Erlang/OTP. Please, avoid to use it; use the application IC instead.

1.1 Overview

The Interface Generator (IG) is a tool which makes it easy to interface C with Erlang. With IG, it is possible to transparently access the “other” language directly. C functions look like Erlang functions on the Erlang side, and Erlang functions look like C functions on the C side. IG accomplishes this by generating interface *stubs* on both sides. These stubs then handle all communication and parameter passing between Erlang and C.

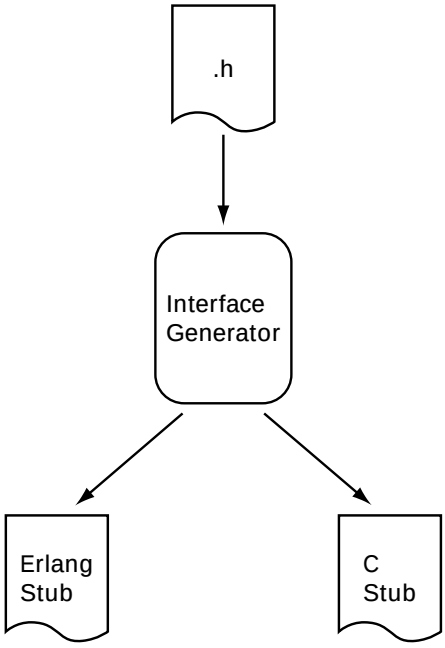

Figure 1.1: The IG Generated Stubs

A header file defines the interface for which IG generates stubs. This header file is a real C header file which is included in user’s C code. The header file must be augmented with some IG keywords, but it otherwise looks like a normal C header file. The keywords do not disturb normal C header file duties. Currently, IG does not handle the complete ANSI C syntax, but a subset thereof.

IG generates interface stub code on both the Erlang side and the C side. The stub code then takes care of argument marshalling and the routing of function calls. The stub on the Erlang side contains the interface functions and acts as shadow for the “real” C functions. When an Erlang stub function is called, it sends a request for the function call to the C stub. The C stub does the real function call and passes the result back to Erlang. The IG stubs use the standard Erlang `port` mechanism for their communication, but this can be changed to sockets if required.

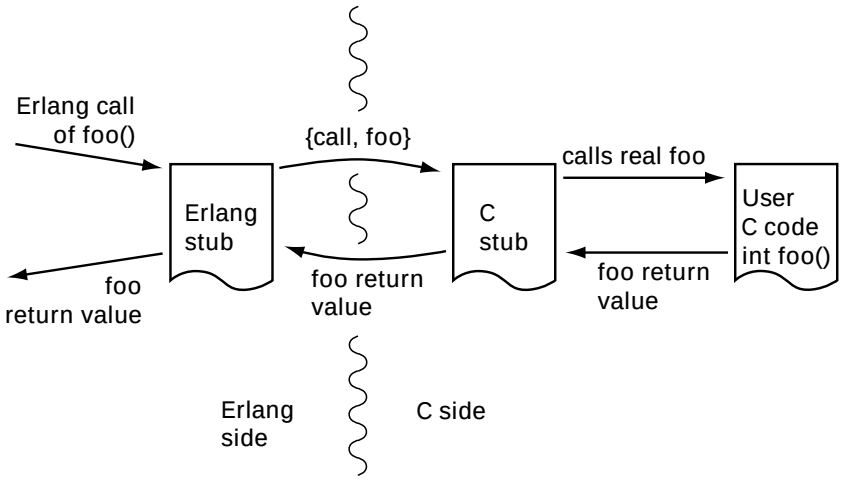

Figure 1.2: The Call Chain

Refer to the IG distribution which contains many examples. The reader is recommended to read through and run these examples.

IG Basics

IG works by generating stub files from a C header file which includes some added IG keywords. Two, and possibly three, files are normally produced:

- the Erlang stub which is an Erlang `.erl` file
- the C stub which is a C `.c` file
- an optional Erlang `.hrl` header file which is produced if pre-processor macros or struct types are used.

IG reads the input header file and interprets the keywords which serve as guides for the generator. They are needed because more information than is normally present in a C header file is needed for the stub generation.

The following section provides a quick summary of all the IG keywords and small examples of their use. The examples include most constructs. These examples illustrate what is written in an IG header file.

Keywords Used in Erlang to C Situations

IG_fun This keyword marks a function as accessible from Erlang. `IG_fun int plus(int a, intb);` means that the function `plus` can be called in Erlang.

IG_void This keyword marks a function as accessible from Erlang, but the function does not return a value. It is a one-way call as shown in the example `IG_void void do_it();`

/*IG_var*/ This keyword marks a C variable as accessible from Erlang. The function call generates a get and set function in Erlang. Example: `/*IG_var*/ extern int my_int;`. The variable must be declared extern.

Keywords Used in C to Erlang Situations

IG_call This keyword marks a function as accessible from C. `IG_call int get_time();` is in effect a declaration of an Erlang function.

IG_cast One-way call of an Erlang function. It is to `IG_call` what `IG_void` is to `IG_fun`. Example: `IG_cast void throw_msg(IG_string str);`

General Keywords and Types

/*IG*/ The `/*IG*/` comment keyword is used together with pre-processor directives. `/*IG*/ #define PI 3.14` defines an Erlang macro with the name `PI`. Note the space after the comment. The definitions are generated to the `.hrl` file only.

IG_binaryPtr The C type definition of the Erlang binary type, which can be used to pass large chunks of memory to and from Erlang. This is not really a keyword, but rather an IG type definition. Example: `IG_fun IG_binaryPtr get_state();`

IG_string This is used for the C `char*` type when the type means a character string as opposed to those that are really pointers to bytes. Do not use `IG_string` when `char*` denotes an arbitrary pointer to bytes. For example, the declaration `IG_fun IG_string get_version_str();` tells IG that the function `get_version` returns a NULL terminated string so that IG will build an Erlang string of the characters. If `char*` is used, then IG treats the return value as an opaque pointer instead of trying to build an Erlang string of the characters.

struct While the `struct` keyword is not really an IG keyword, IG will generate Erlang records for every struct type declaration it sees. Records are generated to the `.hrl` file only.

The IG keywords enable IG to generate stubs which handle all communication link set-up, message passing, marshalling of arguments, and return values.

The design philosophy of IG is to make important aspects of C valid in Erlang, and to find solutions which ensure that it feels natural to use C with Erlang.

A Hands-On Example

This section describes an example which may help in understanding the basic behaviour of IG. Readers can skip this section at their own discretion.

This simple example calls the C function `foo` from Erlang and goes through all steps, from the header file to the final call of `foo` from Erlang. The first step is to write the header file which defines the interface to `foo`. We do this in a file called `ex1.h` which looks as follows:

```
IG_fun int foo(IG_string name, int age);
```

This means that `foo` can be called from Erlang. The next step is to write the actual definition of `foo`. We will place this definition in the file `ex1.c` which looks as follows:

```
#include <stdio.h>
#include "ig.h"
IG_fun int foo(IG_string name, int age)
{
  fprintf(stderr, "foo: name='%s', age=%d\n", name, age);
  return age;
}
```

Note:
We use stderr to print the string because IG uses stdin and stdout for its port communication with Erlang. This can be configured (see also Sockets in Advanced IG Topics [page 14]).

The stubs are then generated in an Erlang shell:

```
UNIX> erl
Erlang (JAM) emulator version 4.4.2

Eshell V4.4.2  (abort with ^G)
1> ig:gen(ex1).
Calling parser
C symbol allocation..........
Generating C code..........
Generating callbacks
C code generated to file: ex1_stub.c
Generating Erlang code.
Erlang code generated to file: ex1.erl
Generator ready
ok
2>
```

We now have an Erlang stub called ex1.erl and a C stub called ex1_stub.c. These must be compiled into an Erlang jam file and an executable file. We begin by compiling the Erlang file in the Erlang shell:

```
2> c(ex1).
{ok,ex1}
```

The C stub is then compiled and linked into an executable in a Unix shell:

```
UNIX> gcc -L. -o ex1 ex1.c ex1_stub.c igio.o igmain.o -lerl_interface
UNIX>
```

This produces an executable ex1 which implements the foo function. The ig.h, igio.o, igmain.o, and erl_interface files are part of the IG magic. They contain the declarations, the IO routines, the main loop, and the value representation. These are files from the IG distribution. The reader will have to find the place where IG was installed and then try the usr/include directory for the ig.h file, and the usr/lib for the object files and libraries. We have OTP installed at /home/super/otp/otp_sunos5, so we find the ig.h file in the /home/super/otp/otp_sunos5/usr/include directory.

We are now ready to run the example from Erlang. We start the stubs from Erlang and note that this also starts the C program:

```
3> P=ex1:start().
<0.22.0>
4> ex1:foo(P, "Nils", 34).
foo: name='Nils', age=34
                        {ok,34}
5>
```

The `foo` function has one more parameter than its C counterpart and this is the handle to the C program. The return value is the tuple `{ok,34}`.

In summary, the steps for using IG are:

1. write the header file
2. write the C implementation
3. generate the stubs
4. compile the stubs into `jam` and an executable C program
5. start the program from an Erlang shell.

1.2 Using C from Erlang

This section describes the inclusion of C programs in Erlang and includes the following topics:

- Calling C functions
- Access to C variables
- Pointers, struct types and records
- Erlang binaries.

Calling C Functions

Most commonly, IG is used for accessing functions which are written in C. These include sourced software which must be interfaced with Erlang, or certain routines which must be written in C for some reason.

The interface to be IG generated is written in a standard C header file with the appropriate IG keywords. Most of the time, this only involves writing `IG_fun` in front of the function declarations in the header file that is part of the interface which is called from Erlang. These will then be accessible from Erlang and IG includes them in both of the stub files generated. When IG encounters `IG_fun int hello();` in a header file, it puts a shadow function `hello` in the Erlang stub and makes a "real" call to the `hello` function in the C stub. The `hello` function is a normal C function which can be called from C without any overhead and the header file is still a valid C header file. If the C function is a "one-way" function, which means that we are not interested in the return value and we do not want to wait for completion, then the IG keyword `IG_void` may be used instead. In this case, the generated Erlang stub will not block and wait for an answer, but returns to the calling party immediately.

Access to C Variables

The IG keyword `/*IG_var*/` gives access to global C variables. By using this keyword, IG generates access methods (get and set) for the variable into its stubs. It is also possible to get the address and size of the variable to Erlang.

Variables are marked for generation by the keyword `/*IG_var*/`. If `/*IG_var*/ extern int my_count;` is a variable in the input header file, then IG will generate stub code functions `my_count/1` and `set_my_count/2`. These stub code functions will get and set the `my_count` variable.

> Note:
> No checks are made for concurrent updates of the variable. This means that the two sides may not set the value of the variable at the same time.

In summary, the following things can be done in Erlang with a `/*IG_var*/` declared variable:

- get and set its value
- get its address as a pointer

- get its size from the C `sizeof` operator.

Pointers, Struct Types and Records

IG treats pointers as integers and allows them to be passed around. IG will not “understand” what the pointers point to, but it is sometimes useful to be able to pass the pointer around to the next C function call. Be careful that the data the pointer points to is not de-allocated or otherwise freed before use.

C structs are translated to Erlang records to allow symbolic access to members of the structs. The translation is straightforward. Include the following code in the header file:

```
typedef struct {
  int len;
  IG_string name;
} person;
```

The following Erlang record will then be generated to an Erlang `.hrl` header file:

```
-record(person, {len,name}).
```

The record person can now be used as arguments to functions and return values from functions. IG will generate packing and unpacking of the record values on both sides.

Erlang Binaries

Erlang binaries are data structures which can be used for passing large pieces of data to and from Erlang. It is defined in C as a struct with a size and a data pointer. The binary structure corresponds to the Erlang binary base type and it can therefore work both ways:

- C can pass opaque data to Erlang
- Erlang can pass Erlang terms hidden in binaries to C.

The `IG_binaryPtr` type is defined in `ig.h` as:

```
typedef struct {
  int size;
  unsigned char *binp;
} *IG_binaryPtr;
```

IG itself uses binaries when passing data between Erlang and C.

1.3 Using Erlang from C

This section describes how to call Erlang functions from C and includes the following topics:

- Calling Erlang functions
- Pointers to callback functions
- What to avoid.

Calling Erlang Functions

IG generated stubs can call Erlang functions from C, in a similar way as C functions can be called from Erlang. Analogous with the `IG_fun` and `IG_void` keywords, there are two keywords for C to Erlang function calls:

- `IG_call` denotes a standard function call
- `IG_cast` denotes a one-way call where no return value is passed back.

For example, if the declaration `IG_call int get_time();` is in the header file, then IG will generate a stub function `get_time` on the C side and will prepare for the actual Erlang call on the Erlang stub side.

The Erlang call is made into what is called a callback module. The callback module is the module where the interface functions are defined. The name of this module is given to IG as input at generate time so that it can generate correct stubs. The callback module name is given as an option to IG, as shown in the following example:

```
ig:gen(my_stuff, [{cback_mod, my_callbacks}])
```

Pointers to Callback Functions

The address of a callback function can be acquired by using the `ig:adr_of/2` function. `ig:adr_of/2` returns the integer value of the pointer to the function, and that value can in turn be passed on to C in other calls. This is useful when function pointers are used extensively on the C side and the functionality is implemented in Erlang. An example is window systems.

A third keyword can be used if only the address is of interest. With the keyword `IG_extern`, IG stores the address of the function but does not generate any stub code for it.

Precautions

Be careful when mixing callbacks and C function calls from Erlang. IG assumes that one party (C or Erlang) is in charge of the communication because both stubs need strict serialization of messages being passed. If two-way communication is really needed, then two IG interfaces may be used instead, or at least two Erlang stubs; one for each direction.

1.4 Supervising C Programs

This section describes the supervision of C programs and includes the following topics:

- Supervision principles and C
- The supervision hooks
- Example of supervision.

Supervision Principles and C

This section assumes a knowledge of the System Architecture Support Libraries (SASL) and readers who do not plan to use SASL features may choose to skip this material.

The Open Telecom Platform (OTP) Erlang system includes some built-in software management routines and supervision structures which are titled the Systems Architecture Support Libraries (SASL). The SASL supervision principles include supervision trees where supervisors keep track of workers, and release handlers which make it possible to update a running system in an ordered way. IG generated stubs can automatically conform to the supervision principles of SASL. However, the support is not as soft, or nice, or powerful as in Erlang, mainly because C is a much rougher language than Erlang. All supervision actions result in a function call to the C side and this is called a *hook*.

The supervision tree and the release handler are managed by a set of messages. For example, the messages order workers to change code or to return information about their state. Erlang programs generated from IG understand these messages and they can be used in supervision trees as normal Erlang workers. The Erlang stub propagates these messages in the form of hooks to the C program. This implies that the C side must listen to the Erlang side at all times. The provided `igmain` and the sample `igsock_server` both do this.

> Warning:
> There are other difficulties when using sockets and the supervision hooks. Code change does not work for socket communication because the C program is not started from within Erlang, and we discourage the user from using them together. In general, the C side should act as a server and subordinate itself under the command of Erlang.

The Supervision Hooks

The SASL supervision hooks are accessed by declaring them in the interface header file, just like any normal interface C function. All hooks are listed below along with their signatures. The hooks are triggered by the functions in the module `sys` in the standard SASL manner.

The hooks are optional and empty default behaviour is provided for all of them. If the C program is a simple program then no hooks need to be declared, and if hooks are desired they can be declared in any combination. For example, if the C program must preserve its state in a system upgrade from one version to another version, then two hooks must be defined:

- erl_c_get_state to pack the state into a binary
- erl_c_init to unpack the same binary after a restart.

The following hooks are available:

erl_c_init This hook is declared as IG_fun void erl_c_init(IG_binaryPtr state);.
This hook is called just when the C program has started. The state argument is the state returned from a previous call to erl_c_get_state, or a binary of size zero. The default behaviour is to ignore the input parameter and do absolutely nothing.

erl_c_terminate This hook is declared as IG_fun void erl_c_terminate(void);.
This hook is called when the C program should terminate. This function stops the C program and does not return a value, the EXIT signal is sufficient. The default behaviour executes an exit(0).

erl_get_state This hook is declared as IG_fun IG_binaryPtr erl_c_get_state(void);.
This hook is called just after code has been changed on the Erlang side. The intention is that the C program should have a chance to pass whatever vital information it needs to the next version of itself. The default behaviour is to return the empty string ("") which shows up as the empty list on the Erlang side ([]). Do not confuse this hook with the erl_c_get_status hook which returns the status of the C program.

erl_c_suspend This hook is declared as IG_fun void erl_c_suspend(void);.
This hook is called immediately before the IG stub suspends itself and enters a safe loop. A safe loop is a loop which is guaranteed to survive a code change, even if the internal data structures have changed. The hook signifies that the C program will become inactive in the same way as the Erlang side. The Erlang code is in a safe state when suspended and it will only process other system messages. The default behaviour is to do nothing.

erl_c_resume This hook is declared as IG_fun void erl_c_resume(void);.
This hook is called when the IG stub returns from its suspended state and resumes execution. The default behaviour is to do nothing.

erl_c_trace This hook is declared as IG_fun void erl_c_trace(int onoff);.
This hook is called when trace is switched on or off at the Erlang side. The onoff parameter is 1 if trace is turned on, otherwise it is 0. Be aware that these hooks do not include any tracing facilities for C in runtime other than those put there by the user. The default behaviour is to do nothing.

erl_c_get_status This hook is declared a: IG_fun IG_string erl_c_get_status();.
This hook is called when the status of the process is requested. The status is a null terminated string and is printed in a standardized way on the Erlang side. The default behaviour is to return the empty string ("") which shows up as the empty list on the Erlang side ([]).

erl_c_log This hook is declared a: IG_fun void erl_c_log(int onoff);.
This hook is called when logging starts or stops. Note the relation to the erl_c_log_to_file hook. The parameter onoff is 0 when logging stops, and 1 when it starts. The default behaviour is to do nothing.

erl_c_log_to_file This hook is declared as: IG_fun void erl_c_log_to_file(IG_string filename);.
This hook is called when logging to a file. Both the Erlang side and the C side are asked to log at the same time but only one file name is provided (to conform to the rest of the SASL routines). Therefore, precautions must be taken to ensure that the C hook does not overwrite the Erlang logging file by using another file name. The default behaviour is to do nothing.

erl_c_no_debug This hook is declared as IG_fun void erl_c_no_debug(void);.
This hook disables all debugging and logging. The default behaviour is to do nothing.

The complete hook calling sequence during a code change is as follows (refer to the erl_get_state [page 11] hook, which is the most important hook in this sequence):

1. erl_c_suspend is called to stop normal operation
2. the code is changed in Erlang
3. erl_c_get_state is called and the C state is saved and passed back to Erlang
4. erl_c_terminate
5. the port is closed
6. a new port is opened and the new version of the program is started
7. erl_c_init is called and the saved state is passed, allowing the C program to restore its previous internal state
8. erl_c_resume is called to signal the start of normal operations.

Supervision Example

There are two differences between supervised C programs and unsupervised C programs:

- With supervision, the C program must be placed in the priv/bin directory of an application. For example, if the program belongs to the application my_app of version 1.0, then its C program must be placed in ...my_app-1.0/priv/bin. This is how code change works; when code is changed, the version of the application has changed so another program will be found.
- In the starting method, unsupervised code is started with the start/0 function while supervised code is started with the sup_start/1 call. The latter also returns a tuple {ok,Handle} instead of simply Handle.

The following example assumes that the reader is familiar with the first example in this section (see Hands-on Example [page 4]). In order to execute this example, we also assume that the previous example has been worked through and that all necessary files have been fully compiled and linked.

The first step is to assure that the C program is put into a directory that will be recognized as an application directory. We first create the required directory structure in the same directory as the first example. We then move the ex1 executable to the correct place in the structure.

```
UNIX> mkdir apps
UNIX> mkdir apps/lib
UNIX> mkdir apps/lib/my_app
UNIX> mkdir apps/lib/my_app/ebin
UNIX> mkdir apps/lib/my_app/priv
UNIX> mkdir apps/lib/my_app/priv/bin
UNIX> mv ex1 apps/lib/my_app/priv/bin
```

We are now ready to start ex1 in supervised mode using the sup_start/1 function. This function takes the application name as input so that the C program can be found. We also start the Erlang shell with the path to the application ebin directory in order to simulate a real application.

```
UNIX> erl -pa apps/lib/my_app/ebin
Erlang (JAM) emulator version 4.4.2

Eshell V4.4.2  (abort with ^G)
1> {ok,P} = ex1:sup_start(my_app).
{ok,<0.22.0>}
```

```
2> ex1:foo(P, "Kalle", 96).
foo: name='Kalle', age=96
                          {ok,96}
3>
```

The attentive reader might find it difficult to notice any difference in this run of the sample `ex1` program. Indeed, there is no difference except for the starting method. In fact, readers may claim that we have not yet proven conformance to the supervision principles. To this end, we will switch on the SASL trace facility, which is a higher level trace than simple Erlang raw trace.

```
3> sys:trace(P, true).
ok
4> ex1:foo(P, "Kalle", 96).
*DBG* <0.22.0> got call {10,{"Kalle",96}} from <0.21.0>
*DBG* <0.22.0> sent #Bin to {port,#Port}
foo: name='Kalle', age=96
                          *DBG* <0.22.0> got port data 96 from #Port
*DBG* <0.22.0> sent {<0.22.0>,{data,96}} to <0.21.0>
{ok,96}
5>
```

The `sys:trace` function call propagates to the `erl_c_trace` hook. If we had defined the hook we would have the opportunity to turn on some trace flags in the C code. Note that we did not have to recompile anything for this example to work.

1.5 Advanced IG Topics

This section describes the more advanced features of the Interface Generator and includes the following topics:

- IG generator switches
- Socket example
- Using C macros
- IG files
- Inside IG.

IG Generator Switches

The generator accepts a few switches which guide the stub generation process.

cback_mod The `cback_mod` switch sets the name of the Erlang module where all callbacks from C will be found. For example: `ig:gen(ex1, [{cback_mod, my_cb_mod}])`. The name of the callback module defaults to `_cb` appended to the input file name.

nouse_stdio IG stubs normally use `stdin` and `stdout` (which are file descriptors 0 and 1) for port communication, but this switch assigns file descriptors 2 and 3 instead.

Sockets

IG is capable of running its communication over sockets. Sockets provide the advantage of using a server implemented on a different machine where ports are local, and the socket implementation enforces a more stringent use of the communication protocol. IG allows the C program to behave both as a server using calls from Erlang to C, and as a client using calls from C to Erlang. This could lead to confusion in the communication if both concepts are used at the same time. We could find ourselves in a race condition where both sides simultaneously decide to initiate some function calls. Both sides would then treat calls from the other side as the return value to their call (see figure below). We need not explain why this is unlikely to produce correct behaviour.

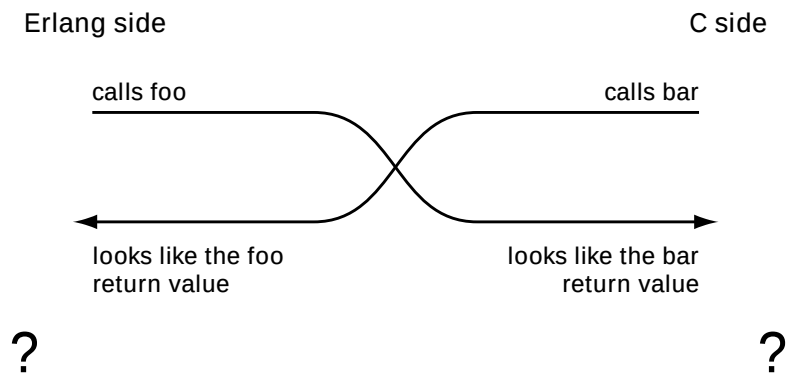

Figure 1.3: Example of Race Condition

This problem can be solved by using two communication links between Erlang and C. One link is used for messages initiated from Erlang, and the other for messages initiated from C. However, it is difficult to have two ports to the same C program but very easy to have two sockets. Therefore, socket communication is preferred if calls are initiated from both parties.

Socket communication requires the following additions:

- more standard C libraries are linked into the executable
- some IG socket routines are added to the C executable (the file `igsock.o`)
- the main loop is replaced with one that can handle sockets.

The `igsock.o` file is included in the distribution and some examples of main loops are included in the examples directory. The new linker command line must be augmented with `-lsocket -lnsl -lresolv`.

> **Note:**
> There is an `erl_interface_nodns` library for users who have no DNS service running.

IG generated socket communication is started differently to ordinary port communication. There are two new start functions, `client/2` and `server/1`. The former is used when the Erlang side is a client and the C side acts as a server. The latter is used when the Erlang side is a server to the C side. The `client/2` function takes two arguments, the first is the host name where the C program is running, and the second is the port number at which the C program is expecting to communicate. The `server/1` only takes a port number as an argument and it is up to the C program to find the correct host where the Erlang stub is running. This is all basic socket programming practice.

Socket Example

This example is based on the previous ex1 example in this section. We will add some libraries at link time and then start the Erlang stub in a different way than port start. We also have to start the C program manually from the Unix prompt. We will use two Unix shells, one for the C program and one for the Erlang program.

The first step is to make a new executable with other libraries than port operation, and with a different main loop. The igsock_server main loop is taken from the examples directory.

```
UNIX 1> gcc -L. -o ex1 ex1.c ex1_stub.c igio.o igsock.o
igsock_server.c -lsocket -lnsl -lresolv -lerl_interface
```

We now start the program with a random port number. In our case, we choose 6758 but any free port number will do.

```
UNIX 1> ./ex1 6758
```

Nothing happens at this point, because the C program waits for Erlang to connect to the port. We start Erlang in the usual way and start the IG stub as a client to the C server. We then make the usual call to foo just to make sure that it works. Note the use of net_adm:localhost/0 which returns the host name as a string. We are in fact connecting to a socket which can be on any machine in the world.

```
UNIX 2> erl
Erlang (JAM) emulator version 4.4.2

Eshell V4.4.2  (abort with ^G)
1> P = ex1:client(net_adm:localhost(), 6758).
<0.23.0>
2> ex1:foo(P, "Andrew", 55).
{ok,55}
3> ex1:stop(P).
ok
```

The call to foo also results in a printout from C. In previous examples this has been printed in the Erlang shell, but this time it is printed in the Unix shell used for the C program. A printout of the type shown below appears in the Unix shell where the C program was started.

```
foo: name='Andrew', age=55
```

Two things are worth mentioning about this example:

- We did not have to regenerate any code, we did not even have to recompile the Erlang stub.
- We only had to replace the main loop and add some socket related libraries to compile the C program.

Using C Macros

IG supports a convenient subset of the standard C macros and pre-processor directives. Macros can be used for declaring constant values, and pre-processor directives can be used for conditional declaration of macros. Macros and conditionals are generated to an Erlang header file and the mapping is straight forward. Therefore, the conditionals do not affect the generation of stubs because they are only generated to the Erlang header file. It is not possible to conditionally declare interface functions in header files intended for IG.

The IG pre-processor directives are only generated to the `.hrl` file, not the `.erl` file.

/*IG*/ #define <name> [<const_value> .] Puts a `-define( <name>, <const_value> ).` or a `-define( <name>, true ).` in the `.hrl` file. The second form is used when the constant value is omitted. The constant value can be either an integer, a boolean value, a string, or a previously defined macro name. This makes it easy to use C macro values in Erlang.

/*IG*/ #ifdef <name>. This works like ordinary C style pre-processor conditional directives. Puts an `-ifdef( <name> ).` in the `.hrl` file. Useful for conditional declarations of macros in the Erlang header file (`.hrl`).

/*IG*/ #ifndef <name> Puts an `-ifndef( <name> ).` in the `.hrl` file.

/*IG*/ #else Puts an `-else.` in the `.hrl` file.

/*IG*/ #endif Puts an `-endif.` in the `.hrl` file.

In the following example, we have the input file `pre.h`:

```
/*IG*/ #define FWRITE 1
/*IG*/ #define FREAD  2
FILE* open_file(char* fname, int mode); /* where mode is FWRITE or FREAD */
```

This will be translated to the following Erlang header file:

```
-define(FWRITE, 1).
-define(FREAD, 2).
```

This header file enables the Erlang programmer to write Erlang code which looks as follows:

```
-module(nils).
-export([n/0]).
-include("pre.hrl").
n() ->
    P = pre:start(),
    Fd = pre:file_open(P, "cake", ?FWRITE).
```

> Note:
> The Erlang code shown above is very similar to an equivalent C code.

IG Files

IG needs some magic files to work, namely the IO routines, the main loop and the value representation. The IO routines are contained in the file `igio.o` and the value representation is contained in the `erl_interface` library. The value representation is a C view of the Erlang values, but it is not the representation used within the emulator.

IG also comes with an example main loop in the file `igmain.o`. It is not mandatory because the main loop may need to be replaced when the method of communication changes . The main loop in the distribution supports port communication only but there are some examples of socket communication main loops in the examples directory. We have made a fair attempt to provide IO and main loops that fit large areas of usage, but we recognize that we cannot cover all territories.

Inside IG

This section attempts to explain how IG really works. It contains details about the implementation and process structure. The reader is encouraged to read generated stub code and the `igserver.erl` file for reference. This information is not a requirement for using IG correctly and can be skipped at the discretion of the reader.

When an IG generated program is set up, one process is started which is in charge of the port at the Erlang side, and a C program is started which can handle the communication on the port. The Erlang process runs a module called `igserver` which handles marshalling of arguments and all port communication. It is the `igserver` process that recognizes the standard Erlang system messages which implement the software manager functions. `igserver` also handles communication link details.

A Real Function Call

We will now follow a function call from Erlang to C. First, an IG generated interface stub function is called from a user process. Each interface function from the header file is given a number at generate time and this number is used when the stub function communicates with `igserver`, along with any arguments. The stub function passes a message to the `igserver` stating the function number and any parameters.

The `igserver` then forwards this request for a function call to the C side where it will first be received by the IG IO routines, decoded using the `erl_interface` library, and finally presented to the IG generated C stub dispatch function. The dispatcher will then extract the function number and use this number as an index in its array of function pointers. The pointers in the array point to specially generated intermediate functions, one for each stub function. These functions unpack all arguments, make the actual function call to the user function, and then pack the return value and send it back to Erlang.

The `igserver` then receives its answer which it sends back to the original user process. All this is illustrated in the figure below.

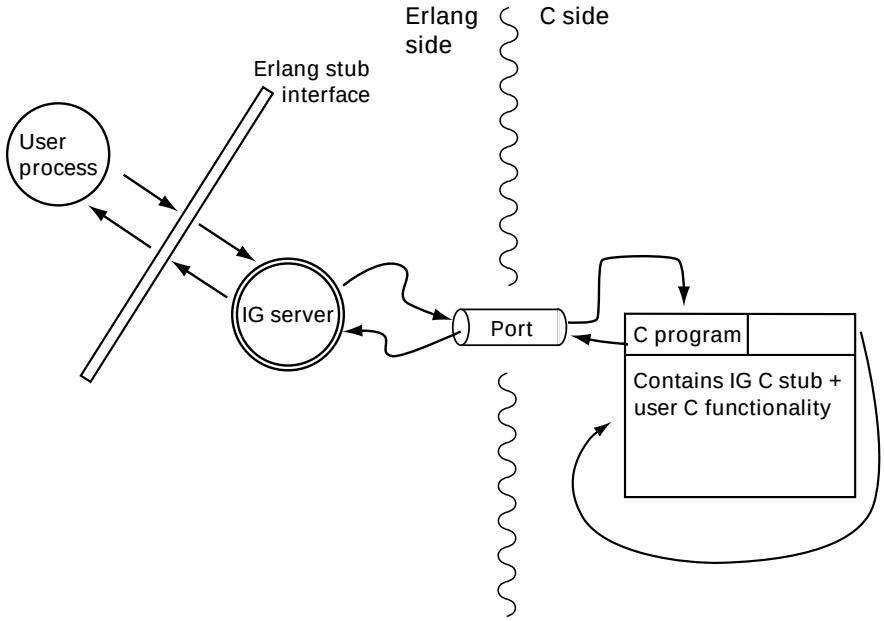

Figure 1.4: The IG Process

IG Reference Manual

Short Summaries

- Erlang Module ig [page 22] – The Interface Generator

ig

The following functions are exported:

- `ig:gen(FileName [ , Options ] ) -> Result`
  [page 22] Generate stub code to interface Erlang/C

ig (Module)

The Interface Generator (IG) is used to make C functions accessible from Erlang, and vice versa.

The input specification to IG consists of a `.h` ANSI C header file, in which the function prototypes and relevant data types are specified. When the code is generated, two or three files with stub code are created: one Erlang file, one C file, and an optional Erlang header file.

The actual communication between Erlang and the C program is handled by the `open_port/2` mechanism, or by using *sockets*. The method used is decided at application start-up by choosing either the generated `start/1` function, or one of the `client/1` or `server/2` functions.

The generated Erlang code has functions which correspond to the C functions to be accessed. The generated C stub code must be linked to an executable. It can either be linked with user provided code, or with the standard modules included with IG. Depending on what was specified in the input file to IG, the stub code contains routines for calling C and/or Erlang functions.

Exports

`ig:gen(FileName [ , Options ] ) -> Result`

Types:

- Result = ok | {error,Reason}
- Filename = Module | ”Module” | ”Module.h”
- Options = {cback_mod,CbackMod} | nouse_stdio | {file,IncFile}
- CbackMod = Module (location of the Erlang call-back functions)
- nouse_stdio = File descriptor 3 and 4 will be used in the open_port/2 command
- IncFile = Module (additional include file to be used)
- Module = Atomic module name

If no CbackMod is specified, it is assumed that the call-back functions are located in a module: <Filename>_cb.erl

Example:

```
ig:gen(xemtP, [nouse_stdio, {file, xmainwindow}, {file, unix}]).
```

The Grammar of the .h file

Pre-process directives and comments are ignored by IG. This means, for example, that any declarations you may have in other `include` files will be ignored. This is also true for conditional pre-processor directives, such as `#ifdef`, which might still be very useful in order to avoid type conflicts.

The actual syntax of the C subset which IG accepts is described below in a BNF like form.

Typedef- and struct statements.

```
TYPEDEF_STMT ::= typedef struct OPT_TAG '{' MEMBER_LIST '}'
                   STD_TYPEDEFNAME ';' |

                 typedef STD_TYPE_SPEC STD_TYPEDEFNAME ';'

STRUCT_STMT  ::= struct IDENTIFIER '{' MEMBER_LIST '}' ';'
```

Variable statements.

```
VAR_STMT  ::= 'IG_var' DECLARATION STD_IDENTIFIER ';' |

              'IG_var' OPT_STORAGE_CLASS STRUCT IDENTIFIER
                STD_IDENTIFIER ';' |

              'IG_var' IDENTIFIER STD_IDENTIFIER ';'
```

Preprocessor directives.

```
HDEF_STMT ::= 'IG_define' IDENTIFIER [HDEF_CONST]? ';' |

              ['IG_ifdef'|'IG_ifndef'] IDENTIFIER ';' |

              ['IG_else'|'IG_endif'] ';'
```

Function prototypes.

```
PROTOTYPE_STMT ::= IG_KEYWORD DECLARATION STD_IDENTIFIER
                     '(' PARAM_DECLARATIONS ')' ';' |

                   IG_KEYWORD struct IDENTIFIER STD_IDENTIFIER
                     '(' PARAM_DECLARATIONS ')' ';' |

                   IG_KEYWORD 'IG_binaryPtr' IDENTIFIER
                     '(' PARAM_DECLARATIONS ')' ';' |

                   IG_KEYWORD IDENTIFIER STD_IDENTIFIER
                     '(' PARAM_DECLARATIONS ')' ';'
```

The rest of the syntax.

```
MEMBER_LIST  ::= MEMBER_DECLARATION |
                 MEMBER_LIST MEMBER_DECLARATION

MEMBER_DECLARATION ::= STD_TYPE_SPEC MEMBER_NAME |
                       'IG_binaryPtr' STD_IDENTIFIER ';'

MEMBER_NAME  ::= STD_IDENTIFIER ';' |
                 STD_IDENTIFIER '[' CONSTANT_EXPRESSION ']' ';'

PARAM_DECLARATIONS ::= PARAM_DECL |
                       PARAM_DECLARATIONS ',' PARAM_DECL

PARAM_DECL   ::= STD_TYPE_SPEC STD_OPT_IDENTIFIER |

                 IDENTIFIER STD_OPT_IDENTIFIER |

                 'IG_binaryPtr' STD_OPT_IDENTIFIER |

                 struct IDENTIFIER STD_OPT_IDENTIFIER

STD_TYPEDEFNAME  ::= ['*'|TYPE_QUALIFIERS]? TYPEDEFNAME

STD_IDENTIFIER   ::= ['*'|TYPE_QUALIFIERS]? IDENTIFIER

HDEF_CONST ::= INTEGERCONSTANT | FLOATINGCONSTANT |
               CHARACTERCONSTANT | OCTALCONSTANT |
               HEXCONSTANT | STRING | IDENTIFIER

CONSTANT_EXPRESSION
             ::= "An arithmetic expression which can be
                  evaluated at compile time. For example:
                  2*(64+2) ."

DECLARATION  ::= "Basically, a non-aggregate, ANSI C
                  variable declaration."

IDENTIFIER   ::= "An identifier is a sequence of letters,
                  digits, and underscores. An identifier
                  must not begin with a digit, and it must
                  not have the same spelling as a reserved
                  word."

TYPEDEFNAME  ::= "An identifier which is currently in
                  scope as a typedef name"
```

See Also

For more detailed information see the IG User's Guide.

List of Figures

Index

Modules are typed in *this way*.
Functions are typed in `this way`.