Getting Started with Erlang

version 5.2

Typeset in LaTeX from SGML source using the DOCBUILDER 3.2.2 Document System.

Contents

Getting Started with Erlang

Chapter 1

Getting Started with Erlang

1.1 Getting Started with Erlang

1.1.1 Introduction

This chapter describes the most common tasks users perform in Erlang/OTP and is intended for readers who are running Erlang/OTP for the first time. It also contains references to more detailed information.

This manual assumes that you have another source of information about how the language Erlang works. The first chapter of the book *Concurrent Programming in ERLANG*, [*Concurrent Programming in ERLANG* [1]], is available online in PDF[1] format.

This manual will describe how to

- compile and run Erlang programs
- work with the *Erlang shell*, this is the most central part of the development environment
- use the basic tools, Debugger and Pman

This manual will also describe differences in development environment on the Unix and Windows platforms.

Manual conventions

The following conventions are used to describe commands, which are entered from the keyboard in the Erlang shell, or on the Emacs command line: window-based

- to enter the command *C-c*: Press the *Control key* and the letter *c* simultaneously. *C-c* is equivalent to ˆc.
- to enter the command *M-f*: Press and release the *Esc key*, then press the letter *f*.
- to enter the command *halt( ).* Enter halt( ). and then press the *Return* key. You must include the full stop that terminates the command.
- the Unix prompt is >.
- the DOS prompt is *C:\>*

---

[1]URL: `http://www.erlang.org/download/erlang-book-part1.pdf`

1.1.2   Working with Erlang

Erlang programs are executed when you instruct the *Erlang Runtime System* (ERTS) to execute your code. They do not run as binary programs, which can be executed directly by your computer. ERTS consists of an Erlang evaluator and some libraries. The Erlang evaluator is often referred to as an *emulator* and is very similar to the Java virtual machine.

ERTS together with a number of ready-to-use components and a set of design principles forms the *Open Telecom Platform*, normally called OTP or Erlang/OTP.

The most central part of the Erlang/OTP development environment is the *Erlang shell, erl.* erl is available on Unix systems, Windows NT and Windows 95. It looks very similar to a Unix shell, Windows DOS box or a Windows NT Command box. The difference is that the Erlang shell understands how to

- compile and load Erlang programs
- run Erlang programs, individual functions or evaluate Erlang expressions
- monitor and control execution of the programs
- command line editing
- history list with previously entered commands

In addition to this text-based interface there are several window-based *tools* like Debugger, Pman and TV.

You can access everything from this command interface but you can also activate the graphic Toolbar where you get buttons to start the window based tools.

Running Erlang/OTP

To start Erlang/OTP on a Unix system you execute in a Unix shell

```
> erl -s toolbar
```

On the Windows platform you should find Erlang/OTP in the start menu, use the entry `Erlang with Toolbar`. If the Erlang/OTP executable is in your execution path you can also start Erlang/OTP from a DOS box or NT command line with

```
C:\> werl -s toolbar
```

> **Note:**
> On the Windows platform there are two executables, `erl` and `werl`. You normally use `werl`, see the *werl* reference manual for details.

The following apears on the screen:

```
Erlang (BEAM) emulator version 4.9

Eshell V4.9  (abort with ^G)
1>
```

The Erlang shell is now ready for you to type some input. The commands you enter could be any Erlang expression, which is understood by an Erlang shell.

Try it out by typing `1 + 2.` followed by pressing the return key (don't forget the terminating dot). You can also call one of the built-in functions, type `date()..` The result from what you entered should make it look like

```
Erlang (BEAM) emulator version 4.9

Eshell V4.9  (abort with ^G)
1> 1 + 2.
3
2> date().
{1998,1,22}
```

The example above can be regarded as having a dialog with the runtime system. You can also run functions or programs, send messages or monitor processes.

To stop Erlang/OTP, enter one of the following commands in the window where Erlang/OTP was started:

1. Enter the command *C-c* (press the *Control key* and the letter *c* simultaneously).
   This command puts Erlang/OTP into BREAK mode, and the system responds as follows:

   ```
   BREAK: (a)bort (c)ontinue (p)roc info (i)nfo (l)oaded
   (v)ersion (k)ill (D)b-tables (d)istribution
   ```

   Enter *a* and then press the *Return key* to abort Erlang/OTP.

2. Enter the command *halt( ).* and then press the *Return key*.

3. Enter the command *C-g* and then press the *Return key*. This command returns with *->* press *q* to quit and then press the *Return key* to abort Erlang/OTP.

Starting Tools

You can start various tools from the toolbar

Figure 1.1: The Toolbar

Compiling a File

This section describes how to compile and run a simple Erlang program, using the Erlang shell. It is assumed that the following program, which calculates the factorial of an integer, has been created with an external editor and saved with the file name `math1.erl`.

Make sure that you are in the directory, where the program file is stored. Use the command `pwd().` to check this and, if needed, change directory by using the command `cd(Dir)..`

```
-module(math1).
-export([factorial/1]).
factorial(0) -> 1;
factorial(N) -> N * factorial(N-1).
```

Follow these steps to compile the program:

1. Enter the command `c(math1)..`

   ```
   1> c(math1).
   {ok,math1}
   2>
   ```

2. The system returns `{ok, math1}` if the compilation was successful. If unsuccessful, the compiler prints an error message in the following format:

   ```
   math1.erl: line number: error message
   ```

Refer to the section Compiling [page 13] in this chapter for more information about compiling code.

Evaluating an Expression

When a module has been successfully compiled, the exported functions in the module can be evaluated by entering the expression at the Erlang shell prompt, followed by a full stop and a carriage return.

The following example illustrates an evaluation, using the `math1.erl` program:

- Enter the following expression at the command prompt:

  ```
  math1:factorial(45).
  ```

- Erlang evaluates the expression and prints the result. The result from the expression entered looks as follows:

  ```
  2> math1:factorial(45).
  119622220865480194561963161495657715064383733760000000000
  3>
  ```

Note:
All Erlang expressions must end with a full stop.

Aborting Evaluation of an Expression

Enter *C-c.* to abort an evaluation (press the *Control key* and the letter *c* simultaneously). This action will also terminate an infinite loop.

How to get help

The Erlang language itself is described in the book *Concurrent Programming in ERLANG* , which is available online in PDF[2] format.

The complete documentation is part of the release in HTML format. The top of the documentation tree is located in the *doc* directory of the Erlang/OTP installation. It is also available on-line [3].

From an Unix shell you can access the manual pages for the Erlang/OTP libraries. As an example to display the manual page for the module titled `lists` you enter

```
% erl -man lists
```

You can list the shell commands from the Erlang shell if you enter

```
1> help().
```

There is lots of information at the commercial Erlang/OTP site http://www.erlang.se/[4]. and the OpenSource Erlang/OTP site http://www.erlang.org/[5].

1.1.3   The Erlang Shell

The Erlang Shell is a program, which provides a front-end to the Erlang evaluator. Edit commands, shell internal commands, and Erlang expressions, which are entered in the shell window, are interpreted by the Erlang evaluator.

In the shell it is possible to:

- evaluate expressions
- recall previous expressions
- create or destroy variable bindings
- manipulate concurrent jobs.

Depending on your preferred working environment, start Erlang/OTP in one of the following ways:

- On a Unix system you enter the command `erl` at the operating system prompt. This command starts the Erlang runtime system and the Erlang shell.
- On the Windows platform you normally start Erlang/OTP from the start menu. You can also enter the command `erl` or `werl` from a DOS box or Command box. Note that starting with `erl` will give you a more primitive Erlang shell than if you start with `werl`, see the *werl* reference manual for details.

---

[2]URL: `http://www.erlang.org/download/erlang-book-part1.pdf`
[3]URL: `http://www.erlang.org/doc/current/doc/`
[4]URL: `http://www.erlang.se/`
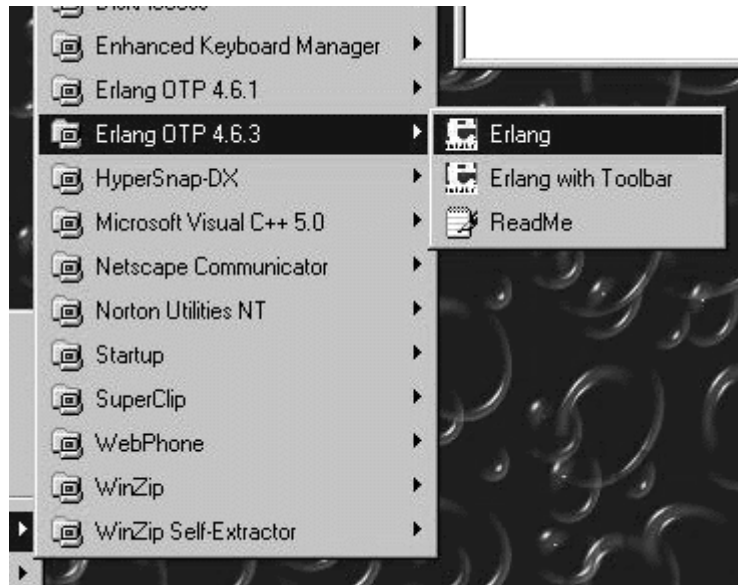[5]URL: `http://www.erlang.org/`

Figure 1.2: Erlang/OTP menu on Windows

Apart from evaluation commands and expression, the shell has the following properties:

- Bindings made to variables are remembered.
- Changes made to the local process dictionary (`get` and `put`) are remembered.
- New bindings and changes to the local process dictionary are lost if an expression is entered and its evaluation fails. In this case, the old values are restored.
- Jobs can be manipulated in the shell. A job can be thought of as a set of processes, which can communicate with the shell.

The shell can run in two separate modes:

- *Normal mode*, in which commands can be edited and expressions evaluated
- *Job control mode (JCL)*, in which jobs can be started, stopped, detached or connected. Only the current job can communicate with the shell.

Editing Commands

a sub-set of the *Emacs* line editing commands. These commands are listed in the following table.

Key Sequence	Function
C-a	Beginning of line
C-b	Backward character
M-b	Backward word
C-d	Delete character
M-d	Delete word

continued ...

... continued

C-e	End of line
C-f	Forward character
M-f	Forward word
C-g	Enter shell break mode
C-k	Kill line
C-l	Redraw line
C-n	Fetch next line from the history buffer
C-p	Fetch previous line from the history buffer
C-t	Transpose characters
M-t	Transpose words
C-y	Insert previously killed text

Table 1.1: Text Editing

Note:

The notation *C-a* means pressing the control key and the letter *a* simultaneously. *M-f* means pressing the *ESC* key followed by the letter *f*.

The *Esc* key is also called the *Meta* key in the Unix environment.

The `Tab` can be used for completion of module and function names.

Shell Internal Commands

The shell has a number of built-in internal commands. The command `help()` displays the following commands:

```
1> help().
** shell internal commands **
b()            -- display all variable bindings
e(N)           -- repeat the expression in query <N>
f()            -- forget all variable bindings
h()            -- history
history(N)     -- set how many old previous commands to keep
results(N)     -- set how many previous command results to keep
v(N)           -- use the value of query <N>
** commands in module c **
bt(Pid)        -- stack backtrace for a process
c(File)        -- compile and load code in <File>
cd(Dir)        -- cd Dir .. example cd("..")
flush()        -- flush any messages sent to the shell
help()         -- help info
i()            -- information about the system
ni()           -- information about the networked system
i(X,Y,X)       -- information about pid <X,Y,Z>
l(File)        -- Load module in <File>
```

```
lc([File])      -- Compile a list of Erlang modules
ls()            -- list files in the current directory
ls(Dir)         -- list files in directory <Dir>
m()             -- which modules are loaded
m(Mod)          -- information about module <Mod>
memory()        -- memory allocation information
memory(T)       -- memory allocation information of type <T>
nc(File)        -- compile and load code in <File> on all nodes
nl(File)        -- Load module in <File> on all nodes
pid(X,Y,Z)      -- convert X,Y,Z to a Pid
pwd()           -- print working directory
regs()          -- information about registered processes
nregs()         -- information about all registered processes
zi()            -- information about the system, including zombies
** commands in module i (interpreter interface) **
ih()            -- print help for the i module
true
2>
```

When an expression with the format `Func(Arg1,Arg2,...,ArgN)` is entered, the shell has to determine what `Func` refers to before the expression can be evaluated. It interprets `Func` in the following order of priority:

1. `Func` refers to a functional object (Fun)
2. `Func` refers to a built-in function (BIF)
3. `user_default:Func` in the module `user_default`
4. `shell_default:Func` in the module `shell_default`

Example Dialogue with the Shell

The following is an extended dialogue with the shell. Detailed explanations of the individual commands are included.

```
host_prompt> erl
Eshell V4.4 (abort with ^G)
1> Str = "abcd".
"abcd"
```

Command 1 sets the variable `Str` to the string `"abcd"`. Remember that `"..."` is the shorthand for the list of ASCII integers which represents the string.

```
2> L = length(Str).
4
```

Command 2 matches the pattern `L` against the return value of `length(Str)`. Since `L` is unbound the match succeeds and L is bound to 4.

```
3> Descriptor = {L, list_to_atom(Str)}.
{4,abcd}
```

Command 3 binds the value of variable `Descriptor` to the tuple `{4,abcd}`.

```
4> L.
4
```

Command 4 prints the value of the variable `L`.

```
5> b().
Descriptor = {4,abcd}
L = 4
Str = "abcd"
ok
```

Command 5 evaluates the shell internal command `b()` (short for bindings). This prints a list of the current shell variables and their bindings.

```
6> f(L).
ok
```

Command 6 evaluates the shell internal command `f(L)` (short for forget) which forgets the value of the variable `L` .

```
7> b().
Descriptor = {4,abcd}
Str = "abcd"
ok
```

Command 7 returns a list of the current variable bindings. The variable `L` has been forgotten.

```
8> {L, _} = Descriptor.
{4,abcd}
```

Command 8 performs a pattern matching operation on `Descriptor`, binding a new value to `L`.

```
9> L.
4
```

Command 9 evaluates `L` and prints its value.

```
10> {P, Q, R} = Descriptor.
** exited: {{badmatch,{4,abcd}},{erl_eval,expr,3}} **
```

Command 10 tries to match {P, Q, R} against Descriptor which is {4, abc}. The match fails and none of the new variables are bound. "** exited: badmatch ... ... **" is printed. This is not the value of the expression, and the expression has no value since its evaluation failed. It is a warning printed by the system to inform you that an error has occurred. The information printed after the word badmatch indicates which term was being matched, and in which function the error occurred. The values of all other variables, such as L and Str are unchanged.

```
11> P.
** exited: {{unbound, 'P'},{erl_eval, expr, 3}} **
```

```
12> Descriptor.
{4,abcd}
```

Commands 11 and 12 show that P is unbound, as the previous command failed, and Descriptor has not changed.

```
13> {P, Q} = Descriptor.
{4,abcd}
```

```
14> P.
4
```

Commands 13 and 14 show successful matches, in which P and Q are bound.

```
15> f().
ok.
```

Command 15 clears all bindings.

For the next commands we use the module test1 defined in the following way:

```
-module (test1).
-export ([demo/1]).
demo(X) ->
put(aa, worked),
X = 1,
X + 10.
```

Compile it with c(test1). and continue with command 16

```
16> put(aa, hello).
undefined
17> get(aa).
hello
```

Commands 16 and 17 set and inspect the value of the item aa in the process dictionary.

```
18> Y = test1:demo(1).
11
19> get().
[{aa, worked}]
```

Command 18 evaluates `test1:demo(1)`, the evaluation succeeds and the changes made in the process dictionary become visible to the shell. The new value of the dictionary item `aa` can be seen in command 19.

```
20> put(aa, hello).
worked
21> Z = test1:demo(2).
!!! Error in process <0.17.0> with exit value:
{{badmatch,1},{test1,demo,[2]}}
** exited: {{badmatch,1},{test1,demo,[2]}} **
```

Commands 20 and 21 reset the value of the dictionary item `aa` to hello and calls `test1:demo(2)`. The evaluation fails, and the changes made to the dictionary in `test1:demo(2)` before the error occurred are discarded.

```
22> Z.
** exited: {{unbound,'Z'},{erl_eval,expr,3}} **
23> get(aa).
hello
```

Commands 22 and 23 show that Z was not bound and that the dictionary item `aa` retained its original value.

```
24> erase(), put(aa, hello).
undefined.
25> spawn(test1, demo, [1]).
<0.15.1>
26> get(aa).
hello
```

Commands 24, 25, and 26 show the effect of evaluating `test1:demo(1)` in the background. In this case the expression is evaluated in a newly spawned evaluator and any changes made in the process dictionary are local to the newly spawned process and therefore not visible to the shell.

```
27> io:format("hello hello\n").
hello hello
ok
28> e(27).
hello hello
ok
29> v(27).
ok
```

Commands 27, 28, and 29 use the history facilities of the shell. Command 28 is `e(27)`, which re-evaluates command 27. Command 29 is `v(27)` which uses the value (result) of command 27. In the case of a 'pure' function, i.e. a function with no side effects, the result is the same. For a function with side effects the result can be different.

In the next command we use the module test2 defined in the following way:

```
-module (test2).
-export ([loop/1]).

loop(N) ->
io:format('Hello Number:~w\n', [N]),
loop(N+1).
```

Compile it with `c(test2).` and continue with command 30.

```
30> test2:loop(0).
Hello Number:1
Hello Number:2
Hello Number:3
Hello Number:4
Hello Number:5
User switch command
-->
```

Command 30 evaluates `test2:loop(0)`, which puts the system into an infinite loop. The user typed `Control G`. This suspends output from the current process, which is in a loop, and enters JCL mode [page 12]. In JCL mode you can start and stop jobs.

Type *q* to exit the Erlang shell, then open a new shell and test following commands:

```
1> F = fun(X) -> 2*X end.
#Fun<erl_eval>
2> F(2).
4
```

Command 1 binds the variable `F` to a *fun*, and command 2 evaluates the expression `F(2)`.

Advanced Shell Usage - Job Control Mode

When the shell starts, it starts a single evaluator process. This process, together with any local processes, which it spawns, is referred to as a *job*. Only the current job, which is said to be *connected*, can perform operations using standard I/O. All other jobs, which are said to be *detached*, will be *blocked* if they attempt to use standard I/O. Jobs which do not use standard I/O run in the normal way.

Typing `Control G` detaches the current job, and the shell enters into JCL mode. The prompt changes and looks as follows:

```
-->
```

Typing `?` at the prompt gives the following help message:

```
--> ?
c [nn]   - connect to the current job
i [nn]   - interrupt job
k [nn]   - kill job
j        - list all jobs
s        - start new job
r [node] - start a remote shell
q        - quit erlang
? | h    - help
```

The *JCL* commands have the following meaning.

Command	*Function*
c	Connect to the current job. The standard shell will be resumed. Operations, which use standard I/O, will be interleaved with user inputs to the shell.
c [nn]	As above, but only for job number nn.
k	Kill the current job. All spawned processes in the job will be killed provided they have not evaluated the group_leader/2 primitive, and provided they are located on the local machine. Processes spawned on remote nodes will not be killed.
k [nn]	As above but specifically only for job number nn.
j	List all jobs. A list of all known jobs is printed. The current job name is prefixed with '*'.
s	Start a new job. This will be assigned a new index [nn], which can be used for referencing.
r [node]	Start a remote job on node. This is used in distributed Erlang to allow a shell running on one node to control a number of applications running on a network of nodes.
q	Quit Erlang/OTP.
?	h	Help.

Table 1.2: The JCL Commands

1.1.4 Compiling

A file is compiled by evaluating the function

```
compile:file(File, Options).
```

File is a file name, and Options is a list of compiler options. Refer to the Reference Manual for a complete list of Options.

You can also enter the command c(File) from the Erlang shell. This command assumes that the following conditions apply:

1. The module name is the same as the file name, minus the .erl extension.
2. Code is to be loaded immediately if the compilation succeeds.

The function make:all() can be used to compile many files. All files must be located in the same directory. Refer to the Reference Manual, function *make(3)* for full details.

Compiler Diagnostics

The Erlang compiler detects a number of different errors. These can be:

- syntax errors
- variable errors
- missing function errors.

The compiler also warns about potential errors.

Syntax Errors   Syntax errors are reported against syntactically incorrect Erlang code. For example, the following code fragment is incorrect and will be reported by the compiler as shown:

```
test(X, Y) ->
    case g(X) o
        Y -> 1;
        2 -> 2
    end.
```

The syntax word `of` in the case construction is incorrectly spelt. This produces the compiler diagnostic:

```
test1.erl: 10 : of expected before 'o'
```

This means that an error occurred in line 10 of the file `test1.erl`. The message, which follows the line number, describes the error.

Variable Errors   Two types of variable errors can be detected by the compiler. Variables can be *undefined*, or *unsafe*. In the following example, the variables `Y` and `Z` are undefined:

```
test(X) ->
    A = f(X, Y, Z),
    b(A, X).
```

When compiled, the following error messages are printed:

```
./test.erl:11: variable 'Y' is unbound
./test.erl:11: variable 'Z' is unbound
```

The reason for these errors are that the variables `Y` and `Z` are undefined.

The following example shows the use of unsafe variables:

```
test1(X) ->
    case f(X) of
        true -> Y = 1, Z = 2;
        false -> Y = 2
    end,
    g(Y, Z).
```

In this example, there are two possible execution paths in the case statement. In one case the variables Y and Z are bound, in the other case only the variable Y is bound. After the case statement, we can only be certain that Y is bound and the variable Z is said to be *unsafe*. Any subsequent use of Z will result in an unsafe variable error message. In this case the following compiler error is generated:

```
./test1.erl:9: variable 'Z' is unsafe
```

If no reference to Z is made after the case statement, the function is correct, as the following example shows:

```
test2(X) ->
    case f(X) of
        true -> Y = 1, Z = 2;
        false -> Y = 2
    end,
    g(Y).
```

For both test1 and test2 above, the compiler will issue a warning:

```
./test1.erl:9: Warning: variable 'Y' exported from ['case']
```

To avoid this warning and lessen the risk for unsafe variables, write the code like this instead:

```
test3(X) ->
    Y = case f(X) of
            true -> 1;
            false -> 2
        end,
    g(Y).
```

Missing Functions   The following module contains a missing function:

```
-module(test).
-export([test/1]).
test(X) -> try(X, abc).
```

When the module is compiled, it results in the following:

```
> c(test).
./test.erl:3: function try/2 undefined
error
```
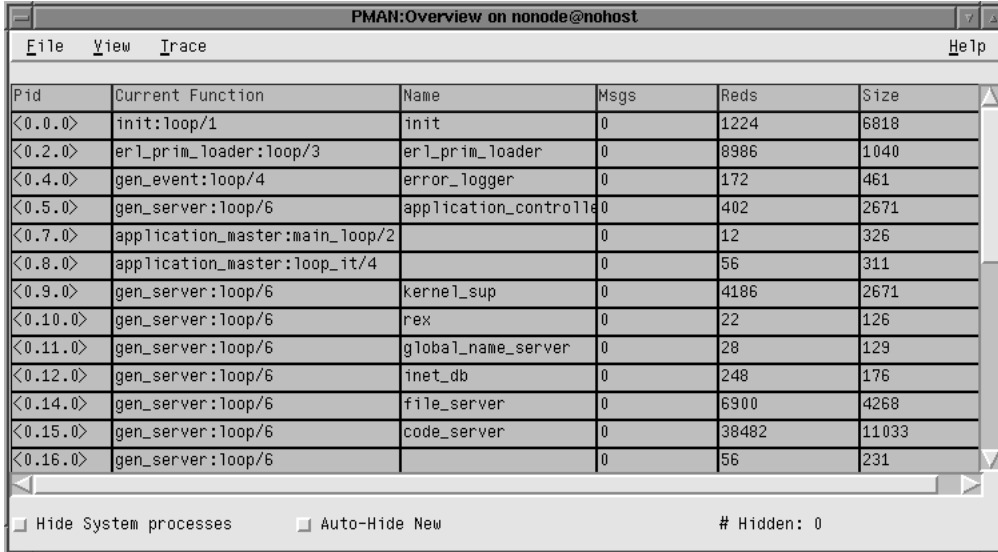
1.1.5   Debugging with Pman

The process manager, Pman, is a tool which traces the evaluation of code. It can trace several different processes, and several different nodes if distributed Erlang is used. Pman is controlled through a simple graphical user interface.

Start Pman as follows:

- run the function `pman:start()` from the Erlang shell, or
- push the `pman` button on the toolbar.

The following window is displayed when starting the Pman:

Pid	Current Function	Name	Msgs	Reds	Size
<0.0.0>	init:loop/1	init	0	1224	6818
<0.2.0>	erl_prim_loader:loop/3	erl_prim_loader	0	8986	1040
<0.4.0>	gen_event:loop/4	error_logger	0	172	461
<0.5.0>	gen_server:loop/6	application_controller	0	402	2671
<0.7.0>	application_master:main_loop/2		0	12	326
<0.8.0>	application_master:loop_it/4		0	56	311
<0.9.0>	gen_server:loop/6	kernel_sup	0	4186	2671
<0.10.0>	gen_server:loop/6	rex	0	22	126
<0.11.0>	gen_server:loop/6	global_name_server	0	28	129
<0.12.0>	gen_server:loop/6	inet_db	0	248	176
<0.14.0>	gen_server:loop/6	file_server	0	6900	4268
<0.15.0>	gen_server:loop/6	code_server	0	38482	11033
<0.16.0>	gen_server:loop/6		0	56	231

Figure 1.3: The Pman Window

This section describes a simple debugging example, which involves tracing the code in a single process.

For a more detailed description of Pman see the Pman User's Guide.

Example

We illustrate the use by tracing the evaluation of the function `ping_pong:ping`, which is defined as follows:

```
-module(ping_pong).
-export([ping/0, pong/0]).

ping() ->
    Pong = spawn(ping_pong, pong, []),
    Pong ! {self(), ping},
    receive
        pong ->
            pong
```

```
    end.

pong() ->
    receive
        {Ping, ping} ->
            Ping ! pong
    end.
```

Follow these steps to trace this process:

1. Compile the module.

2. Select the *Shell Process* item in the *Trace* menu of the Pman main window. The window shown in The Shell Tracer Window [page 17] is displayed.

3. From the shell tracer window select the *Options...* item in the *File* menu. Make sure the *Trace send* and *Trace receive* check boxes are selected and click *OK*.

4. Evaluate the function ping_pong:ping().

```
2> ping_pong:ping().
pong.
```

5. The contents of the shell trace window changes to show a listing of the called functions. See Trace of the function ping_pong:ping() [page 18].

```
┌──────────────────────────────────────────────────────────┐
│ ▬        PMAN:Shell process <0.21.0> on Node nonode@nohost  ▽ △│
├──────────────────────────────────────────────────────────┤
│  File   View   Trace                                  Help │
├──────────────────────────────────────────────────────────┤
│ initial_call: shell:evaluator/3                          △ │
│                                                            │
│ current_function: shell:eval_loop/2                        │
│                                                            │
│ heap_size: 34                                              │
│                                                            │
│ stack_size: 53                                             │
│                                                            │
│ reductions: 4                                              │
│                                                            │
│ links: [                                                   │
│   <0.20.0>]                                                │
│                                                            │
│ trap_exit: false                                           │
│                                                            │
│                                                          ▽ │
├──────────────────────────────────────────────────────────┤
│ ◁                                                        ▷ │
└──────────────────────────────────────────────────────────┘
```
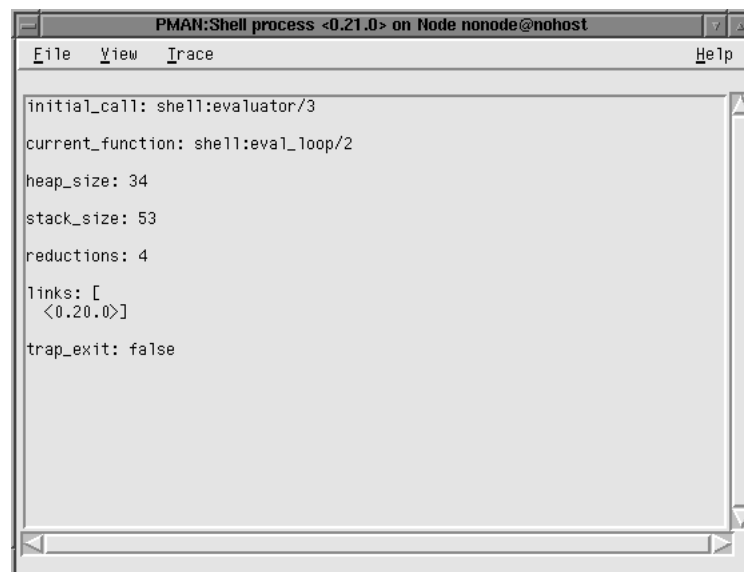
Figure 1.4: The Shell Tracer Window

```
PMAN:Shell process <0.25.0> on Node nonode@nohost

File    View    Trace                                    Help

current_function: shell:eval_loop/2

heap_size: 987

stack_size: 5

reductions: 438

links: [
  <0.22.0>]

trap_exit: false

<0.25.0>: getting_linked   <0.45.0>
<0.25.0>: rec    {shell_cmd, <0.22.0>, {eval, [{call,1,{remote,1,{atom,1,pi
<0.25.0>: spawn <0.48.0>
<0.25.0>:  !     To: <0.48.0> Msg: {<0.25.0>, ping}
<0.48.0>: rec    {<0.25.0>, ping}
<0.48.0>:  !     To: <0.25.0> Msg: pong
<0.25.0>: rec    pong
<0.48.0>: exit    normal
<0.25.0>:  !     To: <0.22.0> Msg: {shell_rep, <0.25.0>, {value, pong, [],
```

Figure 1.5: Trace of the function ping_pong:ping()

1.1.6   Distributed Erlang

Erlang can be used to write distributed applications, which run simultaneously on several computers. Programming distributed systems with Erlang is the subject of the book *Concurrent Programming in ERLANG* .

Nodes

A *node* is a basic concept of distributed Erlang. In a distributed Erlang system, the term node means an executing Erlang runtime system, which can communicate with other Erlang runtime systems. Each node has a name assigned to it, which is returned by the built-in function `node()`. This name is an atom and is guaranteed to be globally unique. The format of the name can vary between different implementations of Erlang, but it is always an atom which consists of two parts separated by an @ character. The following is an example of a node name:

```
dilbert@super.du.ericsson.se
```

Before we can fully account for the structure of a node name, we have to consider how host names are specified.

Host Names   Every host, which is attached to a network, has an IP address. However, it is not practical to use IP addresses when referring to hosts, since they are made up of numeric strings, which are difficult to remember. For this reason, hosts are usually given names.

Host names of hosts, which are connected to the Internet, are organized into *domains*. Domains are built hierarchically, with "top level" domains and "sub-domains". For example, the domain denoted by du.etx.ericsson.se is a sub-domain, which is part of another sub-domain etx.ericsson.se, which is part of the top domain se.

When it is clear from the context what domain a host belongs to, it is not necessary to use the fully qualified host name. In the example super.du.ext.ericsson.se, the first part "super" will suffice. We refer to "super" as the *short host name* of the host.

Hosts which are not connected to the Internet, and do not belong to any particular domain, are simply referred to by their short host name, "super" for example.

We use the term *host name* to denote the name of hosts. It should be clear from the context if we mean the long or the short form of the name.

Node Names   An Erlang node name is an atom constructed as the concatenation of a name supplied by the user, the @ character, and the name of the host where the Erlang runtime system is executed. For example:

```
dilbert@super.du.etx.ericsson.se
```

In this example, "dilbert" is the name of the Erlang node, and "super.du.etx.ericsson.se" is the name of the host where Erlang is running.

Distributed Erlang requires host names to be unique, since Erlang nodes identify each other by node names. We can achieve uniqueness for host names either by using long host names, or using short host names with the additional requirement that communicating hosts belong to the same domain (or a network not connected to Internet).

When using long host names, the corresponding node names are called *long node names*. For example:

```
dilbert@super.du.etx.ericsson.se
```

In this case, distributed Erlang is started with the `-name` flag. The system checks that the host name part of the node name contains at least one dot (.). If not, the host name is considered to be in error.

Similarly, when using short host names, the corresponding node names are called *short node names*. For example:

```
dilbert@super
```

In this case, distributed Erlang is started with the `-sname` flag.

> **Note:**
> An Erlang started with a long node name cannot communicate with a node started with a short node name.

It is often clear from the context if a node name is long or short. When this is the case, we use the term *node name* to refer to a node (the name of which may be short or long).

The first part of a node name which comes before the @ character is the *plain name* of the node. The plain name for a node is unique within the host on which the node is running. When starting distributed Erlang, with either the `-name` or the `-sname` flag, the plain name of the node is provided by the user. The other part of the node name, which is the host name part, is added by the Erlang runtime system at start-up.

Start-Up

Execute the following command to start a distributed Erlang node:

```
% erl -name foobar
Eshell V4.4 (abort with ^G)
(foobar@super.eua.ericsson.se)1>
```

The node name shown in the system response, includes the long name for the host. An alternative way is to start the system with the `-sname` flag. This is sometimes the only way to run distributed Erlang if the Domain Name System (DNS) is not running, or is not available.

Distribution Script Flags

The executable script *erl* starts the Erlang runtime system. Refer to `erl(1)`, for a complete list of the flags, which can be added to the start script. This section describes some of the flags, which control the distribution:

- `-name Name`
  This flag makes the node into a distributed node. It invokes all network servers which are needed to make a node a distributed node. Refer to `net_kernel(3)` for full details. The name of the node will be `Name@Host`, where `Host` is the fully qualified host name of the current host. This option will also ensure that the Erlang port mapper daemon (`epmd`) runs on the current host before Erlang is started. Refer to `epmd(3)`.

- `-sname Name`
  This flag is the same as the `-name` flag, except for the host name used. The name of the node will be `Name@Host`, where `Host` is the short host name of the current host. If the Domain Name System (DNS) is not running, this is sometimes the only way to run distributed Erlang. The `-sname` flag cannot communicate with systems running with the `-name` flag, since node names must be unique in distributed Erlang.

- `-setcookie Cookie`
  This flag sets the magic cookie of the current node to `Cookie`. Since `erlang:set_cookie(node(), Cookie)` is used, all other nodes are assumed to have their cookies set to `Cookie` as well. This way several nodes can share a common magic cookie (see the next section Security [page 21]).

- `-connect_all false`
  This flag causes the node connection graph to become 'leaner'. Only direct connections are established.

Security

Erlang nodes are protected by a magic cookie system. Refer to `auth(3)` for detailed information about the magic cookie system.

1. When a message is sent from one node to another node, the sending node attaches the magic cookie of the receiving node to the message.

2. The runtime system on the receiving node verifies that the magic cookie is the correct one for this node.

3. If the magic cookie is correct, the message is accepted. If the magic cookie is not correct, then the message is handled as follows:

   - The message is transformed into a "badcookie" message.

   - The "badcookie" message is sent to the system process `net_kernel`.

   - By default, the `net_kernel` passes the message to the registered process *auth*.

   - The `auth` process is then responsible for taking appropriate action for the unauthorized message. In the standard system, the default action is to shut down the connection to the sending node.

Magic cookies are assigned to a node as follows:

1. Initially, each node is assigned a random atom as its magic cookie, and each node assumes the atom `nocookie` for all other nodes.

2. When the standard `auth` server is started without the `-setcookie` flag, it reads a file named `$HOME/.erlang.cookie`. If the file `$HOME/.erlang.cookie` does not exist, it is created with a random string as its content. The permission mode of the file is set to `read-only` by owner.

3. The system creates an atom from the contents of this file and the function `erlang:set_cookie(node(), CookieAtom)` sets the magic cookie for the node to the value of this atom.

An Erlang node is completely unprotected when running `erlang:set_cookie(node(), nocookie)`. This can sometimes be appropriate for systems that are not normally networked, or for systems which are run for maintenance purposes only. Refer to `auth(3)` for details on the security system.

Slave Nodes

The Erlang/OTP library module `slave` provides functionality to start or control slave nodes. This functionality is Unix specific.

The remote nodes are started by means of the Unix command `rsh`. However, to use this command, the user must not only use a Unix system, but must also be allowed to `rsh` to the remote hosts without being prompted for a password.

Refer to the following manuals for additional information:

- The Unix documentation set, `rsh(1)`

- `slave(3)`.

1.1.7   Configuration

The standard Erlang/OTP system can be re-configured to change the default behavior on start-up.

The .erlang Start-up File

When Erlang/OTP is started, the system searches for a file named `.erlang` in the directory where Erlang/OTP is started. If not found, the user's home directory is searched for an `.erlang` file.

If an `.erlang` file is found, it is assumed to contain valid Erlang expressions. These expressions are evaluated as if they were input to the shell.

A typical `.erlang` file contains a set of search paths, for example:

```
io:format("executing user profile in HOME/.erlang\n",[]).
code:add_path("/home/calvin/test/ebin").
code:add_path("/home/hobbes/bigappl-1.2/ebin").
io:format(".erlang rc finished\n",[]).
```

user_default and shell_default

Functions in the shell which are not prefixed by a module name are assumed to be functional objects (Funs), built-in functions (BIFs), or belong to the module `user_default` or `shell_default`.

To include private shell commands, define them in a module `user_default` and add the following argument as the first line in the `.erlang` file.

```
code:load_abs("..../user_default").
```

erl

If the contents of `.erlang` are changed and a private version of `user_default` is defined, it is possible to customize the Erlang/OTP environment. More powerful changes can be made by supplying command line arguments in the start-up script `erl`. Refer to `erl(1)` and `init(3)` for further information.

List of Figures

List of Tables

Bibliography

[1] J. Armstrong, R. Virding, C. Wikstrom, M. Williams, Concurrent Programming in ERLANG, Prentice Hall, 1996, ISBN 0-13-508301-X.