Erlang Run-Time System Application (ERTS)

version 4.8

Magnus Fröberg

1997-05-02

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

Contents

Chapter 1

ERTS User's Guide

The Erlang Run-Time System Application (*ERTS*).

1.1 tty - A command line interface

`tty` is a simple command line interface program where keystrokes are collected and interpreted. Completed lines are sent to the shell for interpretation. There is a simple history mechanism, which saves previous lines. These can be edited before sending them to the shell. `tty` is started when Erlang is started with the command:
erl

`tty` operates in one of two modes:

- *normal mode*, in which lines of text can be edited and sent to the shell.
- *shell break* mode, which allows the user to kill the current shell, start multiple shells etc. Shell break mode is started by typing *Control G*.

Normal Mode

In normal mode keystrokes from the user are collected and interpreted by `tty`. Most of the *emacs* line editing commands are supported. The following is a complete list of the supported line editing commands.
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`.

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

continued ...

... continued

C-t	Transpose characters
C-y	Insert previously killed text

Table 1.1: tty text editing

Shell Break Mode

tty enters *shell* break mode when you type *Control G*. In this mode you can:

- Kill or suspend the current shell
- Connect to a suspended shell
- Start a new shell

ERTS Reference Manual

Short Summaries

epmd

The following functions are exported:

- `epmd [-daemon]` Starts a name server as a daemon
- `epmd -names` Requests the names of the registrered Erlang nodes on this host
- `epmd -kill` Kills the `epmd` process
- `epmd -help` List options

erl

The following functions are exported:

- `erl <script-flags> <user-flags>` Starts the Erlang system

erlc

The following functions are exported:

- `erlc flags file1.ext file2.ext...` Compiles files

erlsrv

The following functions are exported:

- `erlsrv {set | add} <service-name> [<service options>]` Adds or modifies an Erlang service
- `erlsrv {start | stop | disable | enable} <service-name>` Manipulates the current service status.
- `erlsrv remove <service-name>` Removes the service.
- `erlsrv list [<service-name>]` Lists all erlang services or all options for one service.
- `erlsrv help` Displays a brief help text

start_erl

The following functions are exported:

- `start_erl [<erl options>] ++ [<start_erl options>]` Start the Erlang emulator with the correct release data

werl

No functions are exported

erl_set_memory_block

The following functions are exported:

- `int erl_set_memory_block(size_t size, void *ptr, int warn_mixed_malloc, int realloc_always_moves, int use_reclaim, ...)` Specifies parameters for Erlang internal memory allocation.
- `int erl_memory_show(...)` A utility similar to VxWorks `memShow`, but for the Erlang memory area.
- `int erl_mem_info_get(MEM_PART_STATS *stats)` A utility similar to VxWorks `memPartInfoGet`, but for the Erlang memory area.

epmd (Command)

This daemon acts as a name server on all hosts involved in distributed Erlang computations. When an Erlang node starts, the node has a name and it obtains an address from the host OS kernel. The name and the address are sent to the `epmd` daemon running on the local host. In a TCP/IP environment, the address consists of the IP address and a port number. The name of the node is an atom on the form of `Name@Node`. The job of the `epmd` daemon is to keep track of which node name listens on which address. Hence, `epmd` map symbolic node names to machine addresses.

The daemon is started automatically by the Erlang start-up script.

The program `epmd` can also be used for a variety of other purposes, for example checking the DNS (Domain Name System) configuration of a host.

Exports

`epmd [-daemon]`

Starts a name server as a daemon. If it has no argument, the `epmd` runs as a normal program with the controlling terminal of the shell in which it is started. Normally, it should run as a daemon.

`epmd -names`

Requests the names of the local Erlang nodes `epmd` has registered.

`epmd -kill`

Kills the `epmd` process.

`epmd -help`

Write short info about the usage including some debugging options not listed here.

Logging

On some operating systems *syslog* will be used for error reporting when epmd runs as an daemon. To enable the error logging you have to edit /etc/syslog.conf file and add an entry

```
!epmd
*&ld;TABs&gt;/var/log/epmd.log
```

where <TABs> are real tab characters. Spaces will silently be ignored.

erl (Command)

The Erlang system is started with the following command:

`erl <script-flags> <user-flags>`

The executable script `erl` is a front-end to the Erlang runtime system. The flags for this program are described below.

Windows 95/NT users will probably want to use the `werl` program instead, which uses its own window and supports command-line editing and has scrollbars. The `erl` program on Windows provides no line editing in its shell and on Windows 95, there is no scroll back to see text which has scrolled off the screen. The `erl` program must be used, however, in pipelines or if you want to redirect standard input or output.

Exports

`erl <script-flags> <user-flags>`

Starts the Erlang system. The `erl` script pre-processes some of the `<script-flags>` and passes the flags to the `init` process in the Erlang system. The `<script-flags>` argument is used to initialize the Erlang kernel processes.

The `<user-flags>` argument is also passed to the `init` process and can be accessed through the `init` module in the running system. See init(3).

Script Flags

The following script flags are supported:

-AppName Key Value Overrides the `Key` configuration parameter of the `AppName` application. See application(3).

-boot File Specifies the name of the boot script, `File`.boot, which is used to start the system. See init(3). Unless `File` contains an absolute path, the system searches for `File`.boot in the current and `<ERL_INSTALL_DIR>/bin` directories .

If this flag is omitted, the `<ERL_INSTALL_DIR>/bin/start.boot` boot script is used.

-boot_var Var Directory [Var Directory] If the boot script used contains another path variable than $ROOT, this variable must have a value assigned in order to start the system. A boot variable is used if user applications have been installed in another location than underneath the `<ERL_INSTALL_DIR>/lib` directory. `$Var` is expanded to `Directory` in the boot script.

-compile mod1 mod2 .... Makes the Erlang system compile `mod1.erl mod2.erl ....` and then terminate (with non-zero exit code if the compilation of some file didn't succeed). Implies `-noinput`.

-config Config Reads the `Config`.config configuration file in order to configure the system. See application(3).

-cookie This flag from release 4.3 is now obsolete. The default of an Erlang system is to read the `$HOME/.erlang.cookie` file when started and to set the node cookie to the contents of the user's cookie file. If no such file is found, the system will create one and it will be filled with a random string. See `-setcookie`.

-env Variable Value Sets the HOST OS environment variable `Variable` to the value `Value` of the Erlang system. For example:

```
% erl -env DISPLAY gin:0
```

In this example, an Erlang system is started with the DISPLAY environment variable set to the value `gin:0`.

-heart Starts heart beat monitoring of the Erlang system. See heart(3).

-hosts Hosts Specifies the IP addresses for the hosts on which an Erlang boot servers are running. This flag is mandatory if the `-loader inet` flag is present. On each host, there must be one Erlang node running, on which the `boot_server` must be started.

The IP addresses must be given in the standard form (four decimal numbers separated by periods, for example "150.236.20.74"). Hosts names are not acceptable, but an broadcast address (preferably limited to the local network) is.

-id Id Specifies the identity of the Erlang system. If the system runs as a distributed node, `Id` must be identical to the name supplied together with the `-sname` or `-name` distribution flags.

-instr Selects an instrumented Erlang system (virtual machine) to run, instead of the ordinary one. When running an instrumented system, some resource usage data can be obtained and analysed using the module `instrument`. Functionally, it behaves exactly like an ordinary Erlang system.

-loader Loader Specifies the name of the loader used to load Erlang modules into the system. See erl_prim_loader(3). `Loader` can be `efile` (use the local file system), or `inet` (load using the `boot_server` on another Erlang node). If `Loader` is something else, the user supplied `Loader` port program is started.

If the `-loader` flag is omitted `efile` is assumed.

-make Makes the Erlang system invoke `make:all()` in the current work directory and then terminate. See make(3). Implies `-noinput`.

-man Module Displays the manual page for the Erlang module `Module`.

-mode Mode The mode flag indicates if the system will load code automatically at runtime, or if all code is loaded during system initialization. `Mode` can be either `interactive` to allow automatic code loading, or `embedded` to load all code during start-up. See code(3).

-name Name Makes the node a distributed node. This flag invokes all network servers necessary for a node to become distributed. See net_kernel(3).

The name of the node will be `Name@Host`, where `Host` is the fully qualified host name of the current host. This option also ensures that `epmd` runs on the current host before Erlang is started. See epmd(1).

-noinput Ensures that the Erlang system never tries to read any input. Implies `-noshell`.

-noshell Starts an Erlang system with no shell at all. This flag makes it possible to have the Erlang system as a component in a series of UNIX pipes.

-nostick Disables the sticky directory facility of the `code server`. See code(3).

-oldshell Invokes the old Erlang shell from Erlang release 3.3. The old shell can still be used.

-pa Directories Adds the directories `Directories` to the head of the search path of the code server, as if `code:add_pathsa/1` was called. See code(3).

-pz Directories Adds the directories `Directories` to the end of the search path of the code server, as if `code:add_pathsa/1` was called. See code(3).

-s Mod [Fun [Args]] Passes the `-s` flag to the `init:boot()` routine. See init(3).

-setcookie Cookie Sets the magic cookie of the current node to `Cookie`. As `erlang:set_cookie(node(),Cookie)` is used, all other nodes will also be assumed to have their cookies set to `Cookie`. In this way, several nodes can share one magic cookie. Erlang magic cookies are explained in auth(3).

-sname Name This is the same as the `-name` flag, with the exception that the host name portion of the node name will not be fully qualified. The following command is used do start Erlang at the host with the name `gin.eua.ericsson.se`

```
% erl -sname klacke
Eshell V4.7 (abort with ^G)
(klacke@gin)1>
```

Only the host name portion of the node name will be relevant. This is sometimes the only way to run distributed Erlang if the DNS (Domain Name System) is not running. There can be no communication between systems running with the `-sname` flag and those running with the `-name` flag, as node names must be unique in distributed Erlang systems.

-version Makes the system print out its version number.

-x Invokes the Erlang Support System `xerl`.

All these flags are processed during the start-up of the Erlang kernel servers and before any user processes are started. All flags are passed to `init:boot(Args)`. See init(3). All additional flags (<`user_flags`>) passed to the script will be passed to `init:boot` as well, and they can be accessed using the `init` module.

System Flags

The `erl` script invokes the code for the Erlang virtual machine. This program supports the following flags:

-B De-activates the break handler for `^C` and `^\`.

-h size Sets the default heap size of processes to the size `size`.

-l Displays info while loading code.

-P Number Sets the total number of processes for this system. The `Number` must be greater than 15 and less than 32769.

-s size Sets the default stack size for Erlang processes to the size `size`.

-v Verbose

-V Prints the version of Erlang at start-up.

The `erl` script must be changed to set or change these flags. They can also be set directly with the special flag +. For example:

```
% erl -name foo +B +l
```

In this example, a distributed node is started with the break handler turned off and a lot of info is displayed while the code is loading.

Notes

The special flag `--` is used to end the system flags. The following example is typical for starting a system:

```
jam47 -- -s mod1 -s mod2 ........
```

> Note:
> It is not recommendable to run the code for the Erlang virtual machine directly, or set any of the flags for it with the special + flag. It should be used with care.

See Also

init(3), erl_prim_loader(3), erl_boot_server(3), code(3), application(3), heart(3), net_kernel(3), auth(3), make(3), epmd(1)

erlc (Command)

The `erlc` program provides a common way to run all compilers in the Erlang system. Depending on the extension of each input file, `erlc` will invoke the appropriate compiler. Regardless of which compiler is used, the same flags are used to provide parameters such as include paths and output directory.

Exports

```
erlc flags file1.ext file2.ext...
```

`Erlc` compiles one or more files. The files must include the extension, for example `.erl` for Erlang source code, or `.yrl` for Yecc source code. `Erlc` uses the extension to invoke the correct compiler.

Generally Useful Flags

The following flags are supported:

-I ***directory*** Instructs the compiler to search for include file in the specified directory. If not given, the compiler assumes that include files are located in the current working directory.

-o ***directory*** The directory where the compiler should place the output files. If not specified, output files will be placed in the current working directory.

-D***name*** Defines a macro.

-D***name***=***value*** Defines a macro with the given value. The value can be any Erlang term. Depending on the platform, the value may need to be quoted if the shell itself interprets certain characters. On Unix, terms which contain tuples and list must be quoted. Terms which contain spaces must be quoted on all platforms.

-W Enables warning messages. Without this switch, only errors will be reported.

-v Enables verbose output.

-b ***output-type*** Specifies the type of output file. Generally, *output-type* is the same as the file extension of the output file but without the period. This option will be ignored by compilers that have a a single output format.

– Signals that no more options will follow. The rest of the arguments will be treated as file names, even if they start with hyphens.

+*term* A flag starting with a plus ('+') rather than a hyphen will be converted to an Erlang term and passed unchanged to the compiler. For instance, the `export_all` option for the Erlang compiler can be specified as follows:

```
erlc +export_all file.erl
```

Depending on the platform, the value may need to be quoted if the shell itself interprets certain characters. On Unix, terms which contain tuples and list must be quoted. Terms which contain spaces must be quoted on all platforms.

Special Flags

The flags in this section are useful in special situations such as re-building the OTP system.

-ilroot ***directory*** Defines the root directory to be used for `include_lib` directives in the Erlang compiler. Defaults to the library directory of the emulator where the compiler is run.

-pa ***directory*** Appends *directory* to the front of the code path in the invoked Erlang emulator. This can be used to invoke another compiler than the default one.

-pz ***directory*** Appends *directory* to the code path in the invoked Erlang emulator.

Supported Compilers

.erl Erlang source code. It generates a `.jam` or `.beam` file, depending on which emulator the compiler is run.
If '*-b jam*' or '*-b beam*' is given, the corresponding format is generated.
The options -P, -E, and -S are equivalent to +'P', +'E', and +'S', except that it is not necessary to include the single quotes to protect them from the shell.
Supported options: -ilroot, -I, -o, -D, -v, -W, -b.

.yrl Yecc source code. It generates an `.erl` file.
Use the -I option with the name of a file to use that file as a customized prologue file (the fourth argument of the `yecc:yecc/4` function).
Supported options: -o, -v, -I (see above).

.mib MIB for SNMP. It generates a `.bin` file.
Supported options: -I, -o, -W.

.bin A compiled MIB for SNMP. It generates a `.hrl` file.
Supported options: -o, -v.

.rel Script file. It generates a boot file.
Use the -I to name directories to be searched for application files (equivalent to the `path` in the option list for `systools:make_script/2`).
Supported options: -o.

.h A interface definition for IG (Interface Generator). It generates C and Erlang files.
Supported options: -o.

Environment Variables

ERLC_EMULATOR The command for starting the emulator. Default is *erl* in the same directory as the *erlc* program itself, or if it doesn't exist, *erl* in any of the directories given in the *PATH* environment variable.

See Also

erl(1), erl_compile(3), compile(3), yecc(3), snmp(3)

erlsrv (Command)

This utility is specific to Windows NT(R) It allows Erlang emulators to run as services on the NT system, allowing embedded systems to start without any user needing to log in. The emulator started in this way can be manipulated through the Windows NT(R) services applet in a manner similar to other services.

As well as being the actual service, erlsrv also provides a command line interface for registering, changing, starting and stopping services.

To manipulate services, the logged in user should have Administrator privileges on the machine. The Erlang machine itself is (default) run as the local administrator. This can be changed with the Services applet in Windows NT(R).

The processes created by the service can, as opposed to normal services, be "killed" with the task manager. Killing a emulator that is started by a service will trigger the "OnFail" action specified for that service, which may be a reboot.

The following parameters may be specified for each Erlang service:

- `StopAction`: This tells `erlsrv` how to stop the Erlang emulator. Default is to kill it (Win32 TerminateProcess), but this action can specify any Erlang shell command that will be executed in the emulator to make it stop. The emulator is expected to stop within 30 seconds after the command is issued in the shell. If the emulator is not stopped, it will report a running state to the service manager.
- `OnFail`: This can be either of `reboot`, `restart`, `restart_always` or `ignore` (the default). In case of `reboot`, the NT system is rebooted whenever the emulator stops (a more simple form of watchdog), this could be useful for less critical systems, otherwise use the heart functionality to accomplish this. The restart value makes the Erlang emulator be restarted (with whatever parameters are registered for the service at the occasion) when it stops. If the emulator stops again within 10 seconds, it is not restarted to avoid an infinite loop which could completely hang the NT system. `restart_always` is similar to restart, but does not try to detect cyclic restarts, it is expected that some other mechanism is present to avoid the problem. The default (ignore) just reports the service as stopped to the service manager whenever it fails, it has to be manually restarted.

  On a system where release handling is used, this should always be set to `ignore`. Use `heart` to restart the service on failure instead.
- `Machine`: The location of the Erlang emulator. The default is the `erl.exe` located in the same directory as erlsrv.exe. Do not specify `werl.exe` as this emulator, it will not work.

  If the system uses release handling, this should be set to a program similar to `start_erl.exe`.
- `Env`: Specifies an *additional* environment for the emulator. The environment variables specified here are added to the system wide environment block that is normally present when a service starts up. Variables present in both the system wide environment and in the service environment specification will be set to the value specified in the service.

- `WorkDir`: The working directory for the Erlang emulator, has to be on a local drive (there are no network drives mounted when a service starts). Default working directory for services is `%SystemDrive%%SystemPath%`. Debug log files will be placed in this directory.
- `Priority`: The process priority of the emulator, this can be one of `realtime`, `high`, `low` or `default` (the default). Real-time priority is not recommended, the machine will possibly be inaccessible to interactive users. High priority could be used if two Erlang nodes should reside on one dedicated system and one should have precedence over the other. Low process priority may be used if interactive performance should not be affected by the emulator process.
- `SName or Name`: Specifies the short or long node-name of the Erlang emulator. The Erlang services are always distributed, default is to use the service name as (short) node-name.
- `DebugType`: Can be one of `none` (default), `new`, `reuse` or `console`. Specifies that output from the Erlang shell should be sent to a "debug log". The log file is named <servicename>`.debug` or <servicename>`.debug.`<N>, where <N> is an integer between 1 and 99. The logfile is placed in the working directory of the service (as specified in WorkDir). The `reuse` option always reuses the same log file (<servicename>`.debug`) and the `new` option uses a separate log file for every invocation of the service (<servicename>`.debug.`<N>). The `console` option opens an interactive Windows NT(R) console window for the Erlang shell of the service. The `console` option automatically disables the `StopAction` and a service started with an interactive console window will not survive logouts. If no `DebugType` is specified (`none`), the output of the Erlang shell is discarded.
- `Args`: Additional arguments passed to the emulator startup program `erl.exe` (or `start_erl.exe`). Arguments that cannot be specified here are `-noinput` (StopActions would not work), `-name` and `-sname` (they are specified in any way. The most common use is for specifying cookies and flags to be passed to init:boot() (`-s`).

The naming of the service in a system that uses release handling has to follow the convention *NodeName_Release*, where *NodeName* is the first part of the Erlang nodename (up to, but not including the "@") and *Release* is the current release of the application.

Exports

`erlsrv {set | add} <service-name> [<service options>]`

The set and add commands adds or modifies a Erlang service respectively. The simplest form of an add command would be completely without options in which case all default values (described above) apply. The service name is mandatory.

Every option can be given without parameters, in which case the default value is applied. Values to the options are supplied *only* when the default should not be used (i.e. `erlsrv set myservice -prio -arg` sets the default priority and removes all arguments).

The following service options are currently available:

-st[opaction [<erlang shell command>]] Defines the StopAction, the command given to the erlang shell when the service is stopped. Default is none.

-on[fail [{reboot | restart | restart_always}]] Specifies the action to take when the erlang emulator stops unexpectedly. Default is to ignore.

-m[achine [<erl-command>]] The complete path to the erlang emulator, never use the werl program for this. Default is the `erl.exe` in the same directory as `erlsrv.exe`. When release handling is used, this should be set to a program similar to `start_erl.exe`.

-e[nv [<variable>[=<value>]] ...] Edits the environment block for the service. Every environment variable specified will add to the system environment block. If a variable specified here has the same name as a system wide environment variable, the specified value overrides the system wide. Environment variables are added to this list by specifying <variable>=<value> and deleted from the list by specifying <variable> alone. The environment block is automatically sorted. Any number of `-env` options can be specified in one command. Default is to use the system environment block unmodified (except for two additions, see below [page 19]).

-w[orkdir [<directory>]] The initial working directory of the erlang emulator. Default is the system directory.

-p[riority [{low|high|realtime}]] The priority of the erlang emulator. The default is the Windows NT(R) default priority.

{-sn[ame | -n[ame]} [<node-name>]] The node-name of the erlang machine, distribution is mandatory. Default is `-sname <service name>`.

-d[ebugtype [{new|reuse|console}]] Specifies where shell output should be sent, default is that shell output is discarded.

-ar[gs [<limited erl arguments>]] Additional arguments to the erlang emulator, avoid `-noinput`, `-noshell` and `-sname`/`-name`. Default is no additional arguments. Remember that the services cookie file is not necessarily the same as the interactive users. The service runs as the local administrator. All arguments should be given together in one string, use double quotes (“) to give an argument string containing spaces and use quoted quotes (\”) to give an quote within the argument string if necessary.

`erlsrv {start | stop | disable | enable} <service-name>`

These commands are only added for convenience, the normal way to manipulate the state of a service is through the control panels services applet. The `start` and `stop` commands communicates with the service manager for stopping and starting a service. The commands wait until the service is actually stopped or started. When disabling a service, it is not stopped, the disabled state will not take effect until the service actually is stopped. Enabling a service sets it in automatic mode, that is started at boot. This command cannot set the service to manual.

`erlsrv remove <service-name>`

This command removes the service completely with all its registered options. It will be stopped before it is removed.

`erlsrv list [<service-name>]`

If no service name is supplied, a brief listing of all erlang services is presented. If a service-name is supplied, all options for that service are presented.

`erlsrv help`

ENVIRONMENT

The environment of an erlang machine started as a service will contain two special variables, ERLSRV_SERVICE_NAME, which is the name of the service that started the machine and ERLSRV_EXECUTABLE which is the full path to the `erlsrv.exe` that can be used to manipulate the service. This will come in handy when defining a heart command for your service. A command file for restarting a service will simply look like this:

```
@echo off
%ERLSRV_EXECUTABLE% stop %ERLSRV_SERVICE_NAME%
%ERLSRV_EXECUTABLE% start %ERLSRV_SERVICE_NAME%
```

This command file is then set as heart command.

The environment variables can also be used to detect that we are running as a service and make port programs react correctly to the control events generated on logout (see below).

PORT PROGRAMS

When a program runs in the service context, it has to handle the control events that is sent to every program in the system when the interactive user logs off. This is done in different ways for programs running in the console subsystem and programs running as window applications. An application which runs in the console subsystem (normal for port programs) uses the win32 function `SetConsoleCtrlHandler` to a control handler that returns TRUE in answer to the CTRL_LOGOFF_EVENT. Other applications just forward WM_ENDSESSION and WM_QUERYENDSESSION to the default window procedure. Here is a brief example in C of how to set the console control handler:

```
#include <windows.h>
/*
** A Console control handler that ignores the log off events,
** and lets the default handler take care of other events.
*/
BOOL WINAPI service_aware_handler(DWORD ctrl){
    if(ctrl == CTRL_LOGOFF_EVENT)
        return TRUE;
    return FALSE;
}

void initialize_handler(void){
    char buffer[2];
```

```
    /*
     * We assume we're running as a service if this environment variable
     * is defined
     */
    if(GetEnvironmentVariable("ERLSRV_SERVICE_NAME",buffer,(DWORD) 2)){
        /*
        ** Actually set the control handler
        */
        SetConsoleCtrlHandler(&service_aware_handler, TRUE);
    }
}
```

NOTES

Even though the options are described in a Unix-like format, the case of the options or commands is not relevant, and the "/" character for options can be used as well as the "-" character.

Note that the program resides in the emulators `bin`-directory, not in the `bin`-directory directly under the erlang root. The reasons for this are the subtle problem of upgrading the emulator on a running system, where a new version of the runtime system should not need to overwrite existing (and probably used) executables.

To easily manipulate the erlang services, put the `<erlang_root>\erts-<version>\bin` directory in the path instead of `<erlang_root>\bin`. The erlsrv program can be found from inside erlang by using the `os:find_executable/1` erlang function.

For release handling to work, use `start_erl` as the Erlang machine. It is also worth mentioning again that the name of the service is significant (see above [page 17]).

SEE ALSO

start_erl(1), release_handler(3)

start_erl (Command)

This describes the start_erl program specific to Windows NT. Although there exists programs with the same name on other platforms, their functionality is not the same.

The start_erl program is distributed both in compiled form (under <Erlang root>\erts-<version>\bin) and in source form (under <Erlang root>\erts-<version>\src). The purpose of the source code is to make it possible to easily customize the program for local needs, such as cyclic restart detection etc. There is also a "make"-file, written for the nmake program distributed with Microsoft(R) Visual C++(R). The program can however be compiled with any Win32 C compiler (possibly with slight modifications).

The purpose of the program is to aid release handling on Windows NT(R). The program should be called by the erlsrv program, read up the release data file start_erl.data and start Erlang. Certain options to start_erl are added and removed by the release handler during upgrade with emulator restart (more specifically the -data option).

Exports

start_erl [<erl options>] ++ [<start_erl options>]

The start_erl program in it's original form recognizes the following options:

++ Mandatory, delimits start_erl options from normal Erlang options. Everything on the command line *before* the ++ is interpreted as options to be sent to the erl program. Everything *after* ++ is interpreted as options to start_erl itself.

-reldir <release root> Mandatory if the environment variable RELDIR is not specified. Tells start_erl where the root of the release tree is placed in the file-system (like <Erlang root>\releases). The start_erl.data file is expected to be placed in this directory (if not otherwise specified).

-data <data file name> Optional, specifies another data file than start_erl.data in the <release root>. It is specified relative to the <release root> or absolute (includeing drive letter etc.). This option is used by the release handler during upgrade and should not be used during normal operation. The release data file should not normally be named differently.

-bootflags <boot flags file name> Optional, specifies a file name relative to actual release directory (that is the subdirectory of <release root> where the .boot file etc. are placed). The contents of this file is appended to the command line when Erlang is started. This makes it easy to start the emulator with different options for different releases.

NOTES

As the source code is distributed, it can easily be modified to accept other options. The program must still accept the `-data` option with the semantics described above for the release handler to work correctly.

The Erlang emulator is found by examining the registry keys for the emulator version specified in the release data file. The new emulator needs to be properly installed before the upgrade for this to work.

Although the program is located together with files specific to emulator version, it is not expected to be specific to the emulator version. The release handler does *not* change the `-machine` option to `erlsrv` during emulator restart. Place the (possibly customized) `start_erl` program so that it is not overwritten during upgrade.

The `erlsrv` program's default options are not sufficient for release handling. The machine `erlsrv` starts should be specified as the `start_erl` program and the arguments should contain the `++` followed by desired options.

SEE ALSO

erlsrv(1), release_handler(3)

werl (Command)

On Windows 95/NT, the preferred way to start the Erlang system is:

`werl <script-flags> <user-flags>`

This will start Erlang in its own window, which is nice for interactive use (command-line editing will work and there are scrollbars). All flags except the `-oldshell` flag work as in `erl`.

In cases there you want to redirect standard input and/or standard output or use Erlang in a pipeline, the `werl` is not suitable, and the `erl` program should be used instead.

erl_set_memory_block (C Module)

This documentation is specific to VxWorks.

The `erl_set_memory_block` function/command initiates custom memory allocation for the Erlang emulator. It has to be called before the Erlang emulator is started and makes Erlang use one single large memory block for all memory allocation.

The memory within the block can be utilized by other tasks than Erlang. This is accomplished by calling the functions `sys_alloc`, `sys_realloc` and `sys_free` instead of `malloc`, `realloc` and `free` respectively.

The purpose of this is to avoid problems inherent in the VxWorks systems `malloc` library. The memory allocation within the large memory block avoids fragmentation by using an “address order first fit” algorithm. Another advantage of using a separate memory block is that resource reclamation can be made more easily when Erlang is stopped.

The `erl_set_memory_block` function is callable from any C program as an ordinary 10 argument function as well as from the commandline.

Exports

```
int  erl_set_memory_block(size_t size, void *ptr, int warn_mixed_malloc, int
              realloc_always_moves, int use_reclaim, ...)
```

The function is called before Erlang is started to specify a large memory block where Erlang can maintain memory internally.

Parameters:

size_t size The size in bytes of Erlang’s internal memory block. Has to be specified. Note that the VxWorks system uses dynamic memory allocation heavily, so leave some memory to the system.

void *ptr A pointer to the actual memory block of size `size`. If this is specified as 0 (NULL), Erlang will allocate the memory when starting and will reclaim the memory block (as a whole) when stopped.

If a memory block is allocated and provided here, the `sys_alloc` etc routines can still be used after the Erlang emulator is stopped. The Erlang emulator can also be restarted while other tasks using the memory block are running without destroying the memory. If Erlang is to be restarted, also set the `use_reclaim` flag.

If 0 is specified here, the Erlang system should not be stopped while some other task uses the memory block (has called `sys_alloc`).

int warn_mixed_malloc If this flag is set to true (anything else than 0), the system will write a warning message on the console if a program is mixing normal `malloc` with `sys_realloc` or `sys_free`.

int realloc_always_moves If this flag is set to true (anything else than 0), all calls to `sys_realloc` result in a moved memory block. This can in certain conditions give less fragmentation. This flag may be removed in future releases.

int use_reclaim If this flag is set to true (anything else than 0), all memory allocated with `sys_alloc` is automatically reclaimed as soon as a task exits. This is very useful to make writing port programs (and other programs as well) easier. Combine this with using the routines `save_open` etc. specified in the reclaim.h file delivered in the Erlang distribution.

Return Value:

Returns 0 (OK) on success, otherwise a value <> 0.

`int erl_memory_show(...)`

Return Value:

Returns 0 (OK) on success, otherwise a value <> 0.

`int erl_mem_info_get(MEM_PART_STATS *stats)`

Parameter:

MEM_PART_STATS *stats A pointer to a MEM_PART_STATS structure as defined in <memLib.h>. A successful call will fill in all fields of the structure, on error all fields are left untouched.

Return Value:

Returns 0 (OK) on success, otherwise a value <> 0

NOTES

The memory block used by Erlang actually does not need to be inside the area known to ordinary `malloc`. It is possible to set the `USER_RESERVED_MEM` preprocessor symbol when compiling the wind kernel and then use user reserved memory for Erlang. Erlang can therefor utilize memory above the 32 Mb limit of VxWorks on the PowerPC architecture.

Example:

In config.h for the wind kernel:

```
#undef LOCAL_MEM_AUTOSIZE
#undef LOCAL_MEM_SIZE
#undef USER_RESERVED_MEM

#define LOCAL_MEM_SIZE        0x05000000
#define USER_RESERVED_MEM     0x03000000
```

In the startup script/code for the VxWorks node:

```
erl_set_memory_block(sysPhysMemTop()-sysMemTop(),sysMemTop(),0,0,1);
```

Setting the `use_reclaim` flag decreases performance of the system, but makes programming much easier. Other similar facilities are present in the Erlang system even without using a separate memory block. The routines called `save_malloc`, `save_realloc` and `save_free` provide the same facilities by using VxWorks own `malloc`. Similar routines exist for files, see the file `reclaim.h` in the distribution.

List of Tables

Index

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