The GNU PIES Manual

version 1.8, 13 August 2022

Sergey Poznyakoff.

Published by the Free Software Foundation, 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA

Short Contents

Table of Contents

1 Introduction

The name `pies` (pronounced '`p-yes`') stands for '`Program Invocation and Execution Supervisor`'. This utility starts and controls execution of external programs. In this document these programs will be referred to as *components*. Each component is a stand-alone program, which is executed in the foreground.

Upon startup, `pies` reads the list of components from its configuration file, starts them, and remains in the background, controlling their execution. Each component is defined by the name of the external program to be run and its arguments (command line). The program is normally run directly (via `exec`), but you can instruct `pies` to run it via `sh -c` as well.

The standard output and standard error streams of a component can be redirected to a file or to an arbitrary `syslog` channel.

The way of handling each component, and in particular the action to be taken upon its termination is determined by the component's *mode*.

A *respawn* component is restarted each time it terminates. If it terminates too often, `pies` puts it to sleep for certain time and logs that fact. This prevents badly configured components from taking too much resources and allows administrator to take measures in order to fix the situation. More specific action can be configured, depending on the exit code of the component.

An *inetd*-style components is not started. Instead, `pies` opens a socket associated with it and listens for connections on that socket. When a connection arrives, `pies` runs this component to handle it. The connection is bound to the component's '`stdin`' and '`stdout`' streams. The '`stderr`' stream can be redirected to a file or to syslog, as described above. This mode of operation is similar to that of the `inetd` utility.

Yet another type of components supported by `pies` are *pass-style* or *meta1-style* components. As the name suggests, this type is designed expressly as a support for MeTA1[1] components, namely `smtps`. This type can be regarded as a mixture of the above two. For each meta1-style component `pies` opens a socket and starts the component executable program. Once the program is running, `pies` passes it the file descriptor of that socket, through another preconfigured UNIX-style socket. Further handling of the socket is the responsibility of the program itself.

An *accept* component is basically handled as '`inetd`', except that after binding to the socket `pies` immediately starts the program, without waiting for actual connections.

Finally, two special component modes are available. *Startup* components are run right after `pies` startup, prior to running any other components. Their counterpart, *shutdown* components are run before program termination, after all other components have finished.

Any number of components of all types can be handled simultaneously.

Components are started in the order of their appearance in the configuration file and terminated in reverse order. This order can be modified by declaring component *prerequisites* or *dependents*. This is described in the following chapter.

[1] See `http://www.meta1.org`

As an exception, this order is reversed for the components read from MeTA1 configuration files, either included by `include-meta1` statement (see Section 3.9 [include-meta1], page 37) or expressly supplied in the command line (see [config syntax], page 5).

2 Inter-Component Dependencies

A component '`A`' may depend on another components, say '`B`' and '`C`', i.e. require them to be running at the moment of its startup. Components '`B`' and '`C`' are called *prerequisites* for '`A`', while '`A`' is called a *dependency* or *dependent* component of '`B`', '`C`'.

Before restarting any component, `pies` verifies if it is a prerequisite for any other components. If so, it first terminates its dependencies, restarts the component, and then starts its dependencies again, in the order of their appearance in the configuration file.

3 Pies Configuration File

`Pies` reads its settings and component definitions from one or more *configuration files*. The default configuration file is named `pies.conf` and is located in the *system configuration directory* (in most cases `/etc` or `/usr/local/etc`, depending on how the package was compiled). This file uses the *native Pies configuration syntax*. Apart from this format, the program also understands configuration files in *inetd* and *meta1* formats.

Alternative configuration files may be specified using `--config-file` (`-c` command line option), e.g.:

```
pies --config-file filename
```

Any number of such options may be given. The files named in `--config-file` options are processed in order of their appearance in the command line. By default, `pies` expects configuration files in its native format. This, however, can be changed by using the `--syntax=format` command line option. This option instructs `pies` that any configuration files given after it have are written in the specified *format*. Valid formats are:

'`pies`' `Pies` native configuration file format.

'`inetd`' `Inetd`-style configuration format.

'`meta1`' MeTA1-style format.

'`inittab`' Format of the `/etc/inittab` file (see Chapter 6 [Init Process], page 51).

The configuration file format set by the `--syntax` option remains in effect for all `--config-file` options that follow it, up to the end of the command line or the next occurrence of the `--syntax` option. This means that you can instruct `pies` to read several configuration files of various formats in a single command line, e.g.:

```
pies --config-file /etc/pies.conf \
     --syntax=inetd --config-file /etc/inetd.conf \
     --syntax=meta1 --config-file /etc/meta1/meta1.conf
```

The rest of this chapter concerns the `pies` native configuration file format. You can receive a concise summary of all configuration directives any time by running `pies --config-help`. The use of inetd configuration files is covered in Section 3.8 [inetd], page 36, and the use of meta1 configuration files is described in Section 3.9 [include-meta1], page 37,

If any errors are encountered in the configuration file, the program reports them on the standard error and exits with status 78.

To test the configuration file without actually starting the server, the `--lint` (`-t`) command line option is provided. It causes `pies` to check its configuration file and exit with status 0 if no errors were detected, and with status 78 otherwise.

Before parsing, configuration file is preprocessed using `m4` (see Section 3.2 [Preprocessor], page 8). To see the preprocessed configuration without actually parsing it, use `-E` command line option.

3.1 Configuration File Syntax

The configuration file consists of statements and comments.

There are three classes of lexical tokens: keywords, values, and separators. Blanks, tabs, newlines and comments, collectively called *white space* are ignored except as they serve to separate tokens. Some white space is required to separate otherwise adjacent keywords and values.

3.1.1 Comments

Comments may appear anywhere where white space may appear in the configuration file. There are two kinds of comments: single-line and multi-line comments. *Single-line* comments start with '#' or '//' and continue to the end of the line:

```
# This is a comment
// This too is a comment
```

The following constructs, appearing at the start of a line are treated specially: '#include', '#include_once', '#line', '# *num*' (where *num* is a decimal number). These are described in detail in Section 3.2 [Preprocessor], page 8.

Multi-line or *C-style* comments start with the two characters '/*' (slash, star) and continue until the first occurrence of '*/' (star, slash).

Multi-line comments cannot be nested.

3.1.2 Statements

A *simple statement* consists of a keyword and value separated by any amount of whitespace. The statement is terminated with a semicolon (';').

Examples of simple statements are:

```
pidfile /var/run/pies.pid;
source-info yes;
debug 10;
```

A *keyword* begins with a letter and may contain letters, decimal digits, underscores ('_') and dashes ('-'). Examples of keywords are: 'group', 'control-file'.

A *value* can be one of the following:

number
: A number is a sequence of decimal digits.

boolean
: A boolean value is one of the following: 'yes', 'true', 't' or '1', meaning *true*, and 'no', 'false', 'nil', '0' meaning *false*.

unquoted string
: An unquoted string may contain letters, digits, and any of the following characters: '_', '-', '.', '/', ':'.

quoted string
: A quoted string is any sequence of characters enclosed in double-quotes ('"'). A backslash appearing within a quoted string introduces an *escape sequence*, which is replaced with a single character according to the following rules:

Sequence	Replaced with
\a	Audible bell character (ASCII 7)
\b	Backspace character (ASCII 8)
\f	Form-feed character (ASCII 12)
\n	Newline character (ASCII 10)
\r	Carriage return character (ASCII 13)
\t	Horizontal tabulation character (ASCII 9)
\v	Vertical tabulation character (ASCII 11)
\\	A single backslash ('\')
\"	A double-quote.

Table 3.1: Backslash escapes

In addition, any occurrence of '\' immediately followed by a newline character (ASCII 10) is removed from the string. This allows to split long strings over several physical lines, e.g.:

```
"a long string may be\
 split over several lines"
```

If the character following a backslash is not one of those specified above, the backslash is ignored and a warning is issued.

Here-document

Here-document is a special construct that allows to introduce strings of text containing embedded newlines.

The `<<`*word* construct instructs the parser to read all the following lines up to the line containing only *word*, with possible trailing blanks. Any lines thus read are concatenated together into a single string. For example:

```
<<EOT
A multiline
string
EOT
```

Body of a here-document is interpreted the same way as double-quoted string, unless *word* is preceded by a backslash (e.g. '`<<\EOT`') or enclosed in double-quotes, in which case the text is read as is, without interpretation of escape sequences.

If *word* is prefixed with `-` (a dash), then all leading tab characters are stripped from input lines and the line containing *word*. Furthermore, if `-` is followed by a single space, all leading whitespace is stripped from them. This allows to indent here-documents in a natural fashion. For example:

```
<<- TEXT
    All leading whitespace will be
    ignored when reading these lines.
TEXT
```

It is important that the terminating delimiter be the only token on its line. The only exception to this rule is allowed if a here-document appears as the last element of a statement. In this case a semicolon can be placed on the same line with its terminating delimiter, as in:

```
help-text <<-EOT
        A sample help text.
EOT;
```

list
: A *list* is a comma-separated list of values. Lists are delimited by parentheses. The following example shows a statement whose value is a list of strings:

```
dependents (pmult, auth);
```

In any case where a list is appropriate, a single value is allowed without being a member of a list: it is equivalent to a list with a single member. This means that, e.g. '`dependents auth;`' is equivalent to '`dependents (auth);`'.

A *block statement* introduces a logical group of another statements. It consists of a keyword, followed by an optional value, and a sequence of statements enclosed in curly braces, as shown in the example below:

```
component multiplexor {
        command "pmult";
}
```

The closing curly brace may be followed by a semicolon, although this is not required.

3.2 Preprocessor

Before parsing, configuration file is preprocessed. This goes in three stages. First, include directives are expanded. An *include directive* begins with a '`#`' sign at the beginning of a line, followed by the word '`include`' or '`include_once`'. Any amount of whitespace is allowed between the '`#`' and the word. The entire text up to the end of the line is removed and replaced using the following rules:

`#include` *`file`*
: The contents of the file *file* is included.

If *file* contains wildcard characters ('`*`', '`[`', '`]`' or '`?`'), it is interpreted as shell globbing pattern and all files matching that pattern are included, in lexicographical order. If no matching files are found, the directive is replaced with an empty line.

Otherwise, the named file is included. Unless *file* is an absolute file name, it will be searched relative to the current working directory. An error message will be issued if it does not exist.

`#include_once` *`file`*
: Same as `#include`, except that, if the *file* has already been included, it will not be included again.

The obtained material is then passed to *external preprocessor*. By default, `pies` uses GNU `m4`. This powerful macro processor is described in *GNU M4 macro processor*. For the rest of this subsection we assume the reader is sufficiently acquainted with the `m4` macro processor.

The external preprocessor is invoked with the following two flags: `-s` flag, instructing it to include line synchronization information in its output, and `-P`, which changes all `m4` built-in macro names by prefixing them with '`m4_`'.

The following command line options are passed to `m4` verbatim:

`--define=`*sym*`[=`*value*`]`
`-D` *symbol*`[=`*value*`]`
: Define symbol *sym* as having *value*, or empty, if the *value* is not given.

`--undefine=`*sym*
`-U` *sym*
: Undefine symbol *sym*.

The `--include-directory=`*dir* or `-I` *dir* option causes the option `-I` *dir* to be appended to the preprocessor command line. This option modifies the `m4` include search path (see Section "Search Path" in *GNU M4 macro processor*).

Finally, the following two options are appended:

`-I` *$prefix*`/share/pies/include`
`-I` *$prefix*`/share/pies/1.8/include`

(where *$prefix* stands for installation prefix chosen when the package was built. Normally it is `/usr`.) This step can be disabled using the `--no-include` option.

These provide the default search path.

The name of the source file is appended to the command line, and the constructed command is executed via `$SHELL -c` and its output is then passed to the configuration parser. When parsing, the following constructs appearing at the beginning of a line are handled specially:

`#line` *num*
`#line` *num* `"`*file*`"`
: This line causes the parser to believe, for purposes of error diagnostics, that the line number of the next source line is given by *num* and the current input file is named by *file*. If the latter is absent, the remembered file name does not change.

`#` *num* `"`*file*`"`
: This is a special form of `#line` statement, understood for compatibility with the C preprocessor.

`#warning "`*text*`"`
: Emits *text* as a warning.

`#error "`*text*`"`
: Emits *text* as an error message. Further parsing continues, but will end with failure.

`#abend "`*text*`"`
: Emits *text* as an error message and stops further processing immediately.

If `#error` or `#abend` is encountered, the effect is the same as if syntax error has been detected. If it occurs at `pies` startup, the program will terminate abnormally. If it occurs as part of the reload sequence in a running instance of `pies`, the configuration file will be rejected and old configuration will remain in effect.

3.2.1 Using M4

Editor's note:

This node is to be written.

This subsection gives some tips on using the default preprocessor.

3.2.2 Using Custom Preprocessor

The default preprocessor can be changed (or even disabled) at compile time as well as on the runtime. When invoked with the `--help` option `pies` reports, among others, the preprocessor it is configured to use and the default include search path.

To disable preprocessing, use the `--no-preprocessor` command line option.

To change the default preprocessor at runtime, use the `--preprocessor` option. Its argument is the initial preprocessor command line. Depending on the `pies` command line, it can be further modified by appending new options as described in [additional preprocessor options], page 9.

When selecting another preprocessor, please bear in mind that `pies` assumes that the preprocessor program understands the following three options:

`-D` *`name`*`[=`*`value`*`]`
    Define the preprocessor symbol *name*.

`-I` *`dir`* Add the directory *dir* to the preprocessor search path for include files.

`-U` *`name`* Undefine the preprocessor symbol *name*.

`pies` never passes `-D` and `-U` options, except as if these were passed to it in the command line.

However, it normally adds one or more `-I` options to configure the default search path.

If the preprocessor of your choice doesn't support some or any of these options, there are several possible solutions.

- If the preprocessor doesn't support `-D` and `-U` options, don't pass them in the `pies` command line.
- If it does not support the `-I` option, run `pies` with the `--no-include` option or create a wrapper script which will consume all `-I` options without affecting the preprocessor command line.

For an example of using alternative preprocessor, See Section 7.2 [xenv], page 58.

3.3 The `component` Statement

`component` [Config]

The `component` statement defines a new component:

```
component tag {
  ...
}
```

The component is identified by its *tag*, which is given as argument to the `component` keyword. Component declarations with the same tags are merged into a single declaration.

The following are the basic statements which are allowed within the `component` block:

`mode` *`mode`* [Config: component]

Declare the type (style) of the component. Following are the basic values for *mode*:

`exec`
`respawn` Define a '`respawn`' component (see [respawn], page 1). This is the default.

`inetd`
`nostartaccept`
Define an '`inetd-style`' component (see [inetd-style], page 1).

`pass`
`pass-fd` Define a '`meta1-style`' component (see [meta1-style], page 1).

`accept` Define a '`accept-style`' component (see [accept-style], page 1).

`startup` The component is run right after startup. Prior to starting any other components, `pies` will wait for all `startup` components to terminate.

`shutdown` These components are started as a part of program shutdown sequence, after all regular components have terminated. See [shutdown sequence], page 39, for a detailed discussion.

When run as init process (see Chapter 6 [Init Process], page 51), the following *mode*s are also allowed:

`boot` The process will be executed during system boot. The '`runlevel`' settings are ignored.

`bootwait` The process will be executed during system boot. No other components will be started until it has terminated. The '`runlevel`' settings are ignored.

`ctrlaltdel`
The process will be executed when `pies` receives the SIGINT signal. Normally this means that the CTRL-ALT-DEL combination has been pressed on the keyboard.

`kbrequest`
The process will be executed when a signal from the keyboard handler is received that indicates that a special key combination was pressed on the console keyboard.

`once` The process will be executed once when the specified runlevel is entered.

`ondemand` The process will be executed when the specified *ondemand* runlevel is called ('`a`', '`b`' and '`c`'). No real runlevel change will occur (see [Ondemand runlevels], page 52). The process will remain running across any eventual runlevel changes and will be restarted whenever it terminates, similarly to `respawn` components.

`powerfail`
: The process will be executed when the power goes down. `Pies` will not wait for the process to finish.

`powerfailnow`
: The process will be executed when the power is failing and the battery of the external UPS is almost empty.

`powerokwait`
: The process will be executed as soon as `pies` is informed that the power has been restored.

`powerwait`
: The process will be executed when the power goes down. `Pies` will wait for the process to finish before continuing.

`sysinit` The process will be executed during system boot, before any `boot` or `bootwait` entries. The ‘`runlevel`’ settings are ignored.

`wait` The process will be started once when the specified runlevel is entered. `Pies` will wait for its termination before starting any other processes.

`command` ***`string`*** [Config: component]

Command line to run. *string* is the full command line. Its first word (in the shell sense) is the name of the program to invoke.

`program` ***`name`*** [Config: component]

Full file name of the program to run. When supplied, `pies` will execute the program *name* instead of the first word in the `command` statement. The latter, however, will be passed to the running program as `argv[0]`.

`flags` **(*`flag-list`*)** [Config: component]

Define flags for this component. The *flag-list* is a comma-separated list of flags. Valid flags are:

`disable` This component is disabled, i.e. `pies` will parse and remember its settings, but will not start it.

`nullinput`
: Do not close standard input. Redirect it from `/dev/null` instead. Use this option with commands that require their standard input to be open (e.g. `pppd nodetach`).

`precious` Mark this component as *precious*. Precious components are never disabled by `pies`, even if they respawn too fast.

`shell` Run command as `/bin/sh -c "$command"`. Use this flag if command contains shell-specific features, such as I/O redirections, pipes, variables or the like. You can change the shell program using the `program` statement. For example, to use Korn shell:

```
component X {
  flags shell;
  program "/bin/ksh";
```

```
        command "myprog $HOME";
      }
```

`expandenv`
: Expand environment variables in the '`command`' statement prior to running it. When used together with the '`shell`' flag, this flag produces a warning and has no effect. See Section 3.3.5 [Early Environment Expansion], page 18, for a detailed discussion.

`wait` This flag is valid only for '`inetd`' components. It has the same meaning as '`wait`' in `inetd.conf` file, i.e. it tells `pies` to wait for the server program to return. See Appendix A [inetd configuration], page 73.

`tcpmux` This is a TCPMUX component. See Section 3.3.9.2 [TCPMUX], page 24.

`tcpmuxplus`
: This is a TCPMUX+ component. See Section 3.3.9.2 [TCPMUX], page 24.

`internal` This is an internal inetd component. See Section 3.3.9.1 [builtin], page 23.

`sockenv` This inetd component wants socket description variables in its environment. See Section 3.3.9.3 [sockenv], page 25.

`resolve` When used with '`sockenv`', the `LOCALHOST` and `REMOTEHOST` environment variables will contain resolved host names, instead of IP addresses.

`siggroup` This flag affects the behavior of `pies` when a stopped process fails to terminate within a predefined timeout (see [shutdown-timeout], page 39. Normally `pies` would send the '`SIGKILL`' signal to such a process. If this flag is set, `pies` would send '`SIGKILL`' to the process group of this process instead.

`sigterm` *`sig`* [Config: component]

Defines signal which should be sent to terminate this component. The default is `SIGTERM`. The argument *sig* is either the name of a signal defined in `/usr/include/signal.h`, or '`SIG+`*`n`*', where *n* is signal number.

The following subsections describe the rest of '`component`' substatements.

3.3.1 Component Prerequisites

Prerequisites (see [component prerequisite], page 3) for a component are declared using the following statement:

`prerequisites` *`tag-list`* [Config: component]

The argument is either a list of component tags or one of the following words:

`all` Declare all components defined so far as prerequisites for this one.

`none` No prerequisites. This is the default.

If you wish, you can define dependents, instead of prerequisites:

`dependents` *`tag-list`* [Config: component]

Declare dependents for this component. *var-list* is a list of component tags.

3.3.2 Component Privileges

The following statements control privileges the component is executed with.

`user` *`user-name`* [Config: component]

Start component with the UID and GID of this user.

`group` *`group-list`* [Config: component]

Retain supplementary groups, specified in *group-list*.

`allgroups` *`bool`* [Config: component]

Retain all supplementary groups of which the user (as given with `user` statement) is a member. This is the default for components specified in `meta1.conf` file (see Section 3.9 [include-meta1], page 37).

3.3.3 Resources

`limits` *`string`* [Config: component]

Impose limits on system resources, as defined by the *string* argument. It consists of *commands*, optionally separated by any amount of whitespace. A command is a single command letter followed by a number, that specifies the limit. The command letters are case-insensitive and coincide with those used by the shell `ulimit` utility:

Command	**The limit it sets**
A	max address space (KB)
C	max core file size (KB)
D	max data size (KB)
F	maximum file size (KB)
M	max locked-in-memory address space (KB)
N	max number of open files
R	max resident set size (KB)
S	max stack size (KB)
T	max CPU time (MIN)
U	max number of processes
P	process priority -20..20 (negative = high priority)

Table 3.2: Limit Command Letters

For example:

```
limits T10 R20 U16 P20
```

Additionally, the command letter '`L`' is recognized. It is reserved for future use ('`number of logins`' limit) and is ignored in version 1.8.

`umask` *`number`* [Config: component]

Set the umask. The *number* must be an octal value not greater than '`777`'. The default umask is inherited at startup.

`max-instances` *`n`* [Config: component]

Sets the maximum number of simultaneously running instances of this component.

3.3.4 Environment

Normally all components inherit the environment of the master `pies` process. You can modify this environment using the `env` statement. It has two variants: *compound* and *legacy*. The legacy one-line statement was used in `pies` versions up to 1.3 and is still retained for backward compatibility. It is described in Section 3.3.4.1 [env legacy syntax], page 16. This subsection describes the modern compound syntax.

The `env` statement can also be used in global context, in which case it modifies environment for the master `pies` program, i.e. the environment that will be inherited by all components (see Section 3.10 [Global Configuration], page 38). The global `env` is available only in compound syntax described here.

`env` *{ ... }* [Config: component]

The compound `env` statement has the following syntax:

```
env {
   clear;
   keep pattern;
   set "name=value";
   eval "value";
   unset pattern;
}
```

Statements inside the `env` block define operations that modify the environment. The `clear` and `keep` statements are executed first. Then, the `set` and `unset` statements are applied in the order of their appearance in the configuration.

`clear` [env]

Clears the environment by removing (unsetting) all variables, except those listed in `keep` statements, if such are given (see below). The `clear` statement is always executed first.

`keep` *pattern* [env]

Declares variables matching *pattern* (a globbing pattern) as exempt from clearing. This statement implies `clear`.

For example, the following configuration fragment removes from the environment all variables except ‘`HOME`’, ‘`USER`’, ‘`PATH`’, and variables beginning with ‘`LC_`’:

```
env {
  clear;
  keep HOME;
  keep USER;
  keep PATH;
  keep "LC_*";
}
```

`keep "`*name*`=`*value*`"` [env]

Retains the variable *name*, if it has the given value. Note, that the argument must be quoted.

`set "`*`name=value`*`"` [env]

Assigns *value* to environment variable *name*. The value is subject to *variable expansion* using the same syntax as in shell. The `set` and `unset` (see below) statements are executed in order of their appearance. For example

```
env {
  set "MYLIB=$HOME/lib";
  set "LD_LIBRARY_PATH=$LD_LIBRARY_PATH${LD_LIBRARY_PATH:+:}$MYLIB";
}
```

`eval "`*`value`*`"` [env]

Perform variable expansion on *value* and discard the result. This is useful for side-effects. For example, to provide default value for the `LD_LIBRARY_PATH` variable, one may write:

```
env {
  eval "${LD_LIBRARY_PATH:=/usr/local/lib}";
}
```

`unset` *`pattern`* [env]

Unset environment variables matching *pattern*. The following will unset the `LOGIN` variable:

```
unset LOGIN;
```

The following will unset all variables starting with ‘`LD_`’:

```
unset "LD_*";
```

Notice, that patterns containing wildcard characters must be quoted.

3.3.4.1 `env`: legacy syntax.

Up to version 1.3 `pies` implemented the one-line variant of the `env` statement. The use of this legacy syntax is discouraged. It is supported for backward compatibility only and will be removed in future versions. This subsection describes the legacy syntax.

`env` *`args`* [legacy syntax]

Set program environment.

Arguments are a whitespace-delimited list of specifiers. The following specifiers are understood:

- (a dash) Clear the environment. This is understood only when used as a first word in *args*.

The modern syntax equivalent is:

```
env {
  clear;
}
```

-*name* Unset the environment variable *name*. The modern syntax equivalent is

```
env {
  unset name;
}
```

-*name*=*val*

Unset the environment variable *name* only if its value is *val*. The modern syntax equivalent is:

```
env {
  unset "name=val";
}
```

name

Retain the environment variable *name*. The modern syntax equivalent is

```
env {
  keep name;
}
```

name=*value*

Define environment variable *name* to have given *value*. The modern syntax equivalent is:

```
env {
  keep "name=value";
}
```

name+=*value*

Retain variable *name* and append *value* to its existing value. If no such variable is present in the environment, it is created and *value* is assigned to it. However, if *value* begins with a punctuation character, this character is removed from it before the assignment. This is convenient for using this construct with environment variables like `PATH`, e.g.:

```
PATH+=:/sbin
```

In this example, if `PATH` exists, '`:/sbin`' will be appended to it. Otherwise, it will be created and '`/sbin`' will be assigned to it.

In modern syntax, use shell variable references, e.g.:

```
env {
  set "PATH=${PATH}${PATH:+:}/sbin";
}
```

name=+*value*

Retain variable *name* and prepend *value* to its existing value. If no such variable is present in the environment, it is created and *value* is assigned to it. However, if *value* ends with a punctuation character, this character is removed from it before assignment.

In modern syntax, use shell variable references, e.g. instead of doing

```
env PATH=+/sbin:
```

use

```
env {
  set "PATH=/sbin${PATH:+:}$PATH";
}
```

3.3.5 Early Environment Expansion

By default any references to environment variables encountered in the `command` statement are not expanded. If you need to expand them, there are two *flags* (see [flags], page 12) at your disposal: '`shell`' and '`expandenv`'.

The '`shell`' flag instructs `pies` to pass the command line specified by the the `command` statement as the argument to the '`/bin/sh -c`' command (or another shell, if specified by the '`program`' statement). This naturally causes all references to the environment variables to be expanded, as in shell. The overhead is that two processes are run instead of the one: first the shell and second the command itself, being run as its child. This overhead can be eliminated by using the `exec` statement before the command, to instruct the shell to replace itself with the command without creating a new process.

Use this flag if the command you use in the component definition is a shell built-in, a pipe or another complex shell statement.

Another way to expand environment variables in the command line is by specifying the '`expandenv`' flag. This flag instructs `pies` to expand any variable references the same way that the Bourne shell would expand them, but without actually invoking the shell.

A variable reference has the form '`$variable`' or '`${variable}`', where *variable* is the variable name. The two forms are entirely equivalent. The form with curly braces is normally used if the variable name is immediately followed by an alphanumeric symbol, which will otherwise be considered part of it. This form also allows for specifying the action to take if the variable is undefined or expands to an empty value:

${*variable*:-*word*}
: *Use Default Values.* If *variable* is unset or null, the expansion of *word* is substituted. Otherwise, the value of *variable* is substituted.

${*variable*:=*word*}
: *Assign Default Values.* If *variable* is unset or null, the expansion of *word* is assigned to variable. The value of *variable* is then substituted.

${*variable*:?*word*}
: *Display Error if Null or Unset.* If *variable* is null or unset, the expansion of *word* (or a message to that effect if *word* is not present) is output to the current logging channel. Otherwise, the value of *variable* is substituted.

${*variable*:+*word*}
: *Use Alternate Value.* If *variable* is null or unset, nothing is substituted, otherwise the expansion of *word* is substituted.

When the two flags are used together, the preference is given to '`shell`', and a warning message to that effect is issued.

Also, please note, that whichever option you chose the environment variables available for expansion are those inherited by the parent shell and modified by the `env` statement (see Section 3.3.4 [Environment], page 15).

3.3.6 Actions Before Startup

The statements described in this subsection specify actions to perform immediately before starting the component:

`chdir` *dir* [Config: component]

Change to the directory *dir*.

`remove-file` *file-name* [Config: component]

Remove *file-name*. This is useful, for example, to remove stale UNIX sockets or pidfiles, which may otherwise prevent the component from starting normally.

As of version 1.8 only one `remove-file` may be given.

`pass-fd-timeout` *number* [Config: component]

Wait *number* of seconds for the '`pass-fd`' socket to become available (see Section 3.3.10 [Meta1-Style Components], page 26). Default is 5 seconds.

3.3.7 Exit Actions

The default behavior of `pies` when a '`respawn`' component terminates is to restart it. Unless the component terminates with 0 exit code, a corresponding error message is issued to the log file. This behavior can be modified using `return-code` statement:

`return-code` [Config: component]

```
return-code codes {
  ...
}
```

The *codes* argument is a list of exit codes or signal names. Exit codes can be specified either as decimal numbers or as symbolic code names from the table below:

Name	**Numeric value**
EX_OK	0
EX_USAGE	64
EX_DATAERR	65
EX_NOINPUT	66
EX_NOUSER	67
EX_NOHOST	68
EX_UNAVAILABLE	69
EX_SOFTWARE	70
EX_OSERR	71
EX_OSFILE	72
EX_CANTCREAT	73
EX_IOERR	74
EX_TEMPFAIL	75
EX_PROTOCOL	76
EX_NOPERM	77
EX_CONFIG	78

Table 3.3: Standard Exit Codes

Signal numbers can be given either as '`SIG+`*n*', where *n* is the signal number, or as signal names from the following list: '`SIGHUP`', '`SIGINT`', '`SIGQUIT`', '`SIGILL`', '`SIGTRAP`', '`SIGABRT`', '`SIGIOT`', '`SIGBUS`', '`SIGFPE`', '`SIGKILL`', '`SIGUSR1`', '`SIGSEGV`', '`SIGUSR2`', 'SIGPIPE', 'SIGALRM', 'SIGTERM', 'SIGSTKFLT', 'SIGCHLD', 'SIGCONT', 'SIGSTOP', 'SIGTSTP', 'SIGTTIN', 'SIGTTOU', 'SIGURG', 'SIGXCPU', 'SIGXFSZ', 'SIGVTALRM', 'SIGPROF', 'SIGWINCH', 'SIGPOLL', 'SIGIO', 'SIGPWR', 'SIGSYS'.

If the component exits with an exit code listed in *codes* or is terminated on a signal listed in *codes*, `pies` executes actions specified in that '`return-code`' block. The actions are executed in the order of their appearance below:

`exec` *`command`* [Config: return-code]

Execute the supplied external command. Prior to execution, all file descriptors are closed. The *command* inherits the environment from the main `pies` process with the following additional variables:

`PIES_VERSION`
The `pies` version number (1.8).

`PIES_MASTER_PID`
PID of the master `pies` process.

`PIES_COMPONENT`
Tag of the terminated component (see Section 3.3 [Component Statement], page 10).

`PIES_PID` PID of the terminated component.

`PIES_SIGNAL`
If the component terminated on signal, the number of that signal.

`PIES_STATUS`
Program exit code.

`action` *'`disable | restart`'* [Config: return-code]

If '`restart`' is given, restart the component. This is the default. Otherwise, mark the component as disabled. Component dependents are stopped and marked as disabled as well. Once disabled, the components are never restarted, unless their restart is requested by the administrator.

`notify` *`email-string`* [Config: return-code]

Send an email notification to addresses in *email-string*. See Section 3.4 [Notification], page 30, for a detailed discussion of this feature.

`message` *`string`* [Config: return-code]

Supply notification message text to use by `notify` statement. See Section 3.4 [Notification], page 30, for a detailed discussion of this feature.

Any number of `return-code` statements are allowed, provided that their *codes* do not intersect.

The `return-code` statements can also be used outside of `component` block. In this case, they supply global actions, i.e. actions applicable to all components. Any `return-code` statements appearing within a `component` block override the global ones.

3.3.8 Output Redirectors

Output redirectors allow to redirect the standard error and/or standard output of a component to a file or `syslog` facility.

`stderr` *type channel* [Config: component]
`stdout` *type channel* [Config: component]

Redirect standard error (if `stderr`) or standard output (if `stdout`) to the given channel.

The type of redirection is specified by *type* argument:

`file` Redirect to a file. In this case *channel* gives the full name of the file. For example:

```
stderr file /var/log/component/name.err;
```

`syslog` Redirect to syslog. The *channel* parameter is either the syslog facility and priority separated by dot or the priority alone, in which case the facility will be taken from the `syslog` statement (see [syslog], page 38).

Example:

```
stdout syslog local1.info;
stderr syslog err;
```

Valid facilities are: '`user`', '`daemon`', '`auth`', '`authpriv`', '`mail`', '`cron`', '`local0`' through '`local7`' (all names case-insensitive).

Valid priorities are: '`emerg`', '`alert`', '`crit`', '`err`', '`warning`', '`notice`', '`info`', '`debug`'.

3.3.9 Inetd-Style Components

Inetd-style components are declared using `mode inetd` statement. The '`component`' declaration must contain a '`socket`' statement:

`socket` *url* [Config: component]

Define a socket to listen on. Allowed values for *url* are:

inet[+*proto*]://*ip*:*port*
Listen on IPv4[1] address *ip* (may be given as a symbolic host name), on port *port*. Optional *proto* defines the protocol to use. It must be a valid protocol name as given in `/etc/protocols`. Default is '`tcp`'.

local[+*proto*]://*file*[;*args*]
file[+*proto*]://*file*[;*args*]
unix[+*proto*]://*file*[;*args*]
Listen on the UNIX socket file *file*, which is either an absolute or relative file name, as described above. The *proto* part is as described above. Optional arguments, *args*, control ownership and file mode of *file*. They are a list of assignments, separated by semicolons. The following values are allowed:

`user` User name of the socket owner.

[1] Support for IPv6 will be added in future versions.

`group` Owner group of the socket, if it differs from the `user` group.

`mode` Socket file mode (octal number between '`0`' and '`777`').

`umask` Umask to use when creating the socket (octal number between '`0`' and '`777`').

For example:

```
socket
 "unix:///var/run/socket;user=nobody;group=mail;mode=770";
```

The *file* part may be a relative file name, provided that the `chdir` statement is used for this component (see Section 3.3.6 [Actions Before Startup], page 18).

`socket-type` *`type`* [Config: component]

Sets the socket type. Allowed values for *type* are: '`stream`', '`dgram`', '`raw`', '`rdm`', '`seqpacket`'. Default is '`stream`'. Notice that some socket types may not be implemented by all protocol families, e.g. '`seqpacket`' is not implemented for '`inet`'.

`max-rate` *`n`* [Config: component]

Specifies the maximum number of times the component can be invoked in one minute; the default is unlimited. A rate of '`0`' stands for '`unlimited`'.

`max-instances` *`n`* [Config: component]

Sets the maximum number of simultaneously running instances of this component. It is equivalent to the maximum number of simultaneously opened connections.

`max-instances-message` *`text`* [Config: component]

Text to send back if `max-instances` is reached. This is valid only for TCP sockets.

`max-ip-connections` *`number`* [Config: component]

Maximum number of connections that can be opened simultaneously from a single IP address.

`max-ip-connections-message` *`text`* [Config: component]

Textual message to send in reply to an incoming TCP connection from the IP address that has already reached `max-ip-connections` limit.

`acl` *{ . . . }* [Config: component]

Set access control list for this component. This is valid only for '`inetd`' and '`accept`' components. See Section 3.5 [ACL], page 32, for a detailed description of access control lists.

`access-denied-message` *`text`* [Config: component]

Textual message to send in reply to an incoming TCP connection that has been denied by ACL settings.

3.3.9.1 Built-in Inetd Services

Built-in or *internal* services are such inetd-style components that are supported internally by `pies` and do not require external programs. In `pies` version 1.8 those are:

`echo` Send back any received data. Defined in RFC 862 (`http://tools.ietf.org/html/rfc862`).

`discard` Read the data and discard them. Defined in RFC 863 (`http://tools.ietf.org/html/rfc863`).

`time` Return a machine readable date and time as seconds since the Epoch. Defined in RFC 868 (`http://tools.ietf.org/html/rfc868`).

`daytime` Return current date and time in human-readable format. Defined in RFC 867 (`http://tools.ietf.org/html/rfc867`).

`chargen` Send a continuous stream of ASCII printable characters without regard to the input. Defined in RFC 864 (`http://tools.ietf.org/html/rfc864`)

`qotd` Send a '`quotation of the day`' text without regard to the input. Defined in RFC 865 (`http://tools.ietf.org/html/rfc865`).

`tcpmux` TCP Port Service Multiplexer. Defined in RFC 1078 (`http://tools.ietf.org/html/rfc1078`).

A definition of a built-in service component must have the `internal` flag (see [flags], page 12) set. It may not contain `command` or `program` statements, as built-in services do not need external programs. Instead, a *service* declaration must be present:

`service` ***name*** [Config: component]

Set the built-in service name. Its argument is one of the keywords listed in the above table.

For example, the following component declaration defines a standard TCP-based echo service:

```
component echo {
        socket "inet://0.0.0.0:echo";
        service echo;
        flags internal;
}
```

It corresponds to the following `inetd.conf` line:

```
echo stream  tcp     nowait  root    internal
```

Another built-in services are defined in the same manner, replacing '`echo`' in the `service` field with the corresponding service name.

The '`qotd`' service reads the contents of the *qotd file* and sends it back to the client. By default the '`qotd`' file is located in the local state directory and named ***instance*.qotd** (where *instance* is the name of the `pies` instance; see [instances], page 65). This default location can be changed using the following statement:

`qotd-file` ***file-name*** [Config]

Set the name of the '`quotation-of-the-day`' file.

The text read from the '`qotd`' file is preprocessed, by replacing each LF character (ASCII 10) with two characters: CR (ASCII 13) followed by LF. The resulting text is truncated to 512 characters.

The use of '`tcpmux`' services is covered below.

3.3.9.2 TCPMUX Services

TCPMUX allows to use multiple services on a single well-known TCP port using a service name instead of a well-known number. It is defined in RFC 1078 (`http://tools.ietf.org/html/rfc1078`). The protocol operation is as follows. The *master* TCPMUX component listens on a certain TCP port (usually on port 1) for incoming requests. After connecting to the master, the client sends the name of the service it wants, followed by a carriage-return line-feed (CRLF). `Pies` looks up this name in the list of services handled by the master (*subordinate services*) and reports with '`+`' or '`-`' followed by optional text and terminated with the CRLF, depending on whether such service name is found or not. If the reply was '`+`', `pies` then starts the requested component. Otherwise, it closes the connection.

TCPMUX service names are case-insensitive. The special service '`help`' is always defined; it outputs a list of all the subordinate services, one name per line, and closes the connection.

The master TCPMUX service is declared as a usual built-in service, e.g.:

```
component tcpmux-master {
        socket "inet://0.0.0.0:1";
        service tcpmux;
        flags internal;
}
```

Any number of subordinate services may be defined for each master. A subordinate server component definition must contain at least the following statements:

`service` *`name`* [Config: component]

Sets the name of the subordinate service. The *name* will be compared with the first input line from the client.

`tcpmux-master` *`name`* [Config: component]

Sets the name of the master TCPMUX service.

`flags` *`list`* [Config: component]

The `flags` statement (see [flags], page 12) must contain at least one of the following flags:

`tcpmux` A "dedicated" TCPMUX subordinate service. When invoked, it must output the '`+ CRLF`' response itself.

`tcpmuxplus`

Simple service. Before starting it, `pies` will send the '`+ CRLF`' reply.

`command` *`command-line`* [Config: component]

The command line for handling this service.

For example:

```
component scp-to {
```

```
            service scp-to;
            flags (tcpmuxplus, sockenv);
            tcpmux-master tcpmux;
            command "/usr/sbin/in.wydawca";
    }
```

For TCPMUX services, access control lists are handled in the following order. First, the global ACL is checked. If it rejects the connection, no further checks are done. Then, if the master TCPMUX service has an ACL, that ACL is consulted. If it allows the connection, the subordinate is looked up. If found, its ACL (if any) is consulted. Only if all three ACLs allow the connection, is the service started.

A similar procedure applies for other resources, such as `limits`, `umask`, `env`, `user`, `group`, etc.

3.3.9.3 Socket Environment Variables

If the '`sockenv`' flag is set (see [flags], page 12), the following environment variables are set prior to executing the command:

`PROTO` Protocol name.

`SOCKTYPE` Socket type. See [socket-type], page 22, for a list of possible values.

`LOCALIP` IP address of the server which is handling the connection.

`LOCALPORT`
Local port number.

`LOCALHOST`
Host name of the server. This variable is defined only if the '`resolve`' flag is set (see [flags], page 12).

`REMOTEIP` IP address of the remote party (client).

`REMOTEPORT`
Port number on the remote side.

`REMOTEHOST`
Host name of the client. This variable is defined only if the '`resolve`' flag is set (see [flags], page 12).

The variables whose names begin with `REMOTE` are defined only for TCP connections.

3.3.9.4 Exit Actions in Inetd Components

Exit actions (see Section 3.3.7 [Exit Actions], page 19) work for '`inet-style`' components. The only difference from '`respawn`' components is that the '`restart`' action is essentially ignored, as it makes no sense to start an '`inet-style`' component without a communication socket.

A common use of `return-code` statement is to invoke an external program upon the termination of a component. For example, the following configuration snippet configures an FTP server and ensures that a special program is invoked after closing each FTP connection:

```
    component ftp {
        return-code EX_OK {
            exec "/sbin/sweeper --log";
        }
        mode inetd;
        socket "inet://0.0.0.0:21";
        umask 027;
        program /usr/sbin/in.ftpd
        command "ftpd -ll -C -t180";
    }
```

This approach may be used to process FTP uploads in real time.

3.3.10 Meta1-Style Components

Meta1-style components are declared using `mode pass` statement. For such components, you must declare both a socket to listen on (see [inetd-socket], page 21, and a UNIX socket name to pass the file descriptor to the component. The latter is defined using `pass-fd-socket` statement:

`pass-fd-socket` *file-name* [Config: component]

The argument is an absolute or relative file name of the socket file. In the latter case, the `chdir` *dir* statement must be used for this component (see Section 3.3.6 [Actions Before Startup], page 18), and the socket will be looked under *dir*.

This socket file is supposed to be created by the component binary upon its startup.

3.3.11 Component Visibility ACLs

Pies control interface allows certain users to list and modify components of a running `pies` instance. Two access control lists define who can list and modify the particular component.

`list-acl` *name* [Config: component]
`list-acl` *{ . . . }* [Config: component]

This list controls who can get listing of this component (see [piesctl list], page 44).

In the first form, *name* refers to the name of an already defined global ACL (see [defacl], page 32).

The second form defines new unnamed ACL. The syntax is described in detail in Section 3.5 [ACL], page 32.

`admin-acl` *name* [Config: component]
`admin-acl` *{ . . . }* [Config: component]

This list controls who can stop, restart or otherwise modify this component (see Section 5.4 [components], page 44).

As above, two forms are available: the first one for using an already defined named ACL, and the second one, for defining a new ACL in place.

3.3.12 Component Syntax Summary

This subsection summarizes the `component` statements. For each statement, a reference to its detailed description is provided.

```
component tag {
  # Component execution mode.
  # See Section 3.3 [Component Statement], page 10.
  mode modename;

  # Full name of the program.
  # See Section 3.3 [Component Statement], page 10.
  program name;
  # Command line.
  # See Section 3.3 [Component Statement], page 10.
  command string;

  # List of prerequisites.
  # See Section 3.3.1 [Prerequisites], page 13.
  prerequisites (compnames);
  # List of components for which this one is a prerequisite.
  # See Section 3.3.1 [Prerequisites], page 13.
  dependents (compnames);

  # List of flags.
  # See [flags], page 12.
  flags (flags);

  # For init components: runlevels in which to start this
  # component.
  # See Section 6.1 [Runlevels], page 52.
  runlevels string;

  # Listen on the given url.
  # See Section 3.3.9 [Inetd-Style Components], page 21.
  socket url;

  # Set socket type.
  # See Section 3.3.9 [Inetd-Style Components], page 21.
  socket-type ‘stream | dgram | raw | rdm | seqpacket’;

  # Service name for built-in inetd component.
  # See Section 3.3.9.1 [builtin], page 23.
  service string;

  # Tag of master TCPMUX component, for subordinate components.
  # See Section 3.3.9.2 [TCPMUX], page 24.
  tcpmux-master string;

  # Pass fd through this socket.
  # See Section 3.3.10 [Meta1-Style Components], page 26.
  pass-fd-socket soket-name;
  # Wait number of seconds for pass-fd socket to become available.
  # See Section 3.3.6 [Actions Before Startup], page 18.
```

```
pass-fd-timeout number;

# Maximum number of running instances.
# See Section 3.3.3 [Resources], page 14.
# See Section 3.3.9 [Inetd-Style Components], page 21.
max-instances number;

# For 'inetd' components:
# Text to send back if max-instances is reached.
# See Section 3.3.9 [Inetd-Style Components], page 21.
max-instances-message text;

# Maximum number of times an inetd component can be invoked in
# one minute.
# See Section 3.3.9 [Inetd-Style Components], page 21.
max-rate number;

# For 'inetd' components:
# Max. number of simultaneous connections from a single IP address.
# See Section 3.3.9 [Inetd-Style Components], page 21.
max-ip-connections number;

# For 'inetd' components:
# Text to send back if max-ip-connections is reached.
# See Section 3.3.9 [Inetd-Style Components], page 21.
max-ip-connections-message text;

# For 'inetd' components:
# Text to send back if access is denied by ACL.
# See Section 3.3.9 [Inetd-Style Components], page 21.
access-denied-message text;

# ACL for administrative (read-write) access to this component.
# See Section 3.3.11 [Visibility], page 26.
admin-acl name;
# or:
admin-acl { ... }

# ACL for read-only access to this component.
# See Section 3.3.11 [Visibility], page 26.
list-acl name;
# or:
list-acl { ... }

# ACL for this component.
# See Section 3.5 [ACL], page 32.
acl name;
```

```
# or:
acl { ... }

# Redirect program's standard output to the given
# file or syslog priority.
# See Section 3.3.8 [Output Redirectors], page 21.
stdout 'file | syslog' channel;
# Redirect program's standard error to the given
# file or syslog priority.
# See Section 3.3.8 [Output Redirectors], page 21.
stderr 'file | syslog' channel;

# Run with this user privileges.
# See Section 3.3.2 [Component Privileges], page 14.
user user-name;
# Retain supplementary group.
# See Section 3.3.2 [Component Privileges], page 14.
group group-name;
# Retain all supplementary groups of which user is a member.
# See Section 3.3.2 [Component Privileges], page 14.
allgroups bool;

# Set system limits.
# See Section 3.3.3 [Resources], page 14.
limits string;

# Force this umask.
# See Section 3.3.3 [Resources], page 14.
umask number;

# Set program environment.
# See Section 3.3.4 [Environment], page 15.
env { ... }

# Change to this directory before executing the component.
# See Section 3.3.6 [Actions Before Startup], page 18.
chdir dir;
# Remove file-name before starting the component.
# See Section 3.3.6 [Actions Before Startup], page 18.
remove-file file-name;

# Actions:
# See Section 3.3.7 [Exit Actions], page 19.
return-code exit-code-list {
  # Action to take when a component finishes with this return code.
  action 'disable | restart';
  # Notify these addresses when then component terminates.
```

```
      notify email-string;
      # Notification message text (with headers).
      message string;
      # Execute this command.
      exec command
    }
  }
```

3.4 Notification

Pies provides a *notification* mechanism, which can be used to send email messages when components terminate. The exact contents of such notifications and the list of their recipients may depend on the exit code which the component returned. Notification is configured by '`notify`' and '`message`' statements in a '`return-code`' block.

`notify` *`email-string`* [Config: return-code]

Send email notification to each address from *email-string*. The latter is a comma-separated list of email addresses, e.g.:

```
notify "root@localhost,postmaster@localhost";
```

`message` *`string`* [Config: return-code]

Supply the email message text to be sent. *String* must be a valid RFC 822 message, i.e. it must begin with message headers, followed by an empty line and the actual message body.

The message may contain variable data in the form of variable references. A *variable* is an entity that holds some data describing the event that occurred. Meta-variables are referenced using the following construct:

```
${name}
```

where *name* is the name of the variable. Before actually sending the message, each occurrence of this construct is removed from the text and replaced by the actual value of the referenced variable. For example, the variables '`component`' and '`retcode`' expand to the name of the exited component and its exit code, correspondingly. Supposing that '`component`' is '`ftpd`' and '`retcode`' is 76, the following fragment:

```
Subject: ${component} exited with code ${retcode}
```

will become:

```
Subject: ftpd exited with code 76
```

The table below lists all available variables and their expansions:

Variable	Expansion
`canonical_program_name`	‘`pies`’
`program_name`	Program name of the `pies` binary.
`package`	Package name (‘`GNU Pies`’).
`instance`	Instance name (see [instances], page 65).
`version`	Package version (1.8).
`component`	Name of the terminated component.
`termination`	Termination cause (see below).
`retcode`	Component exit code (or signal number, if exited on signal), in decimal.

Table 3.4: Notification Variables

The ‘`termination`’ variable is set so as to facilitate its use with the ‘`retcode`’ variable. Namely, its value is ‘`exited with`’, if the component exited and ‘`terminated on signal`’, if it terminated on a signal. Thus, using

```
${termination} ${retcode}
```

results in a correct English sentence. This message, however, cannot be properly internationalized. This will be fixed in the future versions.

If `message` statement is not given, the following default message is used instead:

```
From: <>
X-Agent: ${canonical_program_name} (${package} ${version})
Subject: Component ${component} ${termination} ${retcode}.
```

Notification messages are sent using an external program, called *mailer*. By default it is `/usr/sbin/sendmail`. You can change it using the following configuration statement:

`mailer-program` *`prog`* [Config]

Use *prog* as a mailer program. The mailer must meet the following requirements:

1. It must read the message from its standard input.
2. It must treat the non-optional arguments in its command line as recipient addresses.

For example, the following statement instructs `pies` to use `exim` as a mailer:

```
mailer-program /usr/sbin/exim;
```

By default, the mailer program is invoked as follows:

```
/usr/sbin/sendmail -oi -t rcpts
```

where *rcpts* is a whitespace-separated list of addresses supplied in the ‘`notify`’ statement.

The mailer command may be altered using ‘`mailer-command-line`’ statement:

`mailer-command-line` *`string`* [Config]

Set mailer command line. Notice, that *string* must include the command name as well. The ‘`mailer-program`’ statement supplies the full name of the binary which will be executed, while the first word from the 'mailer-command-line' argument gives the string it receives as 'argv[0]'.

The example below shows how to use this statement to alter the envelope sender address:

```
mailer-command-line "sendmail -f root@domain.com -oi -t";
```

3.5 Access Control Lists

Access control lists, or ACLs for short, are lists of permissions that control access to 'inetd', 'accept' and 'meta1'-style components.

An ACL is defined using `acl` block statement:

`acl` [Config]

```
acl {
  definitions
}
```

This statement is allowed both in global context and within a 'component' block. If both are present, the global-level ACL is consulted first, and if it allows access, the component ACL is consulted. As a result, access is granted only if both lists allow it.

A *named ACL* is an access control list which is assigned its own name. Named ACLs are defined using the 'defacl' statement:

`defacl` *name* [Config]

```
defacl name {
  definitions
}
```

The *name* parameter specifies a unique name for that ACL. Named ACLs are applied only if referenced from another ACL (either global or a per-component one, or any named ACL, referenced from these). See [acl-ref], page 32, below.

In both forms, the part between the curly braces (denoted by *definitions*), is a list of *access control statements*. There are two types of such statements:

`allow` [*user-group*] *sub-acl host-list* [Config: acl]
`allow` *any* [Config: acl]

Allow access to the component.

`deny` [*user-group*] *sub-acl host-list* [Config: acl]
`deny` *any* [Config: acl]

Deny access to the component.

All parts of an access statement are optional, but at least one of them must be present. The *user-group* part is reserved for future use and is described in more detail in Appendix B [User-Group ACLs], page 77.

The *sub-acl* part, if present, allows to branch to another ACL. The syntax of this part is:

```
acl name
```

where *name* is the name of an ACL defined previously in '`defacl`' statement.

The *host-list* group allows to match client addresses. It consists of the `from` keyword followed by a list of *address specifiers*. Allowed address specifiers are:

addr
: Matches if the client IP equals *addr*. The latter may be given either as an IP address or as a host name, in which case it will be resolved and the first of its IP addresses will be used.

addr/netlen
: Matches if first *netlen* bits from the client IP address equal to *addr*. The network mask length, *netlen*, must be an integer number in the range from 0 to 32. The address part, *addr*, is as described above.

addr/netmask
: The specifier matches if the result of logical AND between the client IP address and *netmask* equals to *addr*. The network mask must be specified in "dotted quad" form, e.g. '`255.255.255.224`'.

filename
: Matches if connection was received from a UNIX socket *filename*, which must be given as an absolute file name.

The special form '`allow any`' means to allow access unconditionally. Similarly, '`deny any`', denies access unconditionally. Normally, one of these forms appears as the last statement in an ACL definition.

To summarize, the syntax of an access statement is:

```
allow|deny [acl name] [from addr-list]
```

where square brackets denote optional parts.

When an ACL is checked, its entries are tried in turn until one of them matches, or the end of the list is reached. If a matched entry is found, its command verb, `allow` or `deny`, defines the result of the ACL check. If the end of the list is reached, the result is '`allow`', unless explicitly specified otherwise (using the [acl-any], page 33.)

For example, the following ACL allows access for anybody coming from networks '`192.168.10.0/24`' and '`192.168.100.0/24`', or any connection that matches the named ACL '`my-nets`' (which is defined elsewhere in the configuration file). Access is denied for anybody else:

```
acl {
    allow from (192.168.10.0/24, 192.168.100.0/24);
    allow acl "my-nets";
    deny all;
}
```

3.6 The Control Statement

The *control interface* provides a method for communication with the running `pies` instance. It is used by the `piesctl` utility to query information about the instance and components it is currently running and to send it commands for controlling its operation (see Chapter 5 [piesctl], page 43). By default the UNIX socket `/tmp/pies.ctl` is used for this purpose. If `pies` was started with the `--instance=name` option, the socket is named `/tmp/name.ctl`.

Whatever its name, the socket will be owned by the user `pies` runs as (see Section 3.11 [Pies Privileges], page 39) and will have access rights of 0500, allowing only that user to read and write to it. When `pies` is used as init process, the default socket name is `/dev/init.ctl`.

`control` [Config]

The '`control`' statement configures the control interface and limits access to it:

```
control {
    socket url;
    acl { ... }
    admin-acl { ... }
    user-acl { ... }
    realm name;
}
```

`socket` *`url`* [Config: control]

URL of the control socket. The *url* argument is a string of the following syntax:

inet://*ip*:*port*

Listen on IPv4 address *ip* (may be given as a symbolic host name), on port *port*.

local://*file*[;*args*]
file://*file*[;*args*]
unix://*file*[;*args*]

Listen on the UNIX socket file *file*, which is either an absolute or relative file name. Optional arguments *args* control ownership and file mode of *file*. They are a semicolon-separated list of assignments to the following variables:

`user`	User name of the socket owner.
`group`	Owner group of the socket, if it differs from the `user` group.
`mode`	Socket file mode (octal number between '`0`' and '`777`').
`umask`	Umask to use when creating the socket (octal number between '`0`' and '`777`').

`idle-timeout` *`n`* [Config: control]

Disconnect any control session that remains inactive for *n* seconds. This statement is reserved for use in the future. Currently (as of version 1.8) it is a no-op.

The control interface is protected by three access control lists (See Section 3.5 [ACL], page 32, for a discussion of their syntax).

`acl` *`name`* [Config: control]
`acl` *{ . . . }* [Config: control]

Controls who can connect to the interface. The first form refers to a named ACL that must have been defined earlier by `defacl` statement (see [defacl], page 32). Use the second form to define a new ACL in place.

`user-acl` *name* [Config: control]

`user-acl` *{ . . . }* [Config: control]

Control interface provides two kinds of operations: *read-only* (such as getting information about running components) and *write* operations (such as stopping or restarting components).

The `user-acl` controls read access. Access to particular components can also be controlled individually, using the per-component `list-acl` statement (see Section 3.3.11 [Visibility], page 26).

`admin-acl` *name* [Config: control]

`admin-acl` *{ . . . }* [Config: control]

Controls write access to the `pies` instance itself and to the components for which no specific `admin-acl` statements are supplied (see Section 3.3.11 [Visibility], page 26).

In particular, whoever passes `admin-acl` can issue commands for stopping the instance and reloading its configuration.

When checking whether the user has a particular kind of access to a component, first the corresponding ACL from the `control` section is checked. If it allows access, then the per-component ACL is tried. If it allows access too, then the operation is permitted.

`realm` *name* [Config: control]

Defines the realm for basic authentication. Default value is '`pies`'.

3.7 User Identities for Accessing Control Interface

Privileges for using and performing various commands over the control interface can be distributed among several users. For example, it is possible to grant some users the rights to only view the component listing, or even to further limit their rights to only see the components they are authorized to know about. Another user may be able to stop or restart components and so on. This privilege separation requires `pies` to have a notion of user and be able to authenticate it.

Identity provider is an abstract mechanism that `pies` uses to obtain information about the user trying to authenticate himself for accessing a particular control function. As of version 1.8, this mechanism is considered experimental. That means, that although being fully functional, it can change considerably in future releases.

Identity provider supports two operations: authenticating a user, and checking if he is a member of particular *group*. It is defined in the configuration file using the `identity provider` statement.

`identity-provider` *name* [Config]

Defines an identity provider. It is a block statement:

```
identity-provider name {
  type type;
  ...
}

The provider name is used in diagnostic messages.
```

The only required substatement is `type`, which defines the type of the provider. Rest of statements (represented by `...` above) depends on the type.

Pies version 1.8 supports identity providers of two types: '`system`' and '`pam`'.

The '`system`' identity provider uses system user database for authentication and system group database for checking group membership. It is declared using the following statement:

```
identity-provider name {
    type system;
}
```

Obviously, to use the system identity provider for authentication, `pies` must be run as root.

The '`pam`' identity provider uses the Pluggable Authentication Modules (PAM) for authentication, and system group database for checking group membership.

```
identity-provider name {
    type pam;
    service srv;
}
```

The '`service`' statement defines the name of PAM service to use for authentication. If absent, the name '`pies`' is used.

Any number of different identity providers can be declared in the configuration file. When authenticating the user, they will be tried in turn until the one is found where authentication succeeds. Subsequent group membership checks will then use this identity provider.

3.8 Using `inetd` Configuration Files

In addition to its native configuration file format, GNU `pies` is able to read configuration files of several other widely-used utilities. One of these is `inetd`. The simplest way to use such configuration files is by including them to your main `pies.conf` using the *include-inetd* statement:

`include-inetd` ***file*** [Config]

Read components from `inetd`-style configuration file *file*. The argument may also be a directory, in which case all regular files from that directory are read and parsed as `inetd`-style configuration files.

The components read from *file* are appended to the `pies` list of components in order of their appearance.

For example, the following statement reads components from the standard `inetd` configuration file:

```
include-inetd /etc/inetd.conf;
```

Any number of `include-inetd` may be specified. For example, the following reads the contents of the `/etc/inetd.conf` configuration file and all files from the `/etc/inetd.d` directory:

```
include-inetd /etc/inetd.conf;
include-inetd /etc/inetd.d;
```

Another way to read `inetd` configuration files is to supply them in the command line, like this:

```
pies --syntax=inetd --config-file /etc/inetd.conf
```

Notice the `--syntax` option (see [config syntax], page 5). It informs `pies` that the following files are in `inetd` format. Of course, several configuration file may be given:

```
pies --syntax=inetd \
     --config-file /etc/inetd.conf --config-file /etc/inetd.d
```

A special option is provided that instructs `pies` to behave as `inetd`:

`--inetd` Read configuration from *sysconfdir*`/inetd.conf` and make sure `pies` state files (see Section 3.12 [State Files], page 40) do not conflict with those from other `pies` instances.

The GNU Pies package also provides a wrapper that allows to use `pies` instead of `inetd`. It is built if the package is configured with the `--enable-inetd` option. The wrapper is then installed in *sbindir* as `inetd`, possibly replacing the system binary of that name.

The command line usage of the `inetd` wrapper is entirely compatible with that of the usual `inetd` utility, i.e.:

```
inetd [option] [config [config...]] [-- pies-options]
```

Options are:

`-d` Increase debug level.

`-R` *rate* Set maximum rate (see [max-rate], page 22).

For convenience, the following additional options are understood:

`-t`
`--lint` Parse configuration file or files and exit. See [lint], page 5.

`-s`
`--status` Display info about the running instance. See [pies-status], page 65.

`-S`
`--stop` Stop the running instance.

Finally, any additional options `pies` understands may be given to `inetd` after the '`--`' separator.

3.9 Using MeTA1 Configuration File

MeTA1 is a mail transfer agent of new generation, designed to replace Sendmail in the future (`http://www.meta1.org`). It has a modular structure, each module being a component responsible for a particular task. The components are configured in the MeTA1 configuration file `/etc/meta1/meta1.conf`.

`Pies` can take a list of components directly from MeTA1 configuration file:

`include-meta1` *file* [Config]

Parse *file* as MeTA1 configuration file and incorporate components defined there into the current component list.

For example:

include-meta1 /etc/meta1/meta1.conf;

Thus, you can use `pies` instead of the default MeTA1 program manager `mcp`. This is particularly useful if you use '`Mailfromd`' (`http://mailfromd.software.gnu.org.ua`) to control the mail flow.

To ensure compatibility with MeTA1, the components read from its configuration file are started in the reverse order (i.e. from last to first), and stopped in the order of their appearance in *file*.

The following `pies` statements are silently applied to all MeTA1 components:

```
allgroups yes;
stderr file compname.log
chdir queue-dir
```

Here, *compname* stands for the name of the component, and *queue-dir* stands for the name of MeTA1 queue directory. The latter is `/var/spool/meta1` by default. It can be changed using the following statement:

`meta1-queue-dir` *`dir`* [Config]

Set name of MeTA1 queue directory.

To override any default settings for a MeTA1 component, add a `command` section with the desired settings after including `meta1.conf`. For example, here is how to redirect the standard error of the '`smtps`' component to '`local1.debug`' syslog channel:

```
include-meta1 /etc/meta1/meta1.conf

component smtps {
  stderr syslog local1.debug;
}
```

3.10 Global Configuration

The statements described in this section affect `pies` behavior as a whole.

`env` *{ . . . }* [Config]

Modifies the environment for the running `pies` instance. The modified environment will be inherited by all processes started by `pies` in the course of its normal operation.

See Section 3.3.4 [Environment], page 15, for a detailed discussion of the `env` statement syntax.

`syslog` *{ . . . }* [Config]

This block statement configures logging via syslog. It has the following substatements:

`dev` *`address`* [Config: syslog]

Address of the socket the syslog daemon is listening on. By default, `/dev/log` is used.

The *address* argument is either the file name of the UNIX socket file or IPv4 address of the syslog collector optionally followed by the colon and port number (or symbolic service name). If the port number is not supplied, the '`syslog`' port (UDP) from `/etc/services` is used.

`facility` *`string`* [Config: syslog]

Set syslog facility to use. Allowed values are: '`user`', '`daemon`', '`auth`', '`authpriv`', '`mail`', '`cron`', '`local0`' through '`local7`' (case-insensitive).

`tag` *string* [Config: syslog]

Prefix syslog messages with this string. By default, the program name is used.

`umask` *number* [Config]

Set the default umask. The *number* must be an octal value not greater than '777'. The default umask is inherited at startup.

`limits` *arg* [Config]

Set global system limits for all pies components. See Section 3.3.3 [Resources], page 14, for a detailed description of *arg*.

`return-code` *{ . . . }* [Config]

Configure global exit actions. See Section 3.3.7 [Exit Actions], page 19, for a detailed description of this statement.

`shutdown-timeout` *number;* [Config]

Wait *number* of seconds for all components to shut down. Default is 5 seconds.

The normal shutdown sequence looks as follows:

1. Compute shutdown sequence that takes into account dependencies between components, so as to ensure that dependent components stop before their prerequisites. This sequence can be viewed using the `--list-shutdown-sequence` option.
2. For each stage in the shutdown sequence, send the *termination signal* to each component marked for that stage. By default, `SIGTERM` is used, but it can be changed for each component using the `sigterm` configuration statement (see [sigterm], page 13). Wait for the signalled components to terminate. If any of them remain running after `shutdown-timeout` seconds, send them the `SIGKILL` signal.
3. If any `shutdown` components are defined, start them and wait for their termination. If any components are left running after `shutdown-timeout` seconds, terminate them by sending the `SIGKILL` signal.

3.11 Pies Privileges

Normally, `pies` is run with root privileges. If, however, you found such an implementation for it, that requires another privileges, you may change them using the following three statements:

`user` *user-name* [Config]

Start `pies` with the UID and GID of this user.

`group` *group-list* [Config]

Retain the supplementary groups, specified in *group-list*.

`allgroups` *bool* [Config]

Retain all supplementary groups the user (as given with `user` statement) is a member of.

An example of such implementation is using `pies` to start `jabberd` components: `http://www.gnu.org.ua/software/pies/example.php?what=jabberd2`.

3.12 State Files

Pies uses several files to keep its state information. The directory which hosts these files is called *state directory*, it is usually `/var/pies` or `/usr/local/var/pies`). The state directory can be configured at run time:

`state-directory` *dir* [Config]

Set the program state directory.

The table below describes the files kept in the state directory. The *instance* in this table stands for the `pies` instance name (see [instances], page 65). Usually, it is '`pies`'.

instance`.pid`
: The *PID file*. It keeps the PID number of the running `pies` instance.

instance`.qotd`
: The *Quotation-of-the-day file*. It is used by the '`qotd`' built-in service (see [qotd], page 23).

The following statements allow to redefine state file names. Use them only if the defaults do not suit your needs, and neither the `state-directory` statement nor the `--instance` option can help:

`pidfile` *file* [Config]

Sets the PID file name.

`qotd-file` *file-name* [Config]

Sets the name of the '`quotation-of-the-day`' file.

The following statements are retained for compatibility with earlier `pies` versions. They are silently ignored:

`control-file` *arg* [Config]

`stat-file` *arg* [Config]

4 Pies Debugging

The amount of debugging information produced by `pies` is configured by the following statements:

`debug` *`level`* [Config]

Set debugging level. The *level* must be a non-negative decimal integer. In version 1.8 the following debugging levels are used:

1 Log all basic actions: starting and stopping of components, received incoming TCP connections, sending mails. Notify about setting limits. Log pre-startup actions (see Section 3.3.6 [Actions Before Startup], page 18).

2 Log setting particular limits. Log the recomputed alarms.

4 Dump execution environments

6 Debug the parser of MeTA1 configuration grammar.

7 Debug the lexical analyzer of MeTA1 configuration file.

`source-info` *`bool`* [Config]

This statement decides whether debugging messages should contain source information. To enable source information, use:

```
source-info yes;
```

This feature is designed for `pies` developers.

5 Communicating with Running `pies` Instances

The `piesctl` tool allows you to communicate with the running `pies` program. The invocation syntax is:

```
piesctl [options] command [args...]
```

The *command* determines the operation to perform. The following sections describe available commands in detail.

5.1 piesctl id – Return Info About the Running Instance

The `id` subcommand returns information about the `pies` instance organized as key-value pairs. When invoked without arguments, the following data are returned:

`package` Canonical package name.

`version` Version of `pies`.

`instance` Instance name (see [instances], page 65).

`binary` Full pathname of the `pies` executable file.

`argv` Command line arguments supplied upon its startup.

`PID` Process ID.

For example:

```
$ piesctl id
package: GNU Pies
version: 1.8
instance: pies
binary: /usr/sbin/pies
argv: /usr/sbin/pies --config-file=/etc/pies/pies.conf
PID: 15679
```

To request a subset of these data, give the items of interest as command line arguments:

```
$ piesctl id binary PID
binary: /usr/sbin/pies
PID: 15679
```

5.2 Instance Management

Two subcommands are provided for stopping and restarting `pies`.

`shutdown` [piesctl]

> Stop the running `pies` instance

`reboot` [piesctl]

> Restart `pies` instance. Upon receiving this command, `pies` will restart itself with the same command line arguments. Naturally, this means that all running components will be restarted as well.

These subcommands do nothing when init process is selected.

5.3 piesctl config – Configuration Management

`config` *file list* [piesctl]

List currently loaded configuration files.

`config` *file clear* [piesctl]

Clear configuration file list

`config` *file add* ***`syntax`*** ***`file`*** [piesctl]

Add *file* to the list of configuration files. *syntax* specifies its syntax: '`pies`', '`inetd`', '`meta1`', or '`inittab`'.

`config` *file del*[*ete*] ***`name`*** [***`name`***...] [piesctl]

Remove listed names from the list of configuration files.

`config` *reload* [piesctl]

Reload configuration.

5.4 Component Management

`list` [*`condition`*] [piesctl]

List configured components. When used without arguments, all components are listed. Otherwise, only processes matching *condition* are listed.

Each output line contains at least two columns. The first column lists the tag of the component. The second one contains *flags*, describing the type and status of the component. The first flag describes the type:

Flag	**Meaning**
3	SysV init '`ctrlaltdel`' component
A	Accept-style component
B	SysV init '`boot`' component
C	Respawn component
c	SysV init '`once`' component
D	SysV init '`ondemand`' component
E	Command being executed
F	SysV init '`powerfail`' component
f	SysV init '`powerwait`' component
I	Inetd-style component
i	SysV init '`sysinit`' component
k	SysV init '`kbrequest`' component
n	SysV init '`powerfailnow`' component
o	SysV init '`powerokwait`' component
P	Pass-style component
R	Output redirector
W	SysV init '`wait`' component
w	SysV init '`bootwait`' component

The second flag is meaningful only for components. Its values are:

Flag	**Meaning**
-	Disabled component
f	A finished 'once' component
L	Inetd listener
R	Running component
S	Component is stopping
s	Component is sleeping
T	Component is stopped

The next column lists the PID (for running components) or socket address (for Internet listeners), or the string 'N/A' if neither of the above applies.

If the component is sleeping, the time of its scheduled wake-up is listed in the next column.

Rest of line shows the component command line.

```
$ piesctl list
smtps/stderr R  4697
pmult/stderr R  4677
pmult/stdout R  4676
pmult        CR 4678 /usr/local/sbin/pmult
smar         CR 4680 smar -f /etc/meta1/meta1.conf -d 100
qmgr         CR 4691 qmgr -f /etc/meta1/meta1.conf
smtpc        CR 4696 smtpc -f /etc/meta1/meta1.conf
smtps        PR 4698 smtps -d100 -f /etc/meta1/meta1.conf
finger       IL inet+tcp://0.0.0.0:finger /usr/sbin/in.fingerd -u
eklogin      IL inet+tcp://0.0.0.0:eklogin /usr/sbin/klogind -k -c -e
kshell       IL inet+tcp://0.0.0.0:kshell /usr/sbin/kshd -k -c
eklogin      IR 13836  /usr/local/sbin/klogind -k -c -e
```

Use *condition* to select the components to list. In its simplest form, *condition* is one of the following *terms*:

`all` Selects all processes, including internal services, such as output redirectors.

`active` Selects only active components.

`component` *`tag`*
Selects the component with the given tag. See Section 3.3 [Component Statement], page 10.

`type` *`arg`* Selects processes of the given type. Argument is '`component`', to select only components, '`command`', to select commands or '`redirector`' to select output redirectors. When `piesctl list` is used without arguments, `type component` is assumed.

`mode` *`arg`* Selects components of the given mode (see Section 3.3 [Component Statement], page 10). E.g. to list '`inetd`' components:

```
piesctl list mode inetd
```

`status` *arg*

Selects processes with the given status. Argument is one of:

`finished` Component is finished.

`listener` Component is an inet listener.

`running` Component is running.

`sleeping` Component is sleeping.

`stopped` Component is stopped.

`stopping` Component has been sent the SIGTERM signal and `pies` is waiting for it to terminate.

A term may be preceded by the word '`not`' to indicate negation of the condition. For example, the following command will list inactive components:

```
piesctl list not active
```

Furthermore, terms can be combined in logical expressions using boolean '`and`' and '`or`' operators:

```
piesctl list type component and not mode inetd
```

Conjunction ('`and`') has higher precedence than disjunction ('`or`'). In complex expressions parentheses can be used to alter the precedence:

```
piesctl list type component \
        and \( status running or status sleeping \)
```

Notice that parentheses must be escaped to prevent them from being interpreted by the shell.

The following summarizes the syntax of *condition* in BNF:

```
<condition> ::= <disjunction>
<disjunction> ::= <conjunction> | <conjunction> "or" <disjunction>
<conjunction> ::= <unary> | <unary> "and" <conjunction>
<unary> ::= <term> | "not" <condition> | "(" <condition> ")"
<term> ::= "all" | "active" | <keyword> <value>
<keyword> ::= "type" | "mode" | "status" | "component"
<value> ::= <word> | <quoted-string>
<word> ::= <printable> | <word> <printable>
<printable> ::= "A" - "Z" | "a" - "z" | "0" - "9" |
                "_" | "." | "*" | ":" | "@" | "[" | "]" | "-" | "/"
<quoted-string> ::= """ <string> """
<string> ::= <char> | <string> <char>
<char> ::= <any character except "\" and """> | "\\" | "\""
```

`stop` *condition* [piesctl]

Stop components matching *condition*.

`start` *condition* [piesctl]

Start components matching *condition*.

`restart` *condition* [piesctl]

Restart components.

5.5 Init Process Management

The `piesctl telinit` command communicates with `pies` instance running as *init* process (PID 1). See Section 6.5 [piesctl telinit], page 55, for a detailed discussion.

5.6 `Piesctl` Command Line Options

`-c` *file*
`--config-file=`*file*
: Read configuration from *file* instead of the default `/etc/piesctl.conf`. See Section 5.7 [piesctl.conf], page 48, for its description.

`-d`
`--dump`
: Dump obtained responses verbatim. This is useful mainly for debugging purposes.

`-i` *inst*
`--instance=`*inst*
: Talk to `pies` instance *inst*.

`--no-netc`
`-N`
: Don't read `~/.netrc` file.

`-u` *url*
`--url=`*url*
: Specifies the URL of the communication socket. See [piesctl url], page 48, for a description of allowed URL forms.

`-v`
`--verbose`
: Enable verbose diagnostics.

Before parsing, configuration file is preprocessed using external command defined at build time (normally `m4`). The following options control this feature:

`-E`
: Show preprocessed configuration on stdout and exit.

`--no-preprocessor`
: Disable the use of the external preprocessor.

`--preprocessor=`*cmd*
: Use the command *cmd* as the external preprocessor, instead of the default `m4`.

`--define=`*sym*`[=`*value*`]`
`-D` *symbol*`[=`*value*`]`
: Define symbol *sym* as having *value*, or empty, if the *value* is not given.

`--include-directory=`*dir*
`-I` *dir*
: Add directory *dir* to the list of directories to be scanned for preprocessor include files.

`--undefine=`*sym*
`-U` *sym*
: Undefine symbol *sym*.

Finally, the following options can be used to obtain on-line assistance:

`--config-help`
: Show a terse reference to configuration file syntax and exit.

`-h`
`--help`
: Display command line help summary.

`--usage`
: Give a short usage message

`-V`
`--version`
: Show program version.

5.7 Configuration for `piesctl`

The configuration file `/etc/piesctl.conf` helps the `piesctl` tool to determine the URL of the control socket. This file is not mandatory, and its absence is not considered an error. Its syntax is similar to that of `/etc/pies.conf`. The following statements are defined:

`socket` *`url`* [piesctl.conf]
: Sets the default socket URL.

`source` *`ip`* [piesctl.conf]
: Sets the default source IP address. This is used if the control socket is of '`inet`' type.

`instance` *`name`* [piesctl.conf]
: Configures socket URL and (optionally) source address to use when communicating with the `pies` instance *name* (i.e., when invoked as `piesctl -i` *`name`*:

```
instance name {
    # Socket URL for that instance.
    socket url;
    # Source IP address.
    source ip;
}
```

Valid values for *url* in the above statements are:

inet://*ip*:*port*
: Use the IPv4 address *ip* (may be given as a symbolic host name), on port *port*.

local://*file*
file://*file*
unix://*file*
: Use the UNIX socket file *file*.

The following algorithm is used to determine the name of the communication socket:

1. If the `--url` (`-u`) option is given, use its argument.
2. Determine the instance name (*inst*). If the `--instance` (`-i`) is given, *inst* is its argument. Otherwise, assume *inst*='`pies`'.
3. If configuration file `/etc/piesctl.conf` exists, read it. On success:
   a. See if the `instance` *`inst`* statement is present and has `socket` substatement. If so, the argument to `socket` gives the socket URL.

   b. Otherwise, if global `socket` statement is present, its argument gives the URL.

4. Otherwise, suppose that `piesctl` is run on the same box where the target instance of `pies` is running, and see if the file `/etc/`*`inst`*`.conf` exists. If so, parse it as `pies` configuration file and look for `control` block statement. If it has `socket` statement, take its argument as the URL. See Section 3.6 [control], page 33.
5. If socket URL is not determined by these steps, assume `/tmp/`*`inst`*`.ctl`.

6 Init – parent of all processes

`Pies` can be executed directly by the kernel as a program responsible for starting all other processes (a process with PID 1). In this case it becomes also the parent of all processes whose natural parents have died and is responsible for reaping those when they die.

When invoked this way, `pies` reads its configuration from two files: `/etc/inittab` and `/etc/pies.init`. The former has traditional syntax (see [inittab], page 52) and is retained for compatibility with another '`init`' daemons, and the latter is in native `pies` format (see Section 3.1 [Syntax], page 6). Either of the files or even both of them can be missing.

The startup process passes through several states. Transition between states is controlled by *runlevel*, which also defines the set of components that must be executed. Startup states are:

`sysinit`
: System initialization state. This state marks the beginning of the startup process. Only root partition is mounted, and is usually read-only. `Pies` uses console to output diagnostic messages.

  Normally, the configuration instructs `pies` to execute at this point the system initialization script, which checks and mounts the necessary local file systems, initializes devices and loads kernel modules.

  The system then passes to '`boot`' state, unless the default runlevel is '`S`', in which case the '`single`' state is selected.

`boot`
: Upon entering the '`boot`' state, `pies` attempts to log the '`reboot`' login record into the system `utmp/wtmp` files and executes entries marked with `boot` and `bootwait` types. It then enters the '`normal`' state.

`single`
: This is a fallback state for single-user system. It is entered only if the '`S`' runlevel has been selected initially. Normally, this state is used for system maintenance. The configuration usually provides a component which executes a single-user shell when entering this state. If it does not, `pies` executes '`/sbin/sulogin`'.

`normal`
: Upon entering this state, `pies` assumes that components executed previously have brought the system to such condition where normal communication means can already be used. This means that the file systems have been mounted read-write and the `syslog` daemon is operating. Therefore `pies` opens its communication channels and redirects its diagnostic output to syslog facility '`daemon`'.

  Then it starts components scheduled for the default runlevel and begins its normal operation.

`Pies` communication channels are:

`/dev/initctl`
: A FIFO file for communication using legacy protocol (using `telinit`).

`/dev/init.ctl`
: UNIX socket for communication using `piesctl`.

6.1 Runlevels

Runlevel determines the set of components to be run in normal state. It is a decimal digit from '`0`' to '`9`' or letter '`S`'. Traditionally, runlevels are assigned as follows:

0 System halt.

1
S Single user mode.

3 Multiuser mode.

4 Multiuser with X11.

Additionally, three special runlevels '`a`', '`b`' and '`c`' can be used to start *on-demand* components without actually changing the runlevel. Once started, on-demand components persist through eventual runlevel changes.

6.2 Init Process Configuration

The two configuration files are read in this order: `/etc/inittab` first, then `/etc/pies.init`. The `/etc/inittab` file is a simple line-oriented file. Empty lines and lines beginning with '`#`' are ignored (except if '`#`' is followed by the word '`pies`', see below). Non-empty lines consist of 4 fields separated by colons:

```
id:runlevels:mode:command
```

where

id Component identifier. A string uniquely identifying this component.

runlevels List of the runlevels for which the component should be run. Runlevels are listed as a contiguous string of characters, without any whitespace or delimiters.

mode Component execution mode.

command Command to be executed and its arguments.

Component execution modes are:

`respawn` The basic execution mode. A *respawn* component is restarted each time it terminates. If it is restarted more than 10 times in 2 minutes, `pies` puts it in *sleeping* state for the next 5 minutes.

`off` Disabled component. The entry is ignored.

`boot` The process will be executed during system boot. The '`runlevel`' settings are ignored.

`bootwait` The process will be executed during system boot. No other components will be started until it has terminated. The '`runlevel`' settings are ignored.

`sysinit` The process will be executed during system boot, before any `boot` or `bootwait` entries. The '`runlevel`' settings are ignored.

`once` The process will be executed once when the specified runlevel is entered.

`wait` The process will be started once when the specified runlevel is entered. `Pies` will wait for its termination before starting any other processes.

`ctrlaltdel`
The process will be executed when `pies` receives the SIGINT signal. Normally this means that the CTRL-ALT-DEL combination has been pressed on the keyboard.

`kbrequest`
The process will be executed when a signal from the keyboard handler is received that indicates that a special key combination was pressed on the console keyboard.

`ondemand`
The process will be executed when the specified *ondemand* runlevel is called (‘`a`’, ‘`b`’ and ‘`c`’). No real runlevel change will occur (see [Ondemand runlevels], page 52). The process will remain running across any eventual runlevel changes and will be restarted whenever it terminates, similarly to `respawn` components.

`powerfail`
The process will be executed when the power goes down. `Pies` will not wait for the process to finish.

`powerfailnow`
The process will be executed when the power is failing and the battery of the external UPS is almost empty.

`powerokwait`
The process will be executed as soon as `pies` is informed that the power has been restored.

`powerwait`
The process will be executed when the power goes down. `Pies` will wait for the process to finish before continuing.

The special mode ‘`initdefault`’ declares the default runlevel. In the ‘`initdefault`’ entry, the *runlevels* field must consist of exactly one runlevel character. Rest of fields are ignored. For example, the following instructs `pies` that the default runlevel is ‘`3`’:

```
id:3:initdefault:
```

If no ‘`initdefault`’ entry is present, `pies` will ask the user to input the desired default runlevel upon entering the normal state.

Inittab must contain at least one entry with ‘`S`’ in *runlevels* field. This entry is used for system maintenance and recovery. If it is absent, `pies` adds the following default entry implicitly:

```
~~:S:wait:/sbin/sulogin
```

As an exception to traditional syntax, the ‘`#`’ followed by the word ‘`pies`’ (with any amount of white space in between) introduce a pragmatic comment that modifies the behavior of the configuration parser. The following such comments are understood:

`#pies pragma debug` *n*
Set debugging level *n* (a decimal number). See Chapter 4 [Pies Debugging], page 41.

`#pies pragma next` *`syntax file`*

After parsing `/etc/inittab`, read configuration from file *file*, assuming *syntax* (see [config syntax], page 5). Multiple '`next`' pragmas are allowed, the named files will be processed in turn.

The default set up is equivalent to specifying

```
#pies pragma next pies /etc/pies.init
```

`#pies pragma stop`

Stop parsing after this line. The remaining material is ignored.

Both the traditional `/etc/inittab` and pies-native `/etc/pies.init` files are entirely equivalent, excepting that, naturally, the latter is more flexible and gives much more possibilities in defining the system behavior. The declaration of a component in `/etc/pies.init` can contain all the statements discussed in Section 3.3 [Component Statement], page 10. The only difference is that runlevels to start the component is must be specified:

`runlevels` *`string`* [Config: component]

Specifies the runlevel to start the component in. The *string* argument is a string of runlevel characters.

For example, the inittab entry discussed above is equivalent to the following statement in `pies.init` file:

```
component id {
  mode mode;
  runlevels runlevels;
  command command;
}
```

The default runlevel is specified in `/etc/pies.init` using the following construct:

`initdefault` *`rl`* [Config]

Declare the default runlevel. The argument is the runlevel name. E.g.

```
initdefault 3;
```

If both `/etc/inittab` and `/etc/pies.init` are present, the latter can declare components with the same *id* as the ones declared in the former. In that case, the two entries will be merged, the latter one overriding the former. Thus, `/etc/pies.init` can be used to complement definitions in `inittab`. Consider, for example the following inittab entry:

```
upd:3:respawn:/usr/libexec/upload
```

If `pies.init` contains the following:

```
component upd {
    user nobody;
    stderr syslog local1;
}
```

the result will be equivalent to:

```
component upd {
    mode respawn;
    runlevels 3;
```

```
        command /usr/libexec/upload;
        user nobody;
        stderr syslog local1;
    }
```

6.3 Init Command Line

The runlevel to run in can be given as argument in the command line:

```
/sbin/pies 1
```

Apart from this, the following command line arguments are recognized:

`-s`
`single` Initialize default runlevel '`S`'.

`-b`
`emergency`
Run emergency shell `/sbin/sulogin`, prior to initialization.

6.4 Init Environment

Programs run from `pies` init process inherit a basic environment consisting of the following variables:

`PREVLEVEL=`*`L`*
Previous runlevel, or letter '`N`' if the runlevel hasn't been changed since startup.

`RUNLEVEL=`*`L`*
Current runlevel.

`CONSOLE=`*`device`*
Pathname of the console device file.

`INIT_VERSION="GNU Pies 1.8"`
Version of `pies`.

`PATH=/bin:/usr/bin:/sbin:/usr/sbin`

Once the system is booted up, the environment can be controlled using the `piesctl telinit environ` (or `pies -T -e`) command.

6.5 piesctl telinit

`piesctl` *telinit runlevel* [piesctl]
Report the runlevel and state of the process 1.

`piesctl` *telinit runlevel* **`n`** [piesctl]
Switch to runlevel *n*.

`piesctl` *telinit environ list* [*`NAME`*] [piesctl]
List the environment. If *NAME* is given, list only the value of that variable.

`piesctl` *telinit environ set* *`NAME=VALUE`* [piesctl]
Set variable *NAME* to *VALUE*. The environment is capable to hold at most 32 variables.

`piesctl` *telinit environ unset NAME* [piesctl]

Unset variable *NAME*.

6.6 The Telinit Command

When given the `-T` (`--telinit`) option, `pies` emulates the behavior of the traditional `telinit` command. This is a legacy way of communicating with the init process. The commands are sent via named pipe `/dev/initctl`. When the `-T` option is given, the rest of command line after it is handled as `telinit` options. The following command:

```
pies -T [-t n] r
```

tells init process to switch to runlevel *r*. Possible values for *r* are:

0 to 9
: Instructs init to switch to the specified runlevel.

S or s
: Tells init to switch to the single user mode.

a, b, or c
: Tells init to enable on-demand components with the specified runlevel. The actual runlevel is not changed.

Q or q
: Tells init to rescan configuration files.

The `-t` (`--timeout`) option sets the time to wait for processes to terminate after sending them the SIGTERM signal. Any processes that remain running after *n* seconds will be sent the SIGKILL signal. The default value is 5 seconds.

This usage is equivalent to the `piesctl telinit runlevel` command (see Section 6.5 [piesctl telinit], page 55).

The `-e` (`--environment`) option modifies the init process environment. Its argument is either a variable assignment '*name=value*' to set a variable, or the name of a variable to unset it. Several `-e` options can be given to process multiple variables in a single command. Note, however, that given *n* `-e` options, the total length of their arguments is limited to 367 - *n* bytes.

This option provides a limited subset of the functionality offered by the `piesctl telinit environ` command.

The table below summarizes all options available in `telinit` mode:

`-t` *n*
: Wait *n* seconds for processes to terminate after sending them the SIGTERM signal. Any processes that remain running after that time will be sent the SIGKILL signal. The default value is 5 seconds.

`-e` *var=value*
: Define environment variable *var* as having value *value*.

`-e` *var*
: Unset environment variable *var*.

7 Using Pies as Entrypoint for Docker Container

Another use for pies is as an entrypoint in a docker container. This is similar to the init mode described in the previous chapter in that pies runs with PID 1. However, in this case pies uses its regular configuration file.

When started with PID 1 from a docker container, pies tries to detect the fact automatically and switch to the entrypoint mode. As of version 1.8, this detection might fail in containers run under Kubernetes. For such cases, use the --no-init option to inform pies that it should run in entrypoint mode.

The following Dockerfile fragment illustrates how to configure pies to be run from a container:

```
COPY pies.conf /etc
ENTRYPOINT [ "/usr/sbin/pies", "--foreground", "--stderr" ]
```

It is supposed, of course, that the configuration file pies.conf is available in the same directory as Dockerfile.

It is a common practice to supply configuration settings via the environment variables. To implement it in pies.conf, use either expandenv or shell flag (see Section 3.3.5 [Early Environment Expansion], page 18). For example:

```
flags expandenv;
command "syslogd -n -R $LOGHOST";
```

This will expand the environment variable LOGHOST and pass its value as one of the arguments to syslog. The usual shell syntax is supported. For example, to provide a default value for the -R option above (in case LOGHOST is empty or undefined), use:

```
flags expandenv;
command "syslogd -n -R ${LOGHOST:-172.19.255.255}";
```

Quite often a need arises to expand environment variables in other parts of the configuration file and to conditionally exclude portions of configuration, depending on whether a particular variable is set. The following sections describe two approaches to solving this problem.

7.1 Expanding Environment Variables in GNU m4

Configuration preprocessing (see Section 3.2 [Preprocessor], page 8) can be used to conditionally enable parts of the pies.conf file, depending on the value of an environment variable. The technique described below assumes that you use GNU m4 as preprocessor.

Define the following two M4 macros:

CF_WITH_ENVAR *name text* [M4 macro]

Expands the environment variable *name* within *text*. The macro does so by temporarily redefining the symbol *name* to the value of the environment variable *name* and expanding *text*.

The definition of the macro is:

```
m4_define(`CF_WITH_ENVAR',m4_dnl
`m4_pushdef(`$1',m4_esyscmd(printf "$`$1'"))m4_dnl
$2`'m4_dnl
```

```
     m4_popdef(‘$1’)m4_dnl
     ’)
```

This macro allows you to use environment expansion where it is not normally supported. Consider, for example, this fragment:

```
     component {
       CF_WITH_ENVAR(‘WORKDIR’, ‘chdir "WORKDIR";’)
       ...
     }
```

If you set `WORKDIR=/var/wd` prior to invoking `pies`, it will actually expand to

```
     component {
       chdir "/var/wd";
       ...
     }
```

See Section 3.3.6 [Actions Before Startup], page 18, for details about the `chdir` statement.

`CF_IF_ENVAR` *`name if-set if-unset`* [M4 macro]

If the environment variable *name* is defined and has a non-empty value, expand *if-set*, otherwise expand *if-unset*. Expand each occurrence of *name* in *if-set* to the actual value of the environment variable.

Following is the definition of this macro:

```
     m4_define(‘CF_IF_ENVAR’,m4_dnl
     ‘CF_WITH_ENVAR(‘$1’,‘m4_ifelse($1,‘’,$3,$2)’)’)
```

This macro makes it possible to conditionally enable configuration file fragments depending on whether some environment variable is defined. E.g.:

```
     CF_IF_ENVAR(‘LOGHOST’,‘
     component logger {
       command "syslogd -n -R LOGHOST;
     }
     ’)
```

Place both macros in a single file and include it at the top of your `pies.conf` using the `m4_include` command (see Section 3.2.1 [m4], page 10).

7.2 Using xenv

Another way to expand environment variables in the configuration file is to use `xenv`. `xenv` is a specialized preprocessor that expands environment variables in its input. It is also able to conditionally include parts of text depending on whether the environment variable is defined. The program is described in `https://www.gnu.org.ua/software/xenv/`.

To use `xenv` as preprocessor, start `pies` as follows:

```
     pies --foreground --stderr --preprocessor="xenv -s"
```

The `-s` option instructs `xenv` to emit *synchronization lines*, that inform `pies` about actual location of configuration statements in case when the expansion adds or removes portions of text spanning several lines.

You can also combine the functionality of m4 and xenv by running

```
pies --foreground --stderr --preprocessor="xenv -s -m"
```

In this case xenv will automatically feed its output to the standard input of m4, started for this purpose.

By default, xenv uses the shell syntax to expand the variables. For example, in the following configuration file fragment, '$WORKDIR' will expand to the actual value of the WORKDIR environment variable:

```
component {
  chdir "$WORKDIR";
  ...
}
```

There are two ways to conditionally include portions of text. The first one is to use the '${X:+W}' construct. For example:

```
component {
  ${WORKDIR:+chdir "$WORKDIR";}
  ...
}
```

Another way is to use the xenv '$$ifset' (or '$$ifdef') statement:

```
component {
$$ifset WORKDIR
  chdir "$WORKDIR";
$$endif
  ...
}
```

The difference between '$$ifset X' and '$$ifdef X' is the same as between '${X:+W}' and '${X+W}', i.e. '$$ifset' tests whether the variable is set and not-null, and '$$ifdef' tests only whether it is set, no matter its value.

xenv extends the shell syntax by providing a *ternary* operator. The construct '${X|A|B}' expands to 'A' if the variable X is set and to 'B' otherwise (as usual, placing the colon before first '|' checks if the variable is set and not null). This allows for writing compact conditionals:

```
component syslogd {
   mode respawn;
   command "/sbin/syslogd -n ${LOGHOST:|-R $LOGHOST|-O /proc/1/fd/1}";
}
```

In this example syslogd is instructed to relay messages to the IP address specified by the LOGHOST variable and to send messages to the container stdout otherwise.

Using shell indirection operator '$' can be confusing in parts of pies configuration file that deal with environment variables by themselves. The common point of confusion is using env and command statements when shell or expandenv flag is set. For example:

```
component X {
   env {
      set "HOME=/var/lib/nobody";
   }
   flags shell;
   command "marb -C $HOME";
}
```

Here, the intent is to pass '/var/lib/nobody' as the command line argument to marb. However, if pies was started with xenv as preprocessor, the reference '$HOME' will be expanded by xenv at the early stage to whatever value the HOME variable had at pies startup. Consequently, when it comes to launching the 'X' component, the intended expansion won't take place.

There are three options to handle such cases:

1. Escape the '$'

   Use backslash to suppress expansion by xenv:

   ```
   component X {
      env {
         set "HOME=/var/lib/nobody";
      }
      flags shell;
      command "marb -C \$HOME";
   }
   ```

2. Use the *verbatim* operator

   This allows to reproduce the desired part of text verbatim. There are two verbatim operators: inline operator '$[...]' and block operator '$$verbatim ... $$end'. Examples:

   ```
   component X {
      env {
         set "HOME=/var/lib/nobody";
      }
      flags shell;
      $[command "marb -C $HOME";]
   }
   ```

   or

   ```
   component X {
      env {
         set "HOME=/var/lib/nobody";
      }
      flags shell;
   $$verbatim
      command "marb -C $HOME";
   $$end
   }
   ```

3. Change the indirection operator

   The indirection operator ‘$’ can be changed either globally, by using the -S option, or locally by using the ‘$$sigil’ statement. E.g.:

   ```
   $$sigil @
   # From this point on, $ looses its special meaning in xenv.

   component X {
      env {
         set "HOME=/var/lib/nobody";
      }
      flags shell;
      command "marb -C $HOME @FILE";
   }
   ```

   In the command line of this example, @FILE will be expanded by xenv when processing the configuration file, and $HOME will be expanded by shell (to the value ‘/var/lib/nobody’, set by the env statement) when pies will start the command.

8 Configuration Examples

In this section we provide several examples of working `pies` configuration files.

8.1 Simplest Case: Using Pies to Run Pmult

The example below runs `pmult` (see Section “pmult” in *Mailfromd Manual*) utility with the privileges of ‘`meta1`’ user. Both standard error and standard output are redirected to the syslog facility ‘`mail`’, priorities ‘`err`’ and ‘`info`’, correspondingly.

```
component pmult {
  command "/usr/local/sbin/pmult";
  user meta1s;
  stderr syslog mail.err;
  stdout syslog mail.info;
}
```

8.2 Using Pies to Run Pmult and MeTA1

The example below is a working configuration file for running `pmult` and all components of MeTA1, configured in `/etc/meta1/meta1.conf`. The global `return-code` statement is used to configure `pies` behavior for some exit codes.

```
# Sample pies configuration for running pmult and MeTA1

# Special handling for exit codes that mean the program was
# incorrectly used or misconfigured.
return-code (EX_USAGE, EX_CONFIG) {
  action disable;
  notify "root";
  message <<- EOT
    From: Pies <>
    X-Agent: ${canonical_program_name} (${package} ${version})
    Subject: Component ${component} disabled.

    Component "${component}" has terminated with code ${retcode},
    which means it encountered some configuration problem.
    I will not restart it automatically.  Please fix its configuration
    and restart it manually at your earliest convenience.

    To restart, run ‘‘${program_name} -R ${component}’’
    ---
    Wuff-wuff,
    Pies
  EOT;
}

component pmult {
  command "/usr/local/sbin/pmult";
```

```
    user meta1s;
    stderr syslog err;
    stdout syslog info;
  }

  include-meta1 "/etc/meta1/meta1.conf";
```

8.3 Running Pies as Inetd

This configuration file allows to run `pies` instead of `initd`. It starts two services: '`ftp`' and '`pop3d`', and restricts access to them to two local subnets:

```
acl {
   allow from 10.10.10.0/24;
   allow from 192.168.10.0/27;
   deny from any;
}

debug 3;

component ftp {
   mode inetd;
   socket "inet://0.0.0.0:21";
   umask 027;
   program /usr/sbin/ftpd
   command "ftpd -l -C";
}

component pop3d {
   mode inetd;
   socket "inet://0.0.0.0:110";
   program "/usr/sbin/pop3d";
   command "pop3d --inetd";
}
```

The following is almost equivalent configuration in `inetd` format:

```
ftp  stream tcp  nowait  root /usr/sbin/ftpd  ftpd -l -C
pop3 stream tcp  nowait  root /usr/sbin/pop3d pop3d --inetd
```

This configuration is "almost" equivalent, because the `inetd` format has no way of specifying ACLs and setting the umask.

9 Command Line Usage

When run without arguments, `pies` parses and loads the configuration file, detaches itself from the controlling terminal (becomes a daemon), and starts all components. Before actually starting up, it ensures that no another copy is already running, by looking for a PID file and verifying that the PID listed there is alive and responding. If another copy is running, `pies` refuses to start up.

It is often necessary to run several copies of `pies` with different configuration files. To support such usage, `pies` provides a notion of *instance*. Pies instance is an independent invocation of `pies` that uses a separate configuration file and separate state files (see Section 3.12 [State Files], page 40). Instances are created using the `--instance` option:

`--instance=`*`name`*

Read configuration from *`sysconfdir/name`*`.conf`, use *name* as the base name for state files (i.e., they become *`name`*`.pid`, *`name`*`.clt`, etc.) and tag all syslog messages with *name*.

For example, the following invocations create three instances of `pies`:

```
pies
pies --instance=inetd
pies --instance=mta
```

The first instance uses the default configuration and state files. The second one reads configuration from `/etc/inetd.conf`, and the third one reads it from `/etc/mta.conf`.

After startup, you can verify the status of the running process using the `--status` option.

```
$ pies --status
smtps/stderr R  4697
pmult/stderr R  4677
pmult/stdout R  4676
pmult        CR 4678 /usr/local/sbin/pmult
smar         CR 4680 smar -f /etc/meta1/meta1.conf -d 100
qmgr         CR 4691 qmgr -f /etc/meta1/meta1.conf
smtpc        CR 4696 smtpc -f /etc/meta1/meta1.conf
smtps        PR 4698 smtps -d100 -f /etc/meta1/meta1.conf
finger       IL inet+tcp://0.0.0.0:finger /usr/sbin/in.fingerd -u
eklogin      IL inet+tcp://0.0.0.0:eklogin /usr/sbin/klogind -k -c -e
kshell       IL inet+tcp://0.0.0.0:kshell /usr/sbin/kshd -k -c
eklogin      IR 13836  /usr/local/sbin/klogind -k -c -e
```

See [piesctl list], page 44, for a description of the output format.

You can restart any component by using the `--restart-component` (`-R`) option, e.g.:

```
$ pies -R pmult smtps
```

To stop all running components and shut down `pies`, use the `--stop` (`-S`) command line option:

```
$ pies --stop
```

If you modified the configuration file, you can instruct `pies` to read it again using the `--reload` (`-r`) command line option.

The `--status`, `--restart-component`, `--stop`, and `--reload` options actually run the `piesctl` command, which provides a powerful tool for managing `pies`. See Chapter 5 [piesctl], page 43, for a detailed description.

Two options are provided for verifying inter-component dependencies. The `--dump-depmap` option prints on the standard output the *dependency map*. This map is a square matrix with rows representing dependents and columns representing prerequisites. An '`X`' sign is placed on each crossing which corresponds to the actual dependency. For example:

```
$ pies --dump-depmap
Dependency map:
    0  1  2  3  4
 0
 1
 2     X
 3        X
 4     X  X

Legend:
 0: pmult
 1: smar
 2: qmgr
 3: smtpc
 4: smtps
```

This example corresponds to the configuration file shown in Section 8.2 [Hairy Pies], page 63. To illustrate how to read it, consider the 4th row of the table. According to the legend, number 4 means '`smtps`' component. There are two '`X`' marks: in columns 1 and 2. This means that '`smtps`' depends on '`smar`' and '`qmgr`'.

You can also list prerequisites explicitly:

```
$ pies --trace-prereq
qmgr: smar
smtpc: qmgr
smtps: smar qmgr
```

To list prerequisites for a particular component, give its name in the command line:

```
$ pies --trace-prereq smtps
smtps: smar qmgr
```

Any number of components can be given in the command line.

A counterpart option `--trace-depend` lists dependencies. Its usage is similar to the described above:

```
$ pies --trace-depend
smtps
smtpc
qmgr: smtps, smtpc
smar: smtps, qmgr
```

```
$ pies --trace-depend qmgr
qmgr: smtps, smtpc
```

10 Pies Invocation

This section summarizes `pies` command line options.

`--config-file=`*file*
`-c` *file*

Read configuration from *file*, instead of the default `/etc/pies.conf`.

See Chapter 3 [Configuration], page 5.

`--config-help`

Show configuration file summary. See Chapter 3 [Configuration], page 5.

`--define=`*sym*`[=`*value*`]`
`-D` *symbol*`[=`*value*`]`

Define symbol *sym* as having *value*, or empty, if the *value* is not given. See Section 3.2 [Preprocessor], page 8.

`--debug=`*level*
`-x` *level*

Set debug verbosity level. See Chapter 4 [Pies Debugging], page 41, for a description of *level*.

`--dump-depmap`

Dump dependency map. See [dump-depmap], page 66.

`-E`

Preprocess configuration file and exit. See Section 3.2 [Preprocessor], page 8.

`--force`

Force startup even if another instance may be running.

`--foreground`

Remain in foreground.

`--help`

Display a short usage summary and exit.

`--inetd`
`-i`

Run in `inetd`-compatibility mode. It is roughly equivalent to `pies --instance=inetd --syntax=inetd`. See Section 3.8 [inetd], page 36.

`--include-directory=`*dir*
`-I` *dir*

Add directory *dir* to the list of directories to be scanned for preprocessor include files.

`--instance=`*name*

Define the name of the `pies` instance. See [instances], page 65.

`--lint`
`-t`

`--no-init`

Don't assume *init mode* (see Chapter 6 [Init Process], page 51) if running with PID 1. See Chapter 7 [Docker Entrypoint], page 57.

`--list-shutdown-sequence`

List components in order of *shutdown sequence*. Each line lists the sequence stage number and the component name. See [shutdown sequence], page 39, for a detailed discussion of its meaning.

`--no-preprocessor`
: Disable the use of the external preprocessor.

  See Section 3.2 [Preprocessor], page 8.

`--preprocessor=`*`cmd`*
: Use the command *cmd* as the external preprocessor, instead of the default `m4`.

  See Section 3.2 [Preprocessor], page 8.

`--source-info`
: Show source info with debugging messages. See [source-info], page 41.

`--status`
`-s`
: Start `piesctl list` to obtain information about the running processes. See [piesctl list], page 44.

`--stderr`
: Log to standard error.

`--stop`
`-S`
: Stop the running instance. This is equivalent to running `piesctl shutdown`.

`--syntax=`*`type`*
: Define the syntax for parsing the configuration files specified by any `--config-file` options that follow this one. Possible values for *type* are:

  `pies` Native `pies` configuration. See Chapter 3 [Configuration], page 5.

  `inetd` ‘`Inetd`’-style configuration files. See [inetd.conf], page 71.

  `meta1` ‘`meta1`’-style configuration files. See Section 3.9 [include-meta1], page 37.

  `inittab` ‘`Inittab`’ file. See Chapter 6 [Init Process], page 51.

  See [config syntax], page 5, for a detailed description of this option.

`--syslog`
: Log to syslog. This is the default.

`--telinit`
`-T`
: Emulate the `telinit` legacy interface. The rest of command line following this option is processed as `telinit` options. See Section 6.6 [telinit command], page 56, for a detailed description of these.

`--trace-depend`
: List dependencies for components named in the command line. Without arguments, dependencies for each component are listed. See [trace-depend], page 66.

`--trace-prereq`
: List prerequisites for components named in the command line. Without arguments, prerequisites for each component are listed. See [trace-prereq], page 66.

`--rate=r`
: Set maximum connection rate (connections per second) for inetd-style components. See [inetd component rate], page 22.

`-r`
`--reload`
`--hup`
: Reread the configuration files. This is equivalent to running `piesctl config reload` (see [config reload], page 44).

`-R`
`--restart-component`

Restart components named in the command line. See [pies-restart], page 65.

`--version`

Display program version and license information and exit.

`--undefine=`*`sym`*
`-U` *`sym`* Undefine symbol *sym*. See Section 3.2 [Preprocessor], page 8.

`--usage` Display a short summary of available options and exit.

11 How to Report a Bug

Send bug-reports and suggestions to `bug-pies@gnu.org.ua`.

If you think you've found a bug, please be sure to include maximum information needed to reliably reproduce it, or at least to analyze it. The information needed is:

- Version of the package you are using.
- Compilation options used when configuring the package.
- Run-time configuration (`pies.conf` file and the command line options used).
- Detailed description of the bug.
- Conditions under which the bug appears.

Appendix A Inetd.conf Format

This appendix describes the format of inetd compatible configuration files. See Section 3.8 [inetd], page 36, for the discussion on how to use such files with GNU pies.

The inetd configuration file has line oriented format. Comments are denoted by a '#' at the beginning of a line. Empty lines and comments are ignored. Each non-empty line must be either a service definition, or address specification.

Service definition consists of at least 6 fields separated by any amount of the white space. These fields are described in the following table (optional parts are enclosed in square brackets):

[service-node:]service-name

The service-name entry is the name of a valid service in the file /etc/services. For built-in services (see Section 3.3.9.1 [builtin], page 23), the service name must be the official name of the service (that is, the first entry in /etc/services), or a numeric representation thereof. For TCPMUX services, the value of the 'service name' field consists of the string 'tcpmux' followed by a slash and the locally-chosen service name (see Section 3.3.9.2 [TCPMUX], page 24). Optionally, a plus sign may be inserted after the slash, indicating that pies must issue a '+' response before starting this server.

> The 'service-name' part corresponds to component tag in pies.conf (see Section 3.3 [Component Statement], page 10). For built-in components, it corresponds to the service statement (see Section 3.3.9.1 [builtin], page 23).

Optional 'service-node' prefix is allowed for internet services. When present, it supplies the local addresses inetd should listen on for that service. 'Service-node' consists of a comma-separated list of addresses. Both symbolic host names and numeric IP addresses are allowed. Symbolic hostnames are looked up in DNS service. If a hostname has multiple address mappings, a socket is created to listen on each address. A special hostname '*' stands for INADDR_ANY.

socket type

The socket type should be one of 'stream', 'dgram', 'raw', 'rdm', or 'seqpacket'. TCPMUX services must use 'stream'.

> This field corresponds to the socket-type statement in pies.conf. See [socket-type], page 22.

protocol

The protocol must be a valid protocol as given in /etc/protocols. Examples might be 'tcp' or 'udp'. TCPMUX services must use 'tcp'.

The '`service-node`' prefix and '`socket-type`' field correspond to the `socket` statement in `pies.conf`. See [inetd-socket], page 21.

For example, the following line:

```
10.0.0.1:ftp dgram   udp     wait    root  ftpd
```

is equivalent to

```
  socket inet+udp://10.0.0.1:ftp;
  socket-typle dgram;
```

wait/nowait[.max-rate]

The '`wait/nowait`' entry specifies whether the invoked component will take over the socket associated with the service access point, and thus whether `pies` should wait for the server to exit before listening for new service requests. Datagram servers must use '`wait`', as they are always invoked with the original datagram socket bound to the specified service address. These servers must read at least one datagram from the socket before exiting. If a datagram server connects to its peer, freeing the socket so that `pies` can go on receiving further messages from the socket, it is said to be a *multi-threaded* server; it should read one datagram from the socket and create a new socket connected to the peer. It should fork, and the parent should then exit to allow `pies` to check for new service requests to spawn new servers. Datagram servers which process all incoming datagrams on a socket and eventually time out are said to be *single-threaded*. Examples of such servers are `comsat` and `talkd`. `tftpd` is an example of a multi-threaded datagram server.

Servers using stream sockets generally are multi-threaded and use the '`nowait`' entry. Connection requests for these services are accepted by `pies`, and the server is given only the newly-accepted socket connected to a client of the service. Most stream-based services and all TCPMUX services operate in this manner. For such services, the invocation rate may be limited by specifying optional '`max-rate`' suffix (a decimal number), e.g.: '`nowait.15`'.

Stream-based servers that use '`wait`' are started with the listening service socket, and must accept at least one connection request before exiting. Such a server would normally accept and process incoming connection requests until a timeout. Datagram services must use '`nowait`'. The only stream server marked as '`wait`' is `identd` (see Section "identd" in *identd manual*).

The '`wait`' field corresponds to `flags wait` in the `pies.conf` file. The '`nowait`' corresponds to `flags nowait`. See [flags], page 12.

The '`max-rate`' suffix corresponds to the `max-rate` statement. See [max-rate], page 22.

user

The user entry contains the name of the user as whom the component should run. This allows for components to be given less permission than root.

This corresponds to the `user` statement in `pies.conf`. See Section 3.3.2 [Component Privileges], page 14.

program The program entry contains the full file name of the program which is to be executed by pies when a request arrives on its socket. For built-in services, this entry should be 'internal'.

It is common usage to specify /usr/sbin/tcpd in this field.

> This field corresponds to the program statement in pies.conf. See Section 3.3 [Component Statement], page 10.

server program arguments

The server program arguments should be just as arguments normally are, starting with argv[0], which is the name of the program. For built-in services, this entry must contain the word 'internal', or be empty.

> This corresponds to the command statement. See Section 3.3 [Component Statement], page 10.

Address specification is a special statement that declares the 'service-node' part (see above) for all the services declared below it. It consists of a host address specifier followed by a colon on a single line, e.g.:

```
127.0.0.1,192.168.0.5:
```

The address specifier from such a line is remembered and used for all further lines lacking an explicit host specifier. It remains in effect until another address specification or end of the configuration is encountered, whichever occurs first.

The following address specification:

```
*:
```

causes any previous default address specifier to be forgotten.

An example of inetd.conf file with various services follows:

```
ftp            stream  tcp nowait root  /usr/libexec/ftpd    ftpd -l
ntalk          dgram   udp wait   root  /usr/libexec/ntalkd  ntalkd
tcpmux         stream  tcp nowait root  internal
tcpmux/+scp-to stream  tcp nowait guest /usr/sbin/in.wydawca wydawca
tcpmux/docref  stream  tcp nowait guest /usr/bin/docref      docref
```

Appendix B User-Group ACLs

This appendix describes the '`user-group`' extension for GNU `Pies` ACLs. This extension is reserved for the future use.

The *user-group* ACL statement specifies which users match this entry. Allowed values are the following:

`all` All users.

`authenticated`
Only authenticated users.

`group` *`group-list`*
Authenticated users which are members of at least one of groups listed in *group-list*.

For example, the following statement defines an ACL which allows access for any user connected via local UNIX socket `/tmp/pies.sock` or coming from a local network '`192.168.10.0/24`'. Any authenticated users are allowed, provided that they are allowed by another ACL '`my-nets`' (which should have been defined before this definition). Users coming from the network '`10.10.0.0/24`' are allowed if they authenticate themselves and are members of groups '`pies`' or '`users`'. Access is denied for anybody else:

```
acl {
    allow all from ("/tmp/pies.sock", "192.168.10.0/24");
    allow authenticated acl "my-nets";
    allow group ("pies", "users") from "10.10.0.0/24";
    deny all;
}
```

Appendix C Control API

This appendix describes *control API* used to communicate with the running `pies` daemon via the control interface (see Section 3.6 [control], page 33). This API is used by `piesctl` (see Chapter 5 [piesctl], page 43).

The API is designed as a REST service and uses HTTP. Queries are sent to pies *endpoints*, each of which serves a distinct purpose. Data are serialized using the JSON format.

The sections below describe in detail each endpoint and associated with it request types.

C.1 /instance

This endpoint controls the state of the running `pies` instance and accepts the following HTTP requests: `GET`, `DELETE`, `POST` (or `PUT`).

`GET` */instance* [Request]

Retrieves information about the current instance. The response body is a JSON object with the following attributes:

‘`PID`’ PID of the running daemon.

‘`argv`’ Array of the command line arguments. ‘`argv[0]`’ is the program name.

‘`binary`’ Name of the `pies` binary.

‘`instance`’

The instance name. See [instances], page 65.

‘`package`’ Package name (the string ‘`GNU Pies`’).

‘`version`’ Package version

Any of these can be used in the URI to request the information about that particular attribute, e.g.:

```
GET /instance/argv ⇒ {"argv":["pies", "-x2"]}
```

`DELETE` */instance/PID* [Request]

Stops the current `pies` instance.

`PUT` */instance/PID* [Request]
`POST` */instance/PID* [Request]

Restarts the current `pies` instance.

C.2 /conf

The ‘`/conf`’ endpoint allows the client to inspect and change the configuration of the running `pies` instance.

C.2.1 /conf/files

`GET` */conf/files* [Request]

Return list of configuration files. On success, a JSON array is returned. Each array element is an object with two attributes:

`string` *file* [Attr]

Pathname of the configuration file.

`string` *syntax* [Attr]

Configuration file syntax (see Section 3.1 [Syntax], page 6).

For example:

```
GET /conf/files ⇒
[{"file":"/etc/pies.conf", "syntax":"pies"},
 {"file":"/etc/inetd.conf", "syntax":"inetd"}]
```

`POST` */conf/files* [Request]

Adds a new configuration file. The body must be a JSON object with '`file`' and '`syntax`' attributes, as described above. The '`file`' value must contain a pathname of a configuration file written in a syntax supplied by the '`syntax`' attribute (see Section 3.1 [Syntax], page 6).

This request returns 201 code on success. To actually parse and load the added configuration file, send a '`PUT`' request to '`/conf/runtime`' (see Section C.2.2 [/conf/runtime], page 80).

`DELETE` */conf/files/true* [Request]

Clears all previously configured configuration files. Responds with:

```
{ "message":"file list cleared", "status":"OK" }
```

`DELETE` */conf/files/*[*`list`*] [Request]

Removes files named in the *list* from the list of configuration files.

The '`DELETE`' response is 200 on success. To actually update the configuration of the running process, send a '`PUT`' request to '`/conf/runtime`' (see Section C.2.2 [/conf/runtime], page 80).

C.2.2 /conf/runime

This is a write-only URI. The only request supported is '`PUT /conf/runtime`'. It initiates reloading of the `pies` configuration. Usually, this request is sent after one or more '`POST`' and/or '`DELETE`' requests to '`/conf/files`', in order to finalize the changes applied to the configuration.

C.3 /programs

A request sent to this URI selects one or more components and applies operation defined by the request type to all of them.

Components are selected using a query in the form of JSON object (a *selector*). Valid selectors are:

'`null`'
'`false`' Matches nothing.

'`true`' Matches all components.

'`{ "op": "component", "arg":` *`tag`* `}`'
Matches component with the given *tag* (see [tag], page 10).

'`{ "op": "type", "arg": "component" }`'
: Matches all components.

'`{ "op": "type", "arg": "command" }`'
: Matches all commands.

'`{ "op": "mode", "arg":` *mode* `}`'
: Matches all components with the given *mode*. See [component mode], page 11.

'`{ "op": "active" }`'
: Matches all active components.

'`{ "op": "status", "arg":` *status* `}`'
: Matches all components with the given *status* (one of '`stopped`', '`running`', '`listener`', '`sleeping`', '`stopping`', '`finished`'). See [component status], page 82, for a discussion of these values.

'`{ "op: "not", "arg":` *condition* `}`'
: Negates *condition*, which is any valid selector.

'`{ "op": "and", "arg":` *array* `}`'
: Returns the result of logical conjunction on the *array* of selectors.

'`{ "op": "or", "arg":` *array* `}`'
: Returns the result of logical disjunction on the *array* of selectors.

For example, the following selector matches all components that are in '`running`' state, excepting components of '`inetd`' mode:

```
{ "op": "and",
   "arg": [ { "op": "type", "arg": "component" },
            { "op": "not", "arg": { "op": "mode", "arg": "inetd" }
          ]
}
```

The following requests are supported:

`GET` */programs?*`selector` [Request]
`GET` */programs/*`tag` [Request]

This request returns information about components matched by *selector* (see below for the '`/programs/tag variant`'. The response is a JSON array of descriptions. If no component matches the *selector*, empty array is returned. Each description is a JSON object with the following attributes:

`string type` [Attr]

Type of the described entity: '`component`' for an instance of a configured component, and '`command`' for a command run as a part of exit action (see Section 3.3.7 [Exit Actions], page 19), including mailer invocations (see Section 3.4 [Notification], page 30).

`string mode` [Attr]

Mode of the entity. See [component mode], page 11.

`string status` [Attr]

Entity status. Possible values are:

`finished` A '`once`' or '`startup`' component has finished.

`listener` Component is an inetd listener.

`running` Component is running.

`sleeping` Component has been put to sleep because of excessive number of failures (see [respawn], page 1).

`stopped` Component is stopped.

`stopping` Component is being stopped (a `SIGTERM` was sent).

`boolean active` [Attr]

Whether this component is active. By default, all components are active, unless marked with a '`disable`' flag (see [flags], page 12) or administratively stopped.

`integer PID` [Attr]

PID of the running process.

`string URL` [Attr]

(for '`inetd`' components) URL of the socket the component is listening on.

`string service` [Attr]

(for '`tcpmux`' components) TCPMUX service name. See Section 3.3.9.2 [TCPMUX], page 24.

`string master` [Attr]

(for '`tcpmux`' components) Tag of master TCPMUX component. See Section 3.3.9.2 [TCPMUX], page 24.

`string runlevels` [Attr]

For inittab components, the string of runlevels this component is configured to run in. See Chapter 6 [Init Process], page 51.

`integer wakeup-time` [Attr]

If component is in the '`sleeping`' state, this attribute gives the number of seconds after which an attempt will be made to restart it.

`array argv` [Attr]

Component command line split into words.

`string command` [Attr]

Component command.

`DELETE` */programs?*`selector` [Request]
`DELETE` */programs/*`tag` [Request]

Stop components matched by the *selector*. On success returns:

```
{ "status":"OK" }
```

On failure, returns

```
{ "status":"ER", "message": text }
```

where *text* is a textual human-readable description of the failure.

`PUT` */programs?`selector`* [Request]
`PUT` */programs/`tag`* [Request]

Start components matched by *selector*.

`POST` */programs* [Request]

Restart components. The selector is supplied in the request content.

Wherever a selector is passed via query parameters, a simplified form with component tag passed as query path is also allowed. For example:

```
 GET /programs/tag
```

is a shortcut for:

```
{ "op":"and",
   "arg":[ {"op":"type", "arg":"component"},
           {"op":"component", "arg":tag } ] }
```

C.4 /alive

This entry point accepts only '`GET`' requests. The URI must not be empty and must not include sub-directories (parts separated with slashes). It is treated as the name of the component to return the status of. E.g. querying '`/alive/foo`' returns the status of the component named '`foo`'. The status is returned as HTTP status code:

200 The component is up and running. For regular components that means that the corresponding program is running. For '`inetd`' components that means that the listener is listening on the configured socket.

403 No component specified.

404 There is no such component.

503 The component is not running. This means that it has failed, or has been stopped administratively or (for '`once`' and '`startup`' components) that it has run once and finished.

If the component has failed, the '`Retry-After:`' HTTP header contains the number of seconds after which `pies` will retry starting this component.

C.5 /runlevel

This URI is active when `pies` runs as init process (see Chapter 6 [Init Process], page 51). It supports two requests:

`GET` */runlevel* [Request]

Returns the current state of the program as a JSON object with the following attributes:

`string` *runlevel* [Attr]

Current runlevel. See Section 6.1 [Runlevels], page 52.

`string` *prevlevel* [Attr]

Previous runlevel ('`N`' if none).

`string` *bootstate* [Attr]

Boot state. See [startup states], page 51.

`string` *initdefault* [Attr]

Default runlevel.

`PUT` */runlevel/{"runlevel":L}* [Request]

Initiates transition from the current runlevel to runlevel *L* (see Section 6.1 [Runlevels], page 52).

C.6 /environ

This URI is active when `pies` runs as init process (see Chapter 6 [Init Process], page 51). It manipulates the program initial environment, i.e. the environment that all programs inherit. See Section 6.4 [Init Environment], page 55.

`GET` */environ/* [Request]

Returns entire environment formatted as a JSON array of strings. On success, the 200 response is returned:

```
["RUNLEVEL=3", "CONSOLE=/dev/tty", ...]
```

`GET` */environ/`var`* [Request]

Returns the value of the environment variable *var*, if such is defined. On success, the 200 response carries the object:

```
{ "status":"OK", "value":string }
```

If the variable *var* is not defined, a 404 response is returned. On error, a 403 response is returned. In both cases, the response body is the usual `pies` diagnostics object:

```
{ "status":"ER", "message":text }
```

`DELETE` */environ/`var`* [Request]

Deletes from the environment the variable *var*. On success, responds with HTTP 200:

```
{ "status":"OK" }
```

Error responses are the same as for ‘`GET`’.

`PUT` */environ/`name`=`value`* [Request]

Initializes environment variable *name* to *value*. See ‘`GET`’ for the possible responses.

Appendix D GNU Free Documentation License

Version 1.3, 3 November 2008

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

   If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

   If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

   If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

   It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

   You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

   A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

   You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

   The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

   In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

   However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

   Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

   Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `http://www.gnu.org/copyleft/`.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

    "Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

    "CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

    "Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

    An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

    The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.3
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts.  A copy of the license is included in the section entitled ‘‘GNU
Free Documentation License’’.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the “with. . . Texts.” line with this:

```
  with the Invariant Sections being list their titles, with
  the Front-Cover Texts being list, and with the Back-Cover Texts
  being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Concept Index

This is a general index of all issues discussed in this manual