

NAME

piper – using `spec` as the back end of a pipe

DESCRIPTION

`spec` contains certain facilities to aid in its use as the back end of a parent program such as a graphical user interface (GUI). This document describes those facilities.

INVOCATION FLAGS

`spec` needs to be invoked with several special flags when invoked from a parent process.

```
spec -T fake_tty -t fake_tty -p pid fd -q
```

`spec` is the name of the `spec` executable. The `-T` flag and `fake_tty` argument select the `tty` portion of the state file name used to save the user's state when `spec` exits. The `-t fake_tty` forces the current state to be restored from the state file associated with `fake_tty` when `spec` starts up.

The `-p` flag takes two arguments: the parent process ID and the file descriptor `spec` should use for its standard input. If a `tty`-based program is filtering input from the user before the commands are sent to `spec`, that program and `spec` cannot both reading from the standard input. For a GUI-type program, the `fd` argument can be zero. When `fd` is not zero, `spec` will echo text received from that file descriptor to its standard output. The `pid` argument must be present, but is currently unused.

The `-q` flag is optional. If present, all output from files and devices in `spec` may be turned off at the same time, including the "tty" output. Normally, if a user tries to turn off all output devices, `spec` automatically turns the output to "tty" back on.

INVOCATION CONTEXT

Before creating a process to execute `spec`, the parent process needs to create two or three pipes that will be used for interprocess communication. One pipe is for sending commands to `spec`'s standard input. The second pipe is for receiving special commands, described later, from `spec`. The optional third pipe receives `spec`'s standard output.

Sample code for spawning `spec` follows:

```
int    p_write[2];    /* pipe to spec */
int    p_read[2];    /* pipe for special messages */
int    p_out[2];     /* stdout pipe from spec */
int    spec_pid;     /* process id of spec */

#define PIPE_FD 4    /* File descriptor for special messages */

/* code to set up pipes, fork and exec spec */
exec_spec() {
    register int    i, pipe_fd;
    char    pid_buf[64], fd_buf[64];

    /*
     * the dup(0) is for the case where we don't
     * want spec reading from tty standard input.
     */
    pipe_fd = dup(0);
    sprintf(fd_buf, "%d", pipe_fd);
    sprintf(pid_buf, "%d", getpid());

    /* open all three pipes */
    if (pipe(p_write) || pipe(p_read) || pipe(p_out)) {
        perror("Can't pipe");
        return(-1);
    }
}
```

```

}
/* spawn a process */
if ((spec_pid = fork()) == 0) {
    /* Restore any caught signals to defaults ... */
    signal(SIGINT, SIG_DFL);

    /* associate read side of p_write with std input */
    if (p_write[0] != pipe_fd) {
        close(pipe_fd);
        dup(p_write[0]);
    }
    /* associate write side of p_out with std output */
    if (p_out[1] != 1) {
        close(1);
        dup(p_out[1]);
        /* also get standard error */
        close(2);
        dup(p_out[1]);
    }
    /* associate write side of p_read with PIPE_FD */
    if (p_read[1] != PIPE_FD) {
        /*
         * could also use dup2(p_read[1], PIPE_FD),
         * historically, though, dup2() has not been
         * robust across all platforms.
         */
        close(PIPE_FD);
        dup(p_read[1]);
    }
    /*
     * close other open files ... unlikely
     * anything past 40 is open?
     */
    for (i = 3; i < 40; i++)
        if (i != PIPE_FD && i != pipe_fd)
            close(i);

    execl("/cert/spec/fourc", "fourc",
          "-pq", pid_buf, fd_buf,
          "-T", "piper",
          "-t", "piper", (char *) 0);

    perror("Can't exec spec");
    _exit(1);
}
if (spec_pid < 0) {
    perror("Can't fork");
    return(-1);
}
/* close off unused ends of the pipes */
close(p_write[0]);
close(p_read[1]);
close(p_out[1]);
return(0);
}

```

SPECIAL MESSAGES

The following is a list of messages sent over the special pipe from `spec` to the parent process.

- [B] `errmsg {message}[E]` – Transmits error messages produced by the built-in C code and by the user-level `eprint` command and `eprintf()` function. Note the curly brackets can be used to help parse multi-line messages.
- [B] `spec_msg syntax_error parse_error[E]` – This message is sent when there is an parsing error in the user input. `spec` resets to command level.
- [B] `spec_msg error_reset error_reset[E]` – This message is sent on errors other than syntax errors that cause `spec` to reset to command level.
- [B] `spec_msg state_change shell_begin[E]` – This message is sent when `spec` creates a subshell using its built-in `unix()` command. Note, while `spec` is in a subshell, it will not respond to commands.
- [B] `spec_msg state_change shell_end[E]` – This message is sent when `spec` returns from a subshell.
- [B] `spec_msg done_waiting flags[E]` – This message is sent when `spec` returns from the `wait()` function and some hardware activity has completed. The bits in `flags` indicate which activity has completed. Bit 1 is set for motors, bit 2 for timers and bit 3 for other data acquisition.
- [B] `spec_msg needs_input motor_sync[E]` – This message is sent when `spec` requires a yes or no response to a motor position synchronization question. `spec` will not accept further input until it receives one of the characters `y, Y, 1, n, N` or `0`.
- [B] `spec_msg needs_input user_input[E]` – This message is sent when `spec` is expecting user input either from the `gethelp()` paginator, or from the `input()` or `getval()` built-in functions.
- [B] `spec_msg errmsg message[E]` – Not yet used.
- [B] `motor_sync motor_number 1 controller_position spec_position[E]` – This message is sent when there is a motor synchronization discrepancy. The arguments are the motor number, the digit 1, the motor position (in dial units) as read from the controller and the motor position (in dial units) in `spec`. (Currently only implemented for selected motor controllers.)
- [B] `spec_msg motor_limit motor_number direction[E]` – This message is sent when `spec` detects a motor has hit a hard limit switch. The first argument is the motor number and the `direction` argument is 1 or -1 indicating whether the positive or negative limit has been hit. (Currently only implemented for selected motor controllers.)

The parent process should be prepared to handle `motor_sync` `motor_sync` and `motor_limit` messages at any time.

Besides the special messages above, any `spec` output generated to the special output devices named "pipe" using `fprintf("pipe", ...)` or `on("pipe")` will arrive through the same file descriptor.

CONCLUSION

With the above facilities, one can control `spec` from a parent process in a variety of ways. One can completely hide `spec`'s output from the user, writing special macros that send terse message over the "pipe" stream. Alternatively, one can present the user with the standard `spec` input and output, filtered through the parent program, with the parent program sending additional messages to `spec` from its various GUI-type widgets.

