NAME

Modem::Vgetty − interface to vgetty(8)

SYNOPSIS

```
use Modem::Vgetty;
$v = new Modem::Vgetty;

$string = $v->receive;
$v->send($string);
$string = $v->expect($str1, $str2, ...);
$v->waitfor($string);
$rv = $v->chat($expect1, $send1, $expect2, $send2, ...);

$ttyname = $v->getty;
$rv = $v->device($dev_type);
$rv = $v->autostop($bool);
$rv = $v->modem_type; # !!! see the docs below.

$rv = $v->beep($freq, $len);
$rv = $v->dial($number);
$rv = $v->play($filename);
$rv = $v->record($filename);
$rv = $v->wait($seconds);
$rv = $v->play_and_wait($filename);
$v->stop;

$v->add_handler($event, $handler_name, $handler);
$v->del_handler($event, $handler_name);
$v->enable_events;
$v->disable_events;

$number = $v->readnum($message, $tmout, $repeat);

$v->shutdown;
```

DESCRIPTION

`Modem::Vgetty` is an encapsulation object for writing applications for voice modems using the ***vgetty*** **(8)** or ***vm*** **(8)** package. The answering machines and sofisticated voice applications can be written using this module.

OVERVIEW

Voice modem is a special kind of modem, which (besides the normal data and/or fax mode) can communicate also in voice mode. It means it can record sounds it hears from the phone line to the file, Play-back recorded files, it can beep to the line, and it can detect various standard sounds coming from the line (busy tone, silence, dual tone modulation frequency (DTMF) keypad tones, etc). An example of the voice modem can be the ZyXEL U1496, US Robotics Sportster (not Courier), etc.

To use this software with the voice modem you need to have the ***vgetty*** **(8)** package installed. **Vgetty** is distributed as a part of **mgetty** package. In fact, **vgetty** is a ***mgetty*** **(8)** with the voice extensions. Vgetty has some support for scripting − when it receives an incoming call, it runs a voice shell (it is program specified in the **voice.conf** file) as its child process, establishes the read and write pipes to it, and tells it the number of the appropriate descriptors in the environment variables. Voice shell can now communicate with **vgetty**. It can tell **vgetty** ‘‘Play this file’’, or ‘‘Record anything you hear to that file’’, or ‘‘Notify me when user hangs up’’, etc. Sophisticated voice systems and answering machines can be build on top of **vgetty**.

mgetty (including the **vgetty**) is available at the following URL:

```
ftp://alpha.greenie.net/pub/mgetty/
```

Originally there was a (Bourne) shell interface to **vgetty** only. The **Modem::Vgetty** module allows user to write the voice shell in Perl. The typical use is to write a script and point the **vgetty** to it (in **voice.conf** file). The script will be run when somebody calls in. Another use is running voice shell

from the ***vm*** **(8)** program, which can for example dial somewhere and say something.

QUICK START

```
#!/usr/bin/perl
use Modem::Vgetty;
my $v = new Modem::Vgetty;
$v->add_handler('BUSY_TONE', 'endh', sub { $v->stop; exit(0); });
local $SIG{ALRM} = sub { $v->stop; };
$v->enable_events;
$v->record('/tmp/hello.rmd');
alarm(20);
$v->waitfor('READY');
$v->shutdown;
```

The above example installs the simple ‘exit now’-style handler for the **BUSY_TONE** event (which is sent by **vgetty** when user hangs up) and then records the **hello.rmd** file. Put this text into a file and then point **vgetty** to it in the **voice.conf**. After you dial into your voice modem, you can record a 20−seconds of some message. Verify that **/tmp/hello.rmd** exists. Now delete the line contaning the word ‘‘record’’ and two subsequent lines and insert to the file the following line instead of them:

```
$v->play_and_wait('/tmp/hello.rmd');
```

Now call the voice modem and listen to the sounds you have just recorded.

METHODS

Begin and end of communication

The **Modem::Vgetty** object will initialize the communication pipes to the **vgetty** at the creation time − in the constructor. The closing of the communication is done via the **shutdown** method:

```
$v->shutdown;
```

The module will call this method itself from the destructor, if you do not call it explicitly.

Low-level communication

Users probably don’t want to use these methods directly. Use the higher-level functions instead.

receive
: This method returns a string received from the **vgetty**. It parses the string for the event types and runs appropriate event handlers. If event handler is run it waits for another string.

send($string)
: This method sends the string **`$string`** to the **vgetty** process.

expect($string1, `$string2`, ...)
: Receives a string from **vgetty** (using the **receive** method described above) and returns it iff it is equal to one of the strings in the argument list. When something different is received, this method returns **undef**.

waitfor($string)
: Waits until the string **`$sring`** is received from **vgetty** (using the **receive** method described above). =item chat($expect1, `$sent1`, `$expect2`, `$sent2`, ...)

  A chat-script with **vgetty**. Arguments are interpreted as the received-sent string pairs. A received string equals to the empty string means that no **receive** method will be called at that place. This can be used for constructing chat scripts beginning with the sent string instead of the received one.

Vgetty control methods

There are miscellaneous methods for controllig **vgetty** and querying its status.

getty
: Returns the name of the modem special file (e.g. **/dev/ttyC4**).

device($name)
: Sets the port of the voice modem input and output is done to. Possible values are qw(NO_DEVICE DIALUP_LINE EXTERNAL_MICROPHONE INTERNAL_SPEAKER LOCAL_HANDSET).

autostop($bool)
: With autostop on, the voicelib will automatically abort a play in progress and return READY. This is useful for faster reaction times for voice menus. Possible arguments are qw(ON OFF). **Note:**

The interface should probably be changed to accept the Perl boolean arguments (undef, something else). Returns defined value on success, undef on failure.

modem_type

vgetty currently has no way of telling voice shell the type of the current modem. This method is a proposed interface for determining this type. Currently returns **undef**. The appropriate low-level interface has to be implemented in **vgetty** first.

Voice commands

beep($freq, $len)

Sends a beep through the chosen device using given frequency (HZ) and length (in miliseconds). Returns a defined value on success or undef on failure. The state of the vgetty changes to ''BEEPING'' and **vgetty** returns ''READY'' after a beep is finshed. Example:

```
$v->beep(50,10);
# Possibly do something else
$v->waitfor('READY');
```

dial($number)

Modem tries to dial a given number. The **vgetty** changes its state to ''DIALING'' and returns ''READY'' after the dialing is finished.

play($filename)

The **vgetty** tries to play the given file as a raw modem data. See the ''Voice data'' section for details on creating the raw modem data file. It changes the state to ''PLAYING'' and returns ''READY'' after playing the whole file.

record($filename)

The **vgetty** records the voice it can hear on the line to the given file. It uses the raw modem data format (which can be re-played using the **play** subroutine). **vgetty** changes its state to ''RECORDING'' and you need to manually stop the recording using the **stop** method after some time (or, you can set **autostop** and wait for any event − silence, busy tone, etc).

wait($seconds)

The modem waits for a given number of seconds. Changes its state to ''WAITING'' and returns ''READY'' after the wait is finished. Example:

```
$v->wait(5);
$v->waitfor('READY');
```

stop

The **vgetty** stops anything it is currently doing and returns to the command state. You must use **stop** when you want to call another **beep**, **dial**, **play**, **record** or **wait** before the previous one is finished. The **vgetty** returns ''READY'' after the **stop** is called. So it is possible to interrupt a main routine waiting for ''READY'' from the event handler:

```
my $dtmf;
$v->add_handler('RECEIVED_DTMF', 'readnum',
        sub { my $self=shift; $self->stop; $dtmf = $_[2]; });
$v->enable_events;
$v->wait(10);
$v->waitfor('READY');
```

In the previous example the **waitfor** method can be finished either by the 10−second timeout expired, or by the 'READY' generated by the **stop** in the event handler. See also the **Events** section.

play_and_wait($file)

It is an abbreviation for the following:

```
$v->play($file);
$v->waitfor('READY');
```

It is repeated so much time in the voice applications so I have decided to make a special routine for it. I may add the similar routines for **dial**, **record**, **beep** and even **wait** in the future releases.

Event handler methods

add_handler($event, $handler_name, $handler)

Installs a call-back routine $handler for the event type $event. The call-back routine is called with three arguments. The first one is the Modem::Vgetty object itself, the second one is the event name and the third one is optional event argument. The **$handler_name** argument can be anything. It is used when you want to delete this handler for identificating it.

del_handler($event, $handler_name)

This method deletes the handler $handler_name for the $event event. The result of unregistering the handler from the event handler of the same event is unspecified. It may or may not be called.

enable_events

Tells the **vgetty** that the voice shell is willing to dispatch events. No events are sent by **vgetty** until this method is called.

disable_events

Tells the **vgetty** that the voice shell doesn't want to receive any events anymore.

The readnum method

readnum($message, $tmout, $repeat)

The applications often need to read the multi-digit number via the DTMF tones. This routine plays the **$message** to the voice object and then waits for the sequence of the DTMF keys finished by the '#' key. If no key is pressed for **$tmout** of seconds, it re-plays the message again. It returns failure if no key is pressed after the message is played **$repeat**−th time. It returns a string (a sequence of DTMF tones 0−9,A−D and '*') without the final '#'. When some DTMF tones are received and no terminating '#' or other tone is received for **$tmout** seconds, the routine returns the string it currently has without waiting for the final '#'. DTMF tones are accepted even at the time the **$message** is played. When the DTMF tone is received, the playing of the **$message** is (with some latency, of course) stopped.

NOTE: The interface of this routine can be changed in future releases, because I am not (yet) decided whether the current interface is the best one. See also the **EXAMPLES** section where the source code of this routine (and its co-routine) is discussed.

EVENTS

Introduction

Events are asynchronous messages sent by **vgetty** to the voice shell. The **Modem::Vgetty** module dispatches events itself in the **receive** method. User can register any number of handlers for each event. When an event arrives, all handlers for that event are called (in no specified order).

Event types

At this time, the **Modem::Vgetty** module recognizes the following event types (description is mostly re-typed from the **vgetty** documentation):

BONG_TONE

The modem detected a bong tone on the line.

BUSY_TONE

The modem detected busy tone on the line (when dialing to the busy number or when caller finished the call).

CALL_WAITING

Defined in IS−101 (I think it is when the line receives another call-in when some call is already in progress. −Yenya).

DIAL_TONE

The modem detected dial tone on the line.

DATA_CALLING_TONE

The modem detected data calling tone on the line.

DATA_OR_FAX_DETECTED

The modem detected data or fax calling tones on the line.

FAX_CALLING_TONE
The modem detected fax calling tone on the line.

HANDSET_ON_HOOK
Locally connected handset went on hook.

HANDSET_OFF_HOOK
Locally connected handset went off hook.

LOOP_BREAK
Defined in IS−101.

LOOP_POLARITY_CHANGE
Defined in IS−101.

NO_ANSWER
After dialing the modem didn't detect answer for the time give in dial_timeout in voice.conf.

NO_DIAL_TONE
The modem didn't detect dial tone (make sure your modem is connected properly to your telephone company's line, or check the ATX command if dial tone in your system differs from the standard).

NO_VOICE_ENERGY
It means that the modem detected voice energy at the beginning of the session, but after that there was a period of some time of silence (the actual time can be set using the **rec_silence_len** and **rec_silence_treshold** parameters in **voice.conf**).

RING_DETECTED
The modem detected an incoming ring.

RINGBACK_DETECTED
The modem detected a ringback condition on the line.

RECEIVE_DTMF
The modem detected a dtmf code. The actual code value (one of 0−9, *, #, A−D) is given to the event handler as the third argument.

SILENCE_DETECTED
The modem detected that there was no voice energy at the beginning of the session and after some time of silence (the actual time can be set using the **rec_silence_len** and **rec_silence_treshold** parameters in **voice.conf**).

SIT_TONE
Defined in IS−101.

TDD_DETECTED
Defined in IS−101.

VOICE_DETECTED
The modem detected a voice signal on the line. IS−101 does not define, how the modem makes this decision, so be careful.

UNKNOWN_EVENT
None of the above :)

VOICE DATA

Voice shell can send the voice data to the modem using the **play** method and record them using the **record** method. The ''.rmd'' extension (Raw Modem Data) is usually used for these files. The ''.rmd'' is not a single format − every modem has its own format (sampling frequency, data bit depth, etc). There is a **pvftools** package for converting the sound files (it is a set of filters similar to the **netpbm** for image files). The ***pvftormd* (1)** filter can be used to create the RMD files for all known types of modems.

EXAMPLES

Answering machine

A simple answering machine can look like this:

```
#!/usr/bin/perl
use Modem::Vgetty;
my $voicemaster = 'root@localhost';
my $tmout = 30;
my $finish = 0;
my $v = new Modem::Vgetty;
$v->add_handler('BUSY_TONE', 'finish',
        sub { $v->stop; $finish=1; });
$v->add_handler('SILENCE_DETECTED', 'finish',
        sub { $v->stop; $finish=1; });
local $SIG{ALRM} = sub { $v->stop; };
$v->enable_events;
$v->play_and_wait('/path/welcome.rmd');
$v->beep(100,10);
$v->waitfor('READY');
if ($finish == 0) {
        my $num = 0;
        $num++ while(-r "/path/$num.rmd");
        $v->record("/path/$num.rmd");
        alarm $tmout;
        $v->waitfor('READY');
}
system "echo 'Play with rmdtopvf /path/$num.rmd|pvftoau >/dev/audio'" .
         " | mail -s 'New voice message' $voicemaster";
exit 0;
```

See the **examples/answering_machine.pl** in the source distribution, which contains a more configurable version of the above text. It first sets the event handlers for the case of busy tone (the caller hangs up) or silence (the caller doesn't speak at all). The handler stops **vgetty** from anything it is currently doing and sets the `$finish` variable to 1. Then the reception of the events is enabled and the welcome message is played. Then the answering machine beeps and starts to record the message. Note that we need to check the `$finish` variable before we start recording to determine if user hanged up the phone. Now we find the first filename <number>.rmd such that this file does not exist and we start to record the message to this file. We record until user hangs up the phone or until the timeout occurs.

Readnum routine

An interesting application of the low-level routines is the **Voice::Modem::readnum** method. The calling sequence of this method has been discussed above. The source code for this routine and its co-routine will be discussed here, so that you can write your own variants of **readnum** (which in fact does not have too general interface). See also the source code of **Vgetty.pm** for the **readnum** source.

The **readnum** routine needs to have its own event handler for the **RECEIVED_DTMF** event and the way the handler can communicate with this routine. In our solution we use ‘‘static’’ variables:

```
my $_readnum_number = '';
my $_readnum_timeout = 10;
my $_readnum_in_timeout = 1;
```

The event handler will add the new character to the end of the **`$_readnum_number`** variable. The **`$_readnum_timeout`** is the number of seconds both **readnum** and the event handler should wait for the next keypress, and the **`$_readnum_in_timeout`** is a flag used by the event handler for notifying the main **readnum** routine that it forced the **vgetty** to emit the ‘READY’ message because of the final ‘#’ has been received.

```
sub _readnum_event {
        my $self = shift;
        my $input = shift; # Unused. Should be 'RECEIVED_DTMF'.
        my $dtmf = shift;

        if ($dtmf eq '#') { # Stop the reading now.
                $_readnum_in_timeout = 0;
                $self->stop;
```

```
                $self->{LOG}->print("_readnum_event(): Got #; stopping\n"
                return;
        }
        $_readnum_number .= $dtmf;
        $self->stop;
        $self->expect('READY');
        # Restart the wait again.
        $_readnum_in_timeout = 1;
        $self->wait($_readnum_timeout);
}
```

The event handler is installed for the 'RECEIVED_DTMF' event only, so it doesn't need to check for the **$input** value. The actual DTMF key is in the third parameter, **$dtmf**. Note that the handler will be called when **vgetty** is PLAYING or WAITING and the **readnum** routine will be waiting for the 'READY' message. This allows us to immediately interrupt waiting by the **$self**->stop> (which emits the 'READY' message). So when the '#' DTMF tone is received, we send a **stop** to **vgetty**. If something else is received, we **stop** the **vgetty** too but we enter a new wait using **$self**->wait>.

```
sub readnum {
        my $self = shift;
        my $message = shift;
        my $timeout = shift;
        my $times = shift;
        $_readnum_number = '';
        $_readnum_in_timeout = 1;
        $_readnum_timeout = $timeout if $timeout != 0;
        $times = 3 if $times == 0;

        # Install the handler.
        $self->add_handler('RECEIVED_DTMF', 'readnum', \&_readnum_event);
        while($_readnum_in_timeout != 0 && $_readnum_number eq ''
                && $times-- > 0) {
                $self->play_and_wait($message);
                last if $_readnum_in_timeout == 0;
                while ($_readnum_in_timeout != 0) {
                        $self->wait($_readnum_timeout);
                        $self->expect('READY');
                }
        }
        return undef if $times < 0;
        $self->del_handler('RECEIVED_DTMF', 'readnum');
        $self->stop;
        $self->expect('READY');
        $_readnum_number;
}
```

The **readnum** routine just sets up the event handler, then plays the **$message** and waits for the input (possibly several times). The main work is done in the event handler. At the end the handler is unregistered and the final value is returned.

Callme script

In the **examples** subdirectory of the source distribution there is a **callme.pl** script. This dials the given number and plays the given message. Use the following command to run it:

```
vm shell -S /usr/bin/perl callme.pl <number> <message>.rmd
```

BUGS

There may be some, but it will more likely be in the **vgetty** itself. On the other hand, there can be typos in this manual (English is not my native language) or some parts of the interface that should be redesigned. Feel free to mail any comments on this module to me.

TODO

Modem type recognition

The **vgetty** should be able to tell the voice shell the name of the current modem type.

The *_wait()* routines

I need to implement the routines similar to **play_and_wait** for other **vgetty** states as well.

Debugging information

The module has currently some support for writing a debug logs (use the `$Modem::Vgetty::testing` = 1 and watch the /var/log/voicelog file). This needs to be re-done using (I think) Sys::Syslog. I need to implement some kind of log-levels, etc.

Mgetty/Vgetty 1.1.17

Need to figure out what is new in 1.1.17 (I use 1.1.14 now). I think new **vgetty** can play more than one file in the single 'PLAY' command, it (I think) have some support for sending voice data from/to the voice shell via the pipe, etc.

AUTHOR

The **Modem::Vgetty** package was written by Jan ''Yenya'' Kasprzak <kas AT fi DOT muni DOT cz>. Feel free to mail me any suggestions etc. on this module. Module itself is available from CPAN, but be sure to check the following address, where the development versions can be found:

`http://www.fi.muni.cz/~kas/vgetty/`

COPYRIGHT