Trap Errors, Exit Codes and Line Numbers within a Bash script (and some output redirection, too)

A discussion today was about error handling in shell scripts, Bash in particular.
Well, we all know about the usual knitpicks about error handling and the possible consequences in not doing so properly ;-)

The most simplistic approach in error handling are of course control structures to check the return value:

```
some arbitrary command
if [ "$?" = "0" ]; then
  do something
else
  do something else
fi
```

Or even more simplistic:

```
some arbitrary command && { do something; } || { do something else; }
```

There are more ways to do this. But they all have in common, that it is hard to trap and trace unknown errors, especially if the script runs unattended.
If you want to do some logging and tracing, then you would need to implement a routine which you would need to add to each and every block, to ensure you don't miss some particular important information.
Even if it is a simple function for error reporting, let's call it error_reporter, you would end up with something like this:

```
some arbitrary command
if [ "$?" = "0" ]; then
  do something
  error_reporter args
else
  do something else
  error_reporter args
fi
```

But sometimes it's better to have an error handler, which is able to catch errors and do some special actions, while still allowing your script to continue within the normal flow. Even better if that particular error handler also catches and notifies upon yet-unknown and never discovered errors.

Let's assume a script, which is trying to delete a directory. For the sake of this example, the directory MUST NOT exist, so the script effectively fails upon execution.
We handle the error simply by checking on the non-zero exit code.

Please note: The following code serves as an example of bad script programming. Never (as in NEVER) try to delete something without checking for it's existence first! You'll find a proper code sample at the end of this article ;-)

```
#!/bin/bash

# try to delete non-existing directory
#
rmdir /ksdjhfskdfkshd
if [ "$?" = "0" ]; then
	echo "ok: directory deleted."
else
	echo "failed: directory not deleted."
fi
```

This script will try to delete the non-existing directory. As 'rm' will not find it, it will return a non-zero exit status, leading to this program output:

```
$ bash test.sh
rm: /ksdjhfskdfkshd: No such file or directory
failed: directory not deleted.
```

So far, so good. The error was caught and nothing really bad happened, we even got kinda useful error message from the script itself.
Now, let's think about scripts, which usually run unattended, maybe invoked through cron.
Any halfway serious admininstrator would at least try to capture the error output from cron by redirecto STERR and STDOUT to a logfile.

```
*/5 * * * * root /var/scripts/somescript > /var/log/somescript.log 2>&1
```

But, to be serious, how often would you really check on these logfiles?

Of course, you can have cron configured to send you the script output after every run.
If you use an output redirection as shown above, you could even write another script, which sends these logfiles to you.
This is good enough, you'll end up with dozens if not hundreds mostly useless emails, which you'll most likely never ever get to read.

Wouldn't it be great to have the script report any runtime errors to you by email, directly into a database or via SNMP traps, but only in the event of some real importance to look at?
In this case, you'll end up with error reports driven by occurence. So you KNOW that it's important and some action needs to be performed.

Luckily enough Bash provides us with a very simple TRAP interface, which allows to run additional, event-driven code, upon occurence.
We can trap nearly every thing, from EXIT to CTRL-C, over SIGNALS up to ERROR status (you'll find more about this in the Bash info page).

To trap an ERROR status, we need two things: A trap handler and a trap command.
The first is some code, which does any particular action, for example assemble error information and send it by email, while the trap command itself specifies, under what condition it needs to be invoked.

A trap handler could look something like this:

```
function my_trap_handler()
{
	MYSELF="$0"		# equals to my script name
	LASTLINE="$1"	 # argument 1: last line of error occurence
	LASTERR="$2"		# argument 2: error code of last command
	echo "${MYSELF}: line ${LASTLINE}: exit status of last command: ${LASTERR}"

	# do additional processing: send email or SNMP trap, write result to database, etc.
}
```

To have the trap handler executed, you need to add a "trap" statement to the script. As we want the trap_handler to be invoked only upon a command failure, we consider only the ERR trap, which catches non-zero exit codes only.

```
trap my_trap_handler ${LINENO} ${$?}' ERR
```

Let's have a look at the completed script now.
To demonstrate how accurately the trap handler works, I added some further commands. These commands have been designed so that they WILL fail for the sake of documentary purposes.

```
#!/bin/bash

# trap handler: print location of last error and process it further
#
function my_trap_handler()
{
	MYSELF="$0"		# equals to my script name
	LASTLINE="$1"	 # argument 1: last line of error occurence
	LASTERR="$2"		# argument 2: error code of last command
	echo "${MYSELF}: line ${LASTLINE}: exit status of last command: ${LASTERR}"

	# do additional processing: send email or SNMP trap, write result to database, etc.
}

# trap commands with non-zero exit code
#
trap 'my_trap_handler ${LINENO} $?' ERR

# let's do some rubbish
```

```
#
rm /ksdjhfskdfkshd
if [ "$?" = "0" ]; then
	echo "ok: directory deleted."
else
	echo "failed: directory not deleted."
fi

# next is line 30
format c:		# ;-)

# next is line 33
ls /fdkjhfskdhfks
```

This sample code will result in the following output at runtime. I added the line numbers in front on my own for better illustration.

```
$ bash test.sh
1:  rm: /ksdjhfskdfkshd: No such file or directory
2:  test3.sh: line 22: exit status of last command: 1
3:  failed: directory not deleted.
4:  test3.sh: line 30: format: command not found
5:  test3.sh: line 30: exit status of last command: 127
6:  ls: /fdkjhfskdhfks: No such file or directory
7:  test3.sh: line 33: exit status of last command: 1
```

Now, the first line of output shows the error thrown by the 'rm' command.
The second line shows the output from the trap handler, stating exactly WHERE the error occured (test.sh / line 22 / last exit status).
The third line shows the output of the local error handling routine.
Line four gives us the "format" command not found, while line five is the message from the trap handler.
Line six and seven are the "ls" for the non-existing directory and the trap handler message corresponding to it.

Looking into this outline it quickly becomes clear, that a trap handler can help us a lot in event-driven debugging with just adding a few lines to existing scripts without tampering with existing error handling code.
This script design even permits you to track and trace errors on single-line commands, which you almost never believed to fail.
Even if they fail one day due to unlikely events, you can at least point out WHERE it failed, which makes debugging a lot faster and easier.

And now, finally, the completed script code, as any decent script writer should be propably doing it using some pre-action checking as well.

```
#!/bin/bash

# trap handler: print location of last error and process it further
#
function my_trap_handler()
{
```

```
	MYSELF="$0"		# equals to my script name
	LASTLINE="$1"		# argument 1: last line of error occurence
	LASTERR="$2"		# argument 2: error code of last command
	echo "${MYSELF}: line ${LASTLINE}: exit status of last command: ${LASTERR}"

	# do additional processing: send email or SNMP trap, write result to database, etc.
}

# trap commands with non-zero exit code
#
trap 'my_trap_handler ${LINENO} $?' ERR


# we need to process the DIRECTORY at /ksdjhfskdfkshd
#
my_directory=/ksdjhfskdfkshd


# test if the directory exists
#
echo -n "check if directory exists: '${my_directory}': "

if [ -d "${my_directory}" ]; then
	echo "ok."

	# try to delete the file
	#
	rmdir "${my_directory}"

	# try to handle delete failure (exit != 0)
	#
	if [ "$?" = "0" ]; then
		echo "ok: directory deleted."
	else
		echo "failed: directory not deleted."
	fi
else
	echo "failure, directory does not exist or is not a directory."
fi

# we're done
#
exit
```

This would result in the following output. The trap is completely circumvented by testing the directory for it's existence before deleting it.

```
$ bash test.sh
```

check if file exists: '/ksdjhfskdfkshd': failure, directory does not exist or is not a directory.

One more thing to consider is what happens, if a trap is caught within a control structure block (IF, WHILE, etc).

To illustrate this, I create the directory I wanted to delete, so the script will dive into the IF-THEN-ELSE-FI block.

$ mkdir /ksdjhfskdfkshd

I then changed the "rmdir" command within the script to "rmdiir" (misspelled), so it WILL fail upon execution.
Please note the script output we get this time:

$ bash test.sh
check if file exists: '/ksdjhfskdfkshd': ok.
test.sh: line 34: rmdiir: command not found
test.sh: line 45: exit status of last command: 127
failed: directory not deleted.

While Bash itself states the misspelled command being on line 34, the trap catches the error on line 45.

Now, why is this?

The reason for this is very simple: Any control structure is regarded as some sort of multi-line command within the script.
So when the trap catches the erronous command on line 34, it sees it's origin on line 45 because the "IF-THEN-ELSE-FI" clause ends on line 45.

The same happens if you use any other control structure.

The trap in this case is only capable of outlining the "general direction" to where the error happened, but it cannot pin-point to it.
So it's still recommended to also capture the script output, either by redirecting the script output manually from the shell, or more elegantly by adding some lines to the script, which always redirect the output (STDERR and STDOUT) to a logfile automatically.

In the end, the trap handler could be setup to send and email using both information from the intercepted trap and the logfile.

So the final script may look like this:

```
#!/bin/bash

# initialize upon startup
#
my_temp_dir=`mktemp -d /tmp/test.XXXXXX`  # we want a unique temp dir
my_log_file=${my_temp_dir}/output.log
my_out_pipe=${my_temp_dir}/output.pipe
```

```
# initialize output redirection
#
mkfifo ${my_out_pipe}                          # we need a FIFO for the output pipe
exec 3>&1                                      # assign a new file descriptor 3 to STDOUT
tee ${my_log_file} &3 &                        # background redirect file descriptor 3 throught the FIFO
tee_pid=$!                                     # store the PID for 'tee'
exec 2>&1 > ${my_out_pipe}                     # redirect STDERR and STDOUT to the output pipe


# do some final cleanup upon exit
#
function my_exit_handler()
{
    exec 1>&3 3>&-                             # restore file descriptors
    wait ${tee_pid}                            # wait for 'tee' to exit

    # remove temp dir upon exit
    #
    #[ -d ${my_temp_dir} ] && rm -rf ${my_temp_dir}
}

# trap handler: print location of last error and process it further
#
function my_trap_handler()
{
      MYSELF="$0"              # equals to my script name
      LASTLINE="$1"            # argument 1: last line of error occurence
      LASTERR="$2"             # argument 2: error code of last command
      echo "script error encountered at `date` in ${MYSELF}: line ${LASTLINE}: exit status of last command: ${LASTERR}"

      # do additional processing: send email or SNMP trap, write result to database, etc.
 #
 # let's assume we send an email message with
 # subject: "script error encountered at `date` in ${MYSELF}: line ${LASTLINE}: exit status of last command: ${LASTERR}"
 # with additional contents of ${my_log_file} as email body
}


# trap commands with non-zero exit code
# trap script EXIT
#
trap 'my_trap_handler ${LINENO} $?' ERR
trap 'my_exit_handler' EXIT


# we need to process the DIRECTORY at /ksdjhfskdfkshd
#
my_directory=/ksdjhfskdfkshd
```

```
# test if the file exists
#
echo -n "check if file exists: '${my_directory}': "

if [ -d "${my_directory}" ]; then
 echo "ok."

 # try to delete the file
 #
 rmdir "${my_directory}"

 # try to handle delete failure (exit != 0)
 #
 if [ "$?" = "0" ]; then
        echo "ok: directory deleted."
 else
        echo "failed: directory not deleted."
 fi
else
 echo "failure, directory does not exist or is not a directory."
fi


# we're done
#
exit
```

To conclude this: Adding some further logic to a script using output redirection and traps adds some great debugging aid, especially when it comes to unknown and yet-undiscovered errors.
The implementation requires just a few additional lines to work with any script and will save countless hours worth of debugging.
Also, the sample trap handler presented herein can be extended to do virtually anything, from adding additional information like the environment, to submitting errors into a MySQL database, sending SNMP traps, or whatever you could imagine.