Debugging in QDL

The basic process of debugging is running your program in a controlled environment that allows you to find errors. In the case of QDL, there is the debugger command, which allows for selective writing of information to the error stream and the state indicator system, which permits you to halt execution of a program and explore the state of it before resuming.

The debugger command.

The function reference has a good section on it. An example is included in the standard distribution in $QDL_HOME/examples/debugger-ex.qdl which allows you to set the debugger level in the workspace and see how it affects the output. The debugger lets you put statements at different levels throughout your code so you can ratchet up or down the level or warnings as needed.

E.g.

```
   debugger(1); // set to lowest level, show everything
   script_run(info('user').'qdl_home'+'/examples/debugger-ex.qdl');
```

Which will print out the following to the console (in the Swing gui, the output window):

```
current debugger settings:
   delimiter :  |
     enabled : true
       level : 1
       title : WorkspaceCommands
       ts_on : true
debugger example done!
```

And (if in Swing GUI) the following to the window where you started QDL:

```
2025-05-14T11:33:44.073Z | debugger-ex.qdl | trace: this is a debugger message at level 1
2025-05-14T11:33:46.149Z | debugger-ex.qdl | info: this is a debugger message at level 2
2025-05-14T11:33:46.743Z | debugger-ex.qdl | warn: this is a debugger message at level 3
2025-05-14T11:33:47.252Z | debugger-ex.qdl | severe: this is a debugger message at level 4
2025-05-14T11:33:47.738Z | debugger-ex.qdl | error: this is a debugger message at level 5
```

Or this will end up in the console window if you are running QDL in ascii/text or ANSI mode. Note that the output is formatted with the timestamp and process (or script name) that called the debugger. See the manual for all the particulars. But running this with a different debugger level, say 4, so only severe or worse output is show yields

```
   debugger(4);
   script_run(info('user').'qdl_home'+'/examples/debugger-ex.qdl');
```

and wherever system error is showing you will see the following new entries:

```
2025-05-14T12:12:05.848Z | debugger-ex.qdl | severe: this is a debugger message at level 4
2025-05-14T12:12:05.848Z | debugger-ex.qdl | error: this is a debugger message at level 5
```

Using the SI (state indicator) and halt functions.

The halt function will send an interrupt to the process and suspend it according to the current configuration of the SI.

Sending interrupts

To *send an interrupt* means to call the **`halt`** command which will tell the processor to suspend execution as per configuration of the SI. You may then inspect the state (variables, functions et.) and view information about the process by listing it in the SI. It may be resumed to the next viable interrupt. The **`halt`** command allows for a message and a label. Note that the basic unit of execution in QDL is the statement. Since **`halt`** is a function, it may be called any place but the effect is to terminate the current statement. The number of the statement (which may span many lines) is show as is the line number. It is for this reason that best practices dictate having **`halt`** by itself on a single line. That said as a function, it may be executed conditionally, so this is perfectly fine

```
   size(args()) != 2 ⇒ halt('No username or password supplied');
```

indicating that during debugging, if the correct arguments are not passed, an interrupt is always sent.

The use of labels

`halt` may have a first argument that is called a label. In fact, this is any QDL object that can be matched exactly with ==. This includes strings, null, integers, booleans and sets. When running the code, you may tell the system which interrupts to include (run only these) or to exclude (skip these) based on matching the label. The system also supports label matching with regular expressions, as well as exactly, but you cannot mix and match lists of labels and regexes.

What not to do

It would be possible – since QDL is a notation – to write some monstrosity like

```
   a. ≔[;3√halt('my nasty interrupt')];
```

but this would mostly be confusing. Another bad idea would be to use a label like a decimal which has potential issues with matching it, e.g.

```
halt(cos(pi()/6), 'cosinus interruptus');
```

and checking against the value of √3/2, but that is probably not going to work due to internal rounding of decimals. If you really need to check decimals, the right way to do it is with a comparison tolerance, so

```
   ct := 1.0E-10; comparison tolerance
   halt(cos(pi()/6) - √3/2 < ct, 'cosinus interruptus');
```

and testing for the boolean **`true`**.

Viable interrupts

If you specify a list of interrupt labels, the if a match is termed *viable* and action is taken. Normally when you start running code all interrupts are viable. You may turn them off with the **-go** switch:

```
   ) 0 -go
```

tells the workspace to execute the buffer numbered 0 (you could also use its handle) and to ignore all interrupts.

Resuming from an interrupt

To resume execution, you issue **)si resume [pid]** or the shorthand **)) [pid]**. Note that the execution begins with *the next statement*, as the statement where the **halt** was sent was aborted at the instant it was called.

Where interrupts are disabled.

If QDL is running in server mode, then all interrupts are disabled, lest since there is no way to have the programmer interact with the runtime environment.

Using the SI facility on workspace buffers.

The SI (State Indicator) facility is a powerful interactive debugging tool. You set your interrupts in your code and then when you run it, it will stop at the first viable interrupt, list the process id and wait for you to take some action.

A sample SI session.

Included in the standard distribution is an example, **$QDL_HOME/examples/halt-test.qdl**, which has various interrupts so you can see how it works. There are several other examples in the comments, so do be sure to have a look at the source code for more. First ,we create a buffer

```
   h := info('user').'qdl_home' + '/examples/halt-test.qdl'
 )b create h_test >h
1| |h_test: /home/ncsa/apps/qdl/examples/halt-test.qdl
```

This sets the path in the workspace and then uses the variable to create the buffer. To run the test with no interrupts, use the **-go** flag

```
   ) h_test -go
starting halt test
done!
```

1. Running the example with all interrupts enabled.

This section shows how to run the example mostly to show how the SI should be used. So, we start the test, this time allowing every interrupt (and there are a lot of them!)

```
   ) h_test
```

```
starting halt test
Stop #1.
pid: 101
```

What this means is that the first line of output is just printed by the test at startup. The second line is the message that the first interrupt sends. The pid is the number of the process. Let's look at the state indicator:

```
pid   | active | stmt  | line  | time                          | size   | message
0     |   *    |       |       | 2025-05-14T19:24:48.555Z  | 0      | system
101   |  ---   | 4     | 34    | 2025-05-14T19:25:13.985Z  | 0      | Stop #1.
```

The first line is always there – it is the system pid and its information. You really can't do anything with the system process except make it active again (and the asterisk in column 2 makes it as the active process).

The last line is the useful one. It tells us that the interrupt with in the 4th executable statement, which happened at line 34. It gives the timestamp and the size is a rough estimate at used system resources. Since the process didn't do much more than send an interrupt as soon as it started, it stands to reason there is not a lot of state. Let's make this process the default one:

```
   )si set 101
pid set to 101
```

So now to resume, you don't have to give the pid, just issue the resume command which can be either

```
   )si resume
```

or the much easier to type

```
   ))
arg stop #1.: at line 35
```

This is the next message followed by the line number and again the SI shows this

```
   )si
pid   | active | stmt  | line  | time                          | size   | message
0     |  ---   |       |       | 2025-05-14T19:24:48.555Z  | 0      | system
101   |   *    | 5     | 35    | 2025-05-14T19:31:34.485Z  | 0      | arg stop #1.
```

Now, let's run it again

```
   ))
Stop #2. a. is set: at line 37
```

Now for the utility of all of this. Let's look at the variables in the workspace:

```
   )vars
A  B  C  a.  h
```

We defined h initially, but the others are defined by the test as it runs and we can explore what the current state is. Let's show how to skip over all the other interrupts to send the one with label 'label 5' since that happens right after a function is defined. To do this we need to give the set (or list or

stem) of interrupts to execute, skipping all others. This is done with the **-ii** (or the much more verbose **-include_interrupts**) flag. Remember that the workspace manages QDL, but does not process it directly. The best you can do it create a variable and pass that in using the > to mark it.

```
   includes :={'label 5'}
   )) -ii >includes
Stop #5. Check your functions.: at line 50
```

This skips every halt call until it hits the one with the given label. At this point, we have a function defined:

```
   )funcs
q([2])
1 total functions
```

To run without any interrupts, you can still use the -go flag

```
   )) -go
done!
exit pid 101
```

Using includes and excludes

Now that we have seen the SI in action, we should talk about included and excluded interrupts. These are lists that are exclusive, so

- included <==> only those give are viable
- excluded <==> those given are skipped, all other are viable.

You may give these at any time including

- When running the buffer initially
- When resuming the buffer
- (once a pid is assigned) explicitly setting them with

```
   )si set pid {-ii regex | set | stem} {-xi regex | set | stem}
```

-ii = -included_interrupts, **-xi = -excluded_interrupts**.

The workspace is not a QDL parser (Gödel would approve), so you cannot include general QDL as an argument. You can set a variable and point to that, as indicated above, with >. So all of these work:

```
   i_set := {'a','b','c'};
 i_list. := ['a','b','c'];
 i_stem. := {'a_key':'a', 'b_key':'b', 'c_key':'c'}
```

In the case of a stem, the values are taken and the keys are ignored.

If you supply and argument without >. it is assumed to be a regular expression and matching will be done against that. So another way in the previous example to hop to label 5 would have been

```
)) -ii 'label 5'
```

(which would match exactly although you can get quite clever with regexes).

Note that if both -ii and -xi are given, the net effect is to do all.

To clear a list, you can set it on the command line to ∅ (empty set), {} or [], e.g.

```
   )) -ii >my_i_set
```

would run until it hits an element in my_i_set. To resume with -xi now, just set the includes to the empty set

```
   )) -ii {} -xi >my_x_set
```

Final note: includes and excludes apply to halt functions that have a label. No label is interpreted as an unconditional interrupt I, *i.e.*, it must be done no matter what.

Conditional execution of halt

You may also set halt to run as part of a conditional in the script, without recourse to using the SI. Just treat it like any other expression

```
   size(args() <3) ? halt('missing arguments!');
```

means that if the number of arguments to the script is less than 3, an interrupt will be sent.

Example running with &

These allow for keeping the state of the code separate from the main workspace. In the case of &, local state will be created, and the state of the workspace is inherited. When done, it is lost (although like named variables might be overwritten). See the next section for an example using **!**.

Taking our example above in a new workspace.

```
      ) h_test &
starting halt test
Stop #1. at line 34
pid: 101
   )si set 101
pid set to 101
   ))
arg stop #1.: at line 35
   ))
Stop #2. a. is set: at line 37
   )vars
A  B  C  a.  h
```

If we decide to abandon this and go back to the main workspace, we set the pid to 0

```
   )si set 0
system process restored
   )vars
h
```

which is just what we defined before we started. This is very convenient if the script is misbehaving and we need a bunch of bad state to just go away.

Example running with !

Here, the state of the script is completely independent of the workspace. Starting in a new workspaces

```
  h := info('user').'qdl_home' + '/examples/halt-test.qdl'halt_test.qdl'
  )b create h_test >h
  ) h_test !
starting halt test
Stop #1. at line 34
pid: 101
```

Running this a few more times to get state:

```
   )si set 101
pid set to 101
   ))
arg stop #1.: at line 35
   ))
Stop #2. a. is set: at line 37
   )vars
A  B  C  a.
```

Note that h, which we defined in the main workspace, is absent. If we pop back over to the main workspace:

```
   )si set 0
system process restored
   )vars
h
```

We can toggle back and forth between these processes as much as we like if we need to compare them.

Debugging scripts

Running with script_load or script_run

Generally you can just invoke the script with **`script_load`** or **`script_run`** and if an interrupt is sent, execution passes to the SI facility. One useful little trick for keeping your state wholly separate when using script_load is to encase the call in a block statement:

```
block[script_load(. . .);];
```

which will make a new local environment and run the script in it. This is equivalent to the buffer run command with a **&** appended.

Running from the command line

If you have set your QDL program to be runnable from the command line, then if you have interrupts, they will be caught and you will be dumped out in the workspace. A few things to note:

1. The configuration of the workspace will be whatever you set and normally this does not have echo, pretty_print or other niceties automatically configured. When in doubt, issue

```
)ws get
```
to see what the current state is.

2. When you finish running the script, you will still be in the workspace until you exit it.
3. It is not possible to specify initial include, excludes when running a script.

The halt test will run from the command line, so give it a try.

Sample debugging session with halt_test.qdl

Start by creating a buffer for the script.

```
   )b create h_test /home/jeff/apps/qdl/examples/halt_test.qdl
0| |h_test: /home/jeff/apps/qdl/examples/halt_test.qdl
```

This sample has multiple halt statement in it. Invoking it (the 0 argument is the buffer number):

```
  ) 0
starting halt test
103
```

The script is now running with the same state as the workspace, so any variables etc. the script creates will be in your workspace after the process ends. This shows the startup message and the associated pid. This process is now suspended. After running it a couple of times to get more interesting state messages, we can list the current state indicator. These are all the currently running processes in QDL.

```
  )si list
pid   | active | stmt  | line  | time                          | size | message
0     |  ---   |       |       | 2025-05-11T12:44:46.182Z  | 0    | system
103   |   *    | 7     | 21    | 2025-05-11T12:53:43.028Z  | 0    | Stop #4. Check your variables
```

Meaning there are exactly two processes. (The process id or ***pid*** identifies these.) The main system process is always pid = 0. The other process, with pid = 103, is currently suspended at line 21 and is the active process. In this example, we are already running another copy of halt_test, complete with its own state.

Starting the buffer with **&** added will completely clone the current workspace (allowing all current state to be used by the script), load the buffer in to that and run it until the first **`halt`** is invoked. This means in both cases once the process ends, the workspace is returned to its original state. Compare this with using a **!** rather than the **&**: That would start a new instance of QDL with *nothing* from the current state. Useful for other operations. Both **!** and **&** cannot be given at the same time and you will get an error if you try.

```
  ) 0 &
starting halt test
104
```

What this does is start evaluating the script (in this case it just prints a message) and once a halt() is encountered, the pid is returned. Since the script is now suspended, you can just view its state in the state indicator:

```
  )si list
pid | active | stmt  | line  | time                        | size  | message
0   |  ---   |       |  --   | 2025-05-11T12:51:40.033Z  | 0     | system
103 |   *    | 7     | 21    | 2025-05-11T12:53:43.028Z  | 2965  | Stop #4. Check your variables
104 |  ---   | 1     | 15    | 2025-05-11T12:57:19.341Z  | 2965  | Stop #1.
```

What this shows is the new entry with pid 104. It lists what the active state is (0, marked with a *) and what statement number the halt() was encountered on, here stmt 1 in the file.

The time this was started is shown. The size (in bytes) of the state (includes all variables and what not is shown). Finally, when **`halt(msg)`** is invoked, the message is displayed in the last column. It may also be printed to the console if you use the **`message`** command. Let's set the default for the workspace to pid = 104:

```
  )si set 104
pid set to 104
```

Now we can simply use the *resume* shorthand (a double left parentheses) without argument to run to the next halt command:

```
  ))
```

Since there was no output from the script, but a variable was set, nothing displays. Now let's show the state indicator again:

```
  )si list
pid | active | stmt  | line  | time                        | size  | message
0   |  ---   |       |  --   | 2025-05-11T12:51:40.033Z  | 0     | system
103 |   *    | 7     | 21    | 2025-05-11T12:53:43.028Z  | 2965  | Stop #4. Check your variables
104 |  ---   | 1     | 15    | 2025-05-11T12:57:53.002Z  | 3001  | Stop #2. a. is set
```

The message in the halt command is that a variable, *a.* was set. Let's check the variables in the workspace:

```
  )vars
a.  java#eg.
  a.
[2,4,6]
```

If we want we can swap back to the original state of the workspace by setting the pid to 0:

```
)si set 0
```