Calypsi debugger guide for the Nut

Release 5.18

Håkan Thörngren

Jul 15, 2026

Contents

1 Introduction 1
1.1 Command line debugger . . . 1

2 Getting started 3
2.1 User interface . . . 4
2.2 Starting . . . 4

3 Console command line 7
3.1 Command structure . . . 7
3.2 A simple session . . . 8

4 Visual Studio Code 13
4.1 Installation . . . 14
4.2 Project setup . . . 14

5 Emacs 17
5.1 Starting . . . 17
5.2 Stopping . . . 17
5.3 Many windows mode . . . 18

6 Console command reference 19
6.1 Essential commands . . . 19
6.2 Execution control . . . 20
6.3 Breakpoint commands . . . 20
6.4 Parameters . . . 21
6.5 Data manipulation . . . 22
6.6 File commands . . . 22
6.7 Directory context . . . 22
6.8 Thread commands . . . 23
6.9 Lua commands . . . 23
6.10 HP-41 modules . . . 23
6.11 Miscellaneous . . . 23

Index 25

CHAPTER 1

Introduction

Welcome to the Calypsi debugger, a powerful tool which allows you to see what happens inside programs while it executes. Calypsi debugger provides both interactive as well as script based control.

The design of the debugger is based on ideas coming from command line debuggers like GDB (GNU project debugger) and LLDB (LLVM project debugger). If you are familiar with these you will find lots of similarities.

1.1 Command line debugger

A command line debugger provides powerful commands that are easy to remember and understand. You can use descriptive commands like `step`, `break` and `continue` rather than a set of bindings to function keys and other more or less cryptic key bindings. Thanks to short forms and repeat the last command when entering no command, you can do the many common actions with a single or two key strokes.

The Calypsi debugger can also be used with many debugger front ends, such as Visual Studio Code, Eclipse and Emacs. This is possible thanks to the MI (Machine Interface) protocol designed by the GNU project which allows for very precise control.

The Calypsi debugger also provides powerful scripting abilities via the Lua language. You can use Lua to configure debugging sessions, create your own commands and even make your own debugger plug-ins. This is made possible by that the complete debugger functional API is provided to Lua. The debugger Lua API even goes further and provides additional concepts, such as making it possible to define new console commands and subscribe to internal debugger events.

In other words, with the Calypsi debugger you get the best from both the command line world and integrated graphical development environments.

CHAPTER 2

Getting started

The Calypsi debugger is a command line tool named dbnut. Running it from the command line with the "–help" option will give a sign-on message and display accepted options:

```
Calypsi debugger for Hewlett-Packard Nut version 5.18

Usage: dbnut [--version] [-i|--interpreter INTERPRETER] [--nh] [--nx]
             [-e|--eval-command COMMAND] [-t|--tty TTY] [--batch]
             [--logging-enable] [--logging-to-file PATH] [--logging-overwrite]
             [--silent] [-p|--port PORT-NUMBER] [--disable-web-server]
             [--load-module FILE] [--save-state FILE] [--extended-ram] [FILE]
  use 'dbnut --help' for detailed help

Available options:
  --version               Display version number
  -i,--interpreter INTERPRETER
                          Select command interpreter, one of 'Console', 'MI' or
                          'MI2' (defaults to 'Console')
  --nh                    Do not read '~/.dbnut'
  --nx                    Do not read any '.dbnut' file in any directory
  -e,--eval-command COMMAND
                          Execute a single command (can be specified many
                          times)
  -t,--tty TTY            Use TTY for input/output by the program being
                          debugged
  --batch                 Exit after processing options
  --logging-enable        Enable logging from start
  --logging-to-file PATH  Enable logging from start to specified file
  --logging-overwrite     Enable logging from start and overwrite the file
  --silent                Generate less messages
  -p,--port PORT-NUMBER   Port number, defaults to '8080'
  --disable-web-server    Do not start any web server
  --load-module FILE      Load module file (.mod extension)
  --save-state FILE       Where to save calculator state (defaults to
                          '/home/hth/.dbnut/hp41.state')
```

(continues on next page)

(continued from previous page)

```
  --extended-ram          Provide RAM in the 400-FFF range (NEWT style RAM
                          memory)
  -h,--help               Show this help text
```

2.1 User interface

The debugger simulates an HP-41 calculator internally, but in order to control the calculator we need to be able to push button and watch the display. Rather than opening a traditional graphical user interface, the debugger relies on web technologies and starts a web server.

2.1.1 HP-41 web application

The web application, or HP-41 front end is included in the installation under `hp41-ui/hp41.html`. Simply open it in any web browser and it will connect to the debugger.

The source code for the user interface project can also be found at dbnut-ui[1].

Note: You should only have one HP-41 web application front end open at any time. Having multiple ones will cause them to fight over which is the user interface as the debugger expects to talk to a single front end. If this happens a notification will be written to the debugger console output. Simply close one of the front ends if this occurs.

2.1.2 Protocol

The web server acts on simple JSON-RPC commands passed over a web socket. Typically key presses are passed from the web application and notifications about display updates are passed back. The user interface is in other words quite dumb. All processing takes place in the debugger.

2.2 Starting

The Calypsi debugger simulates the HP-41 Nut processor with an attached HP-41 memory system. In order to do anything meaningful you need to provide a mainframe (operating system) image. Such image contains the internal ROMs of the HP-41 and is not included with this product.

The minimal command line to start the Calypsi debugger in a meaningful way is:

```
$ dbnut --load-module 41cv.mod
```

Where `41cv.mod` is the mainframe as a module file. The Calypsi debugger can also be started as an HP-41CX using an appropriate module file.

Note: Extended memory and the clock chip is currently supported, but the beeper is not.

You can plug in other modules as well, simply specify each one using the `--load-module` option:

[1] https://github.com/hth313/dbnut-ui

```
$ dbnut --load-module 41cx.mod --load-module ppc.mod --load-module zenrom.mod
```

To debug your own module at MCODE level you should build it using the `-g` option to produce debugging information. The linker will normally produce an ELF/DWARF executable image with the `.elf` extension that can be loaded by the Calypsi debugger:

```
$ dbnut --load-module 41cx.mod myimage.elf
```

2.2.1 Command line

Once the Calypsi debugger is started it responds with:

```
$ dbnut --load-module 41cx.mod myimage
Calypsi debugger for Nut
listening for user interface connections on port 8080
(dbnut)
```

The debugger announces its presence and opens a web server for the user interface. The `(dbnut)` text is the command prompt indicating that it is ready for commands.

At this point the debugger is ready to start a debug session with the modules specified by `--load-module` options loaded. However, the simulated process does not yet exist.

To start the program and create a process for it, simply run it with the `run` (or just `r`) command:

```
(dbnut) r
running
```

The debug process is created and execution starts, however it does not get so far as the calculator starts in sleep. To wake it up, press the `ON` button on the calculator user interface.

Note that after the `running` message, you will not get the prompt back. This is because the calculator is active. To take control again in the debugger, press `Ctrl-c` in debugger console window.

```
(dbnut) r
running
^C
program received signal SIGINT, interrupt
0x0000
(dbnut)
```

The running program received an interrupt signal at address `0x0000`. The debugger is now in control and is ready for more commands as indicated by that the prompt is back.

To terminate the session and leave the debugger, use the `quit` command (or `q` for short).

More on running from the command line can be found in Console command line.

2.2.2 Machine interface

The MI (Machine Interface) is intended for debugger front-ends like Visual Studio Code. To start the Calypsi debugger in MI mode, add the `-i mi` or `--interpreter mi` to the command line:

```
$ dbnut --load-module 41cx.mod -i mi myimage
```

If you try it you will find that the MI mode is quite a bit less friendly than the command console. However, it does provide additional commands and some commands provide additional information that are not given in the console mode. For a very experienced user it may sometimes be desirable to issue an MI command, but that is better done using the console command `interpreter-exec MI "command"` which invokes an MI command from the console. This provides you with the usual powerful console mode and allows occasional use of MI commands, should the need arise.

Note: Switching between console and MI modes once the debugger is started is not possible. However, both modes provide a command that allows a command to be issued in a specified command interpreter.

CHAPTER 3

Console command line

The console interpreter is intended for interactive command line use. In this mode you have access to a powerful command line interpreter with command completion, command history and command line editing.

Your command line history is also saved between sessions in `~/.dbnut/dbnut.history`. You will also find a file that preserves the state of your HP-41 calculator in the same directory.

3.1 Command structure

Commands are made up from one or more space separated words. If a given word is long enough to be a unique match, it does not need to spelled out in full.

After the command there may be an argument that is specific to the command, i.e. a string, file path or boolean value.

3.1.1 Command completion

Command completion is available using the `TAB` key. Pressing `TAB` directly at the prompt provides a list of all words that can be used to start a command:

```
(dbnut) <TAB>
b                enable           n                s
break            exec-file        next             set
c                file             nexti            show
cd               frame            ni               si
clear            ignore           p                source
complete         info             platform         step
condition        interpreter-exec plug-in          stepi
continue         interrupt        print            thread
delete           kill             pwd              tty
directory        lua              quit
disable          maintenance      r
```

(continues on next page)

(continued from previous page)

```
disassemble       module            run
(dbnut)
```

Pressing TAB will complete the command to the left of the cursor as far as possible. If the word can be partially completed the first TAB will fill in as far as possible:

```
(dbnut) int<TAB>
```

Results in:

```
(dbnut) inter
```

Pressing TAB a second time results in:

```
(dbnut) inter<TAB>
interpreter-exec  interrupt
(dbnut) inter
```

3.2 A simple session

To get a feeling for how to run a very simple debugging session here is how it can look.

```
$ dbnut --load-module 41cx.mod myimage
Calypsi debugger for Nut
listening for user interface connections on port 8080
(dbnut) b APX
breakpoint 1
(dbnut) info breakpoints
Num      Type            Disp  Enb  Lua/plug-in   Address     What
1        breakpoint      keep  y    no            0x82af      APX
(dbnut) r
running
```

Note: A command only need entered with enough characters to make it unique. For the command `info breakpoints` it is enough to type `i b` followed by return. This works because the command interpreter takes all words in account. The command line completer looks at what is to the left of the cursor, so if you type `i<TAB>` you will get all command alternatives that start `i`. To make it complete to `info` you need to type `inf<TAB>` followed by `b<TAB>`. However, if you type `i b<TAB>` it will expanded to `i breakpoints`.

Now switch to the user interface and turn the calculator on. If this is the first time you start it, you will be greeted with the well known `MEMORY LOST` message indicating that the calculator firmware has initialized the calculator properly for the first time.

Enter the number `48` in the HP-41 user interface. Now execute the `APX` command, which happens to be the name of the command as well as entry point label in the MCODE program.

Once you do `XEQ ALPHA APX ALPHA` the user interface freezes as the debugger takes control when the breakpoint is hit.

```
(dbnut) r
running
program hit breakpoint 1
```

(continues on next page)

(continued from previous page)

```
   1 #include "mainframe.h"
   2
   3               .pubweak APX
   4               .name   "APX"
-> 5 APX:          c=0                  ; initialize REG.9 for DIGENT
   6               c=c-1
   7               c=0     xs
   8               pt=     13
   9               lc      10           ; initial D.P. position
0x82af at stack/apx.s:5
(dbnut)
```

The breakpoint is hit at address `0x82af` in the source file `stack/apx.s:5`. Some source lines are shown as context with the current line indicated.

We can now perform a couple of single steps:

```
(dbnut) s
    2
    3               .pubweak APX
    4               .name   "APX"
    5 APX:          c=0                  ; initialize REG.9 for DIGENT
->  6               c=c-1
    7               c=0     xs
    8               pt=     13
    9               lc      10           ; initial D.P. position
   10               regn=c  9
0x82b0 at stack/apx.s:6
(dbnut)
    3               .pubweak APX
    4               .name   "APX"
    5 APX:          c=0                  ; initialize REG.9 for DIGENT
    6               c=c-1
->  7               c=0     xs
    8               pt=     13
    9               lc      10           ; initial D.P. position
   10               regn=c  9
   11               gosub   STBT10       ; copy status bits for DIGENT
0x82b1 at stack/apx.s:7
(dbnut)
```

We use the step command that can be entered as `s`. This steps one source line. The second step we simply press `RETURN` which repeats the previous command (`s` in this case).

Each step moves us one line further ahead and the source context shown moves along with it.

We can inspect registers with `info registers`.

```
(dbnut) info registers
pc                  82b1
cy                     1 yes
a         00000008000000 0.0000008000.000
b         0000000000e005 0.000000000e.005
c         ffffffffffffff f.fffffffff.fff
m         900a38582afa80 9.00a38582af.a80
n         00000008000000 0.0000008000.000
g                    a3
```

(continues on next page)

(continued from previous page)

```
f                    00
st                   00
sth                  02
p                     1
q                     d
pactive               1 yes
cycles  000000000000487c
(dbnut)
```

The first column shows the register name, the second is the hex value of it and the third column shows the same value in a more decorative way (for some registers).

Hint: The `cycles` register is the cycle counter as a 64-bit hex number. This is very useful for evaluating performance of sections of code. Simply stop at strategic places and read the `cycles` register, then compute the difference.

Hint: A good calculator which can work in different number bases and word sizes is very useful when debugging MCODE. The HP-16C calculator or the Ladybug module for the HP-41 are very powerful tools for this.

Looking at the source code, we realize that we want to stop next at a certain line a bit further down, this can be done as follows:

```
(dbnut) b apx.s:49
breakpoint 2
(dbnut) c
running
program hit breakpoint 2
   45               acex    s
   46               rcr     -1
   47               pt=     0
   48               g=c
-> 49               gosub   DIGENT
   50               gosub   NOREG9        ; normalize and move to X
   51               bcex    x             ; decrement exponent
   52               c=c-1   x
   53               ?c#0    xs            ; negative (hit fractional part?)
0x82dd at stack/apx.s:49
(dbnut)
```

The `continue` command (short form `c`) is used to resume execution (not `run` which runs the program from start).

Here we want to see what we are passing in to `DIGENT`. We could display the registers again, but we can also show that value by just displaying the `G` register. We can do this by printing an expression (using the `print` or `p` command). A register name need to be preceded by a dollar sign:

```
(dbnut) p $g
$1 = 16
(dbnut) p/x $g
$2 = 0x10
(dbnut)
```

We got `16` and realized it was in decimal. Passing the `/x` modifier to the print command gives the value in hexadecimal.

We can also display a part of a register using a field. In this case we have a counter in the `X` (exponent) field of the `B` register:

```
(dbnut) p/x $b.x
$3 = 0x1
(dbnut)
```

It is apparently `1` at this point. Now step over the `gosub` we are standing at. This can be done using the `next` command (or `n` for short).

```
(dbnut) n
   46               rcr     -1
   47               pt=     0
   48               g=c
   49               gosub   DIGENT
-> 50               gosub   NOREG9        ; normalize and move to X
   51               bcex    x             ; decrement exponent
   52               c=c-1   x
   53               ?c#0    xs            ; negative (hit fractional part?)
   54               goc     25$           ; yes
0x82df at stack/apx.s:50
(dbnut)
```

We realize that we do not need to use the first breakpoint anymore. It can be removed using `delete`, but we decide to to disable it using the `disable` command instead:

```
(dbnut) disable 1
(dbnut) info breakpoints
Num      Type            Disp  Enb  Lua/plug-in   Address     What
1        breakpoint      keep  n    no            0x82af      APX
         breakpoint already hit 1 time
2        breakpoint      keep  y    no            0x82dd      apx.s:49
         breakpoint already hit 1 time
(dbnut)
```

Here we can see that the first breakpoint still exists, but it is not enabled (`n` in the `Enb` column). We can also see that the debugger also keeps track of statistics about the breakpoints.

CHAPTER 4

Visual Studio Code

Visual Studio Code[2] is a lightweight but powerful source code editor which runs on multiple platforms.

With the C/C++ for Visual Studio Code extension[3] it is possible to use Calypsi in Visual Studio Code.

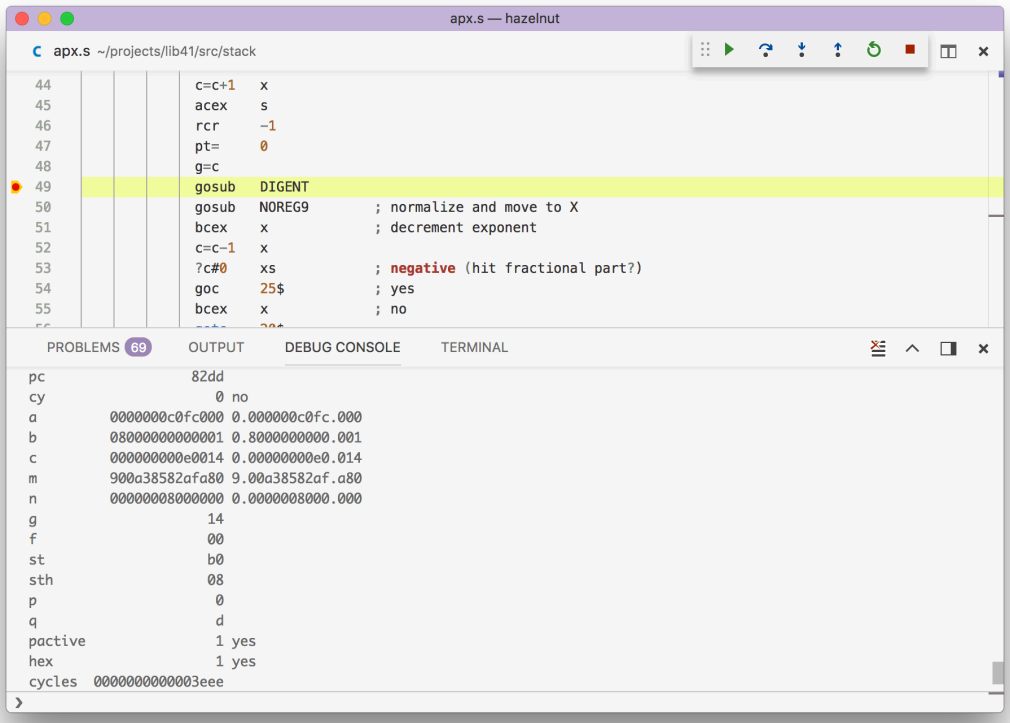
This works as Visual Studio Code can be tailored to use different build and debugger tools. The most crucial part is perhaps the debugger integration which is based on the MI (Machine Interface), which is also what the Calypsi debugger supports.

Visual Studio Code is a good choice for a graphical user interface to Calypsi as it is very actively developed, lightweight, responsive and highly configurable.

Documenting the full Visual Studio Code is beyond the scope of this guide. Refer to the official Visual Studio Code documentation[4] for detailed information.

Some basics on how to get started is covered here as it may not work entirely as you may expect from traditional IDEs.

[2] https://code.visualstudio.com/
[3] https://marketplace.visualstudio.com/items?itemName=ms-vscode.cpptools/
[4] https://code.visualstudio.com/docs



4.1 Installation

Installation is easy to do by downloading it from the official site. You can easily update it inside Visual Studio Code itself.

4.2 Project setup

In contrast to many IDEs you will not find any project setup in the menus.

To create a new project from scratch, use `File>Open` and create a new folder for your project and then press `Open`. You now have an empty project directory.

If you have an existing project, you can just select `File>Open` and browse to the top folder in that project, the press `Open` and you have loaded your project.

Project specific files are kept in an `.vscode` directory in your project directory. Initially there are no such files.

4.2.1 Building

Once you have populated your project and perhaps created a `Makefile`, you can add a task to build it. Select `Tasks>Configure Tasks...` and pick `Create tasks.json file from template`. Then select `Others` to get a `tasks.json` that you can edit. It may end up looking something like:

```
{
    "version": "0.2.0",
    "tasks": [
        {
            "label": "build",
            "type": "shell",
            "command": "make -k",
            "options": {
                "cwd": "${workspaceRoot}/src"
            },
            "problemMatcher": [
                "$gcc"
            ]
        }
    ]
}
```

Here we tell it that we want to run `make -k` in the `src` sub-directory and match error output to something that looks like what `gcc` would produce.

4.2.2 Debugging

To start the debugger you need to add a launch configuration.

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": "C++ Launch",
            "type": "cppdbg",
            "request": "launch",
            "program": "${workspaceRoot}/src/myproject",
            "args": [],
            "stopAtEntry": false,
            "cwd": "${workspaceRoot}/src",
            "environment": [],
            "externalConsole": true,
            "logging": {
                "trace": true,
                "traceResponse": true,
                "engineLogging":true
            },
            "linux": {
                "MIMode": "lldb"
            },
            "osx": {
                "MIMode": "lldb",
                "miDebuggerPath": "dbnut",
                "launchCompleteCommand": "exec-run",
                "setupCommands": [
```

(continues on next page)

(continued from previous page)

```
                    {
                        "text": "module insert /Users/me/projects/modules41/41cx.mod",
                        "description": "load mainframe",
                        "ignoreFailures": false
                    }
                ]
            },
            "windows": {
                "MIMode": "gdb"
            }
        },
    ]
}
```

There are a couple of things to specify here.

1. We need to specify that we are using `dbnut` as the debugger using `miDebuggerPath`
2. We also need to tell that we want to use `lldb` as `MIMode`[5]
3. We also need to load the calculator firmware. Here it is done using `setupCommands` that invokes the debugger console command `module insert`.

[5] Even though `dbnut` is much closer to `gdb` than `lldb` at the command level, we need to specify `lldb` as the MI mode. The reason is that in the `gdb` mode, Visual Studio Code works around a bug on UNIX style platforms for the MI command `-exec-interrupt` (which interrupts execution). The work around is it to send a signal to a real UNIX process to interrupt it. As we are running in a simulation environment the process is not a normal UNIX process and cannot be interrupted that way. Instead we use the `lldb` mode where MI command `-exec-interrupt` works properly, just as it does with `dbnut`.
Reference https://sourceware.org/bugzilla/show_bug.cgi?id=20035

CHAPTER 5

Emacs

Emacs is a powerful editor and much more. Being a GNU project it also comes with a GDB integration that also works with the Calypsi debugger.

You will need to use the GDB-MI mode for Emacs which is based on the MI (Machine Interface) protocol. Old GDB integrations for Emacs will not work with Calypsi debugger.

5.1 Starting

It is easy to start the Calypsi debugger on a recent Emacs, simply use the `M-x gdb` command. Emacs will then prompt for the command line to use. Enter something like:

```
dbnut -i mi --load-module /path-to/41cx.mod mybinary
```

Replace `/path-to/` with the path to your HP-41 operating system image. Here we are using an image for the HP-41CX.

We also need to provide the path to the ELF/DWARF binary you have built with the Calypsi tool chain, which is the image to debug. In order to have proper debugging information you need to build with the `-g` option.

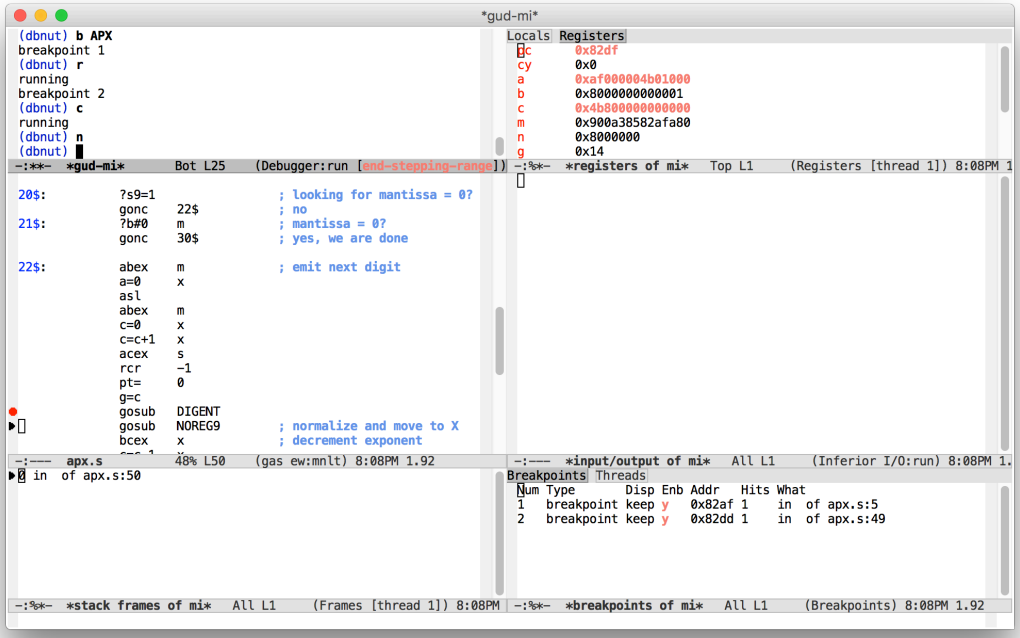
5.2 Stopping

Simply issue the `quit` command on the debugger command line inside Emacs.

5.3 Many windows mode

Out of the box Emacs/GDB is typically used with a command line window[6] and a source window.

With GDB-MI it may be a good idea to issue the command `M-x gdb-many-windows` which configures Emacs with a set of useful windows for debugging. Here you will find a command line window that works like the usual console mode (with command completion and everything). A register window that updates as you step around is also available. To top it off, your source buffer now shows the line you are at with a solid arrow. You can set breakpoints by clicking in the fringe of the source buffer.



5.3.1 More information

The official documentation can be opened inside Emacs using the `M-x info` command.

Select `Gdb` (takes you to Debugging with GDB), then select `Emacs` (takes you to Using GDB under GNU Emacs). Here you should also find a link to `GDB Graphical Interface` which describes the Many Windows mode in detail.

[6] A window in Emacs is a view to a buffer. (A buffer often corresponds to an open file, but it does not have to be tied to a file.)

CHAPTER 6

Console command reference

There is a large number of console commands supported, but you only need to be familiar with a small subset to make good use of the Calypsi debugger.

6.1 Essential commands

Table 6.1: Essential commands

Command	Description
`b` label	Set a breakpoint at given label
`run`	Start the program
`c`	Continue execution
`n`	Step to next line, skipping over calls
`s`	Step to next line, going into calls
`quit`	Exit the debugger
`Ctrl-c`	Interrupt execution

6.2 Execution control

Table 6.2: Execution control

Command	Description
`Ctrl-c`	Interrupt execution
`interrupt`	Interrupt execution
`r`	Start the program
`run`	Start the program
`s`	Step to next source line, going into calls
`step`	Step to next source line, going into calls
`si`	Step at instruction level, going into calls
`stepi`	Step at instruction level, going into calls
`n`	Step to next source line, skipping over calls
`next`	Step to next source line, skipping over calls
`ni`	Step at instruction level, skipping over calls
`nexti`	Step at instruction level, skipping over calls
`c`	Continue execution
`continue`	Continue execution
`kill`	Abort execution

6.3 Breakpoint commands

Created breakpoints are given a numeric identity. Conditional expressions and skip counts for breakpoints are supported.

Table 6.3: Breakpoint commands

Command	Description
`info breakpoints`	Display all breakpoints
`b` label	Set a breakpoint at given label
`b` [file:]line	Set a breakpoint at source position
`b` address	Set a breakpoint at given address. The address must be preceded by a `*`. A banked address can be entered as with `bank:`, i.e. `*2:0x5234`.
`break` arg	Same as `b`
`clear` label	Remove breakpoint at given label
`clear` [file:]line	Remove breakpoint at source position
`clear` address	Remove breakpoint at given address.
`delete` [n]	Remove breakpoint with given number, or all breakpoints if no argument
`enable` [n]	Enable breakpoint with given number, or all breakpoints if no argument
`disable` [n]	Disable breakpoint with given number, or all breakpoints if no argument
`condition` n expr	Attach a conditional expression to given breakpoint

6.4 Parameters

Parameters are provided to control behavior.

Table 6.4: Parameter commands

Command	Description
`set` parameter value	Set a parameter
`show` parameter	Display the value of a parameter

Note: Many parameters are booleans, they can be set to true using `on` or `1`, and to false using `off` or `0`.

Table 6.5: Parameters

Parameter	Description
`architecture`	The target architecture
`breakpoint pending`	Whether to allow pending breakpoints when a breakpoint cannot be set
`directories`	Prepend the search path for source files
`endian`	Target endian, normally derived from the debug image
`host-charset`	The character set used by the host (normally UTF-8)
`language`	The source language being used
`logging`	Controls logging to file
`lua errors`	Controls how Lua errors are handled
`mi-async`	Whether MI commands are processed or not when target is running (Calypsi always try to processes MI commands, regardless of the state of the program being debugged)
`print sevenbit-strings`	Controls how non-ASCII strings are displayed
`prompt`	The current prompt text
`stop-line-count-before`	Number of source lines to show before the current line when stopping
`stop-line-count-after`	Number of source lines to show after the current line when stopping, plus 1 for the current line itself
`target-charset`	The character set used by the target (normally UTF-8)
`target-wide-charset`	The wide character set used by the target (normally UTF-32)

The following parameters are recognized, but are not implemented and purely exist for the benefit of various debugger front ends which may issue them to initialize the debugger session.

Table 6.6: Other parameters

Parameter	Description
`auto-solib-add`	Whether to load symbols automatically when shared libraries are being loaded
`detach-on-fork`	Only present for compatibility with debugger front ends
`inferior-tty`	The terminal used by program being debugged
`pagination`	Whether pagination is enable or not (pagination is not currently supported)
`non-stop`	Allow program threads to be examined while other threads are running freely
`print object`	Whether an object's type should be based on vtable information
`set height`	The height of a paginated display
`solib-search-path`	Search path for shared libraries
`stop-on-solib-events`	Whether to stop when shared libraries are being loaded or unloaded
`target-async`	Only for compatibility with older versions, use `mi-async` instead

6.5 Data manipulation

Table 6.7: Data manipulation

Command	Description
`info registers`	Pretty print the register values
`p` expr	Evaluate an expression
`print` expr	Evaluate an expression (same as `p`)
`disassemble` start, end	Disassemble a region of memory

6.6 File commands

Table 6.8: File commands

Command	Description
`exec-file` filepath	Specify the program to be debugged
`file` filepath	Specify the program to be debugged (same as `exec-file`)
`info sources`	Show the program source files and how they are mapped to real files on the file system
`info source`	Show the current program source file

6.7 Directory context

Table 6.9: Directory context

Command	Description
`pwd`	Display current directory
`cd` path	Set current directory
`directory`	Show the search path for source files

6.8 Thread commands

Table 6.10: Thread commands

Command	Description
`info threads`	Show the existing threads
`thread` n	Switch context to thread n

6.9 Lua commands

Table 6.11: Lua commands

Command	Description
`source` filepath	Run a Lua script
`info lua instances`	Show existing Lua instances
`lua remove instance` n	Remove a Lua instance

6.10 HP-41 modules

In addition to providing modules on the command line you can do it using commands as well. This allows you to display and alter set of plugged in modules without ending the debug session.

Table 6.12: HP-41 modules

Command	Description
`module insert` filepath	Insert a plug-in module, `.mod` extension
`module list`	Display the inserted modules
`module remove` n	Remove the plug-in module that occupy the given page number

Note: You should turn the calculator off before altering the set of modules to avoid any potential strange behavior. This is because the calculator operating system as well as some modules, have code which configures the system for the current set of modules at power on.

6.11 Miscellaneous

Table 6.13: Miscellaneous

Command	Description
`show version`	Show the current version of Calypsi debugger
`quit`	Exit the debugger
`interpreter-exec mi` cmd	Run an MI command
`source` filepath	Run a Lua script
`complete` text	Show completions to given command

Index

B

breakpoint commands, 20

C

command
    completion, 7
    structure, 7
command line, 5
commands
    breakpoints, 20
    data manipulation, 22
    directory, 22
    essential, 19
    execution control, 19
    files, 22
    HP-41 modules, 23
    Lua, 23
    miscellaneous, 23
    parameters, 20
    threads, 22
console mode, 5
context
    program, 8, 21

D

data manipulation commands, 22
directory commands, 22

E

Emacs, 16
execution control, 19

F

file commands, 22

H

HP-41, 4
HP-41 module commands, 23
HP-41 user interface, 4

I

IDE
    Emacs, 16
    Visual Studio Code, 11

L

Lua commands, 23

M

machine interface, 5
mainframe, 4
MI, 5
miscellaneou commands, 23
mode
    console, 5
    MI, 5

P

parameter commands, 20
parameters, 21
program context, 8, 21

T

thread commands, 22

U

user interface, 4

V

Visual Studio Code, 11

W

web application, 4
web server, 4