Getting Started with the Open Dylan Command Line Tools

Release 1.0

Dylan Hackers

December 15, 2018

1 Copyright **3**

2 Hello World **5**

3 Using Source Registries **7**

4 Adding New Libraries **9**
4.1 Adding a Git Submodule . . . 9
4.2 Updating a Git Submodule . . . 9
4.3 Setting Up Registry Entries . . . 9
4.4 Transitive Dependencies . . . 10

5 A Few More Quick Tips **11**

6 Using dylan-compiler interactively **13**

7 An example of dylan-environment interactive functionality **15**

8 Editor Support **17**
8.1 Atom . . . 17
8.2 Code Mirror . . . 17
8.3 Emacs . . . 17
8.4 IntelliJ . . . 17
8.5 Light Table . . . 17
8.6 Sublime Text . . . 17
8.7 Textmate . . . 18
8.8 Vim . . . 18

9 Dylan Interactor Mode for Emacs (DIME) **19**

10 Debugging with GDB or LLDB **21**
10.1 Which compiler backend are you using? . . . 21
10.2 Debugging with the C backend . . . 21
10.3 Debugging with the HARP backend . . . 24

11 Notes for Windows Users **25**
11.1 dylan-compiler . . . 25
11.2 Build Products Location . . . 25
11.3 Setting Environment Variables . . . 25

12 Environment Variables **27**

13 Cross Compilation **29**

14 Platform Specific Projects **31**
14.1 LID File 31
14.2 LID File Inheritance 32
14.3 Registry 32
14.4 Code Layout 32

Index **33**

In Open Dylan, you can develop Dylan applications using the IDE (on Windows) or command-line tools.

The compiler executable is called `dylan-compiler`. There is a helper application called `make-dylan-app`, which can be used to generate some boilerplate for a new project, and finally there's `dswank` which is a back-end for interactive development in Emacs. This document describes these command-line tools.

For help getting started with the IDE on Windows, see the Getting Started with the Open Dylan IDE guide.

CHAPTER

ONE

COPYRIGHT

CHAPTER

TWO

HELLO WORLD

You have just downloaded Open Dylan and installed it in `/opt/opendylan-2014.1`. So how do you write the canonical Hello World app? This example assumes bash is being used. You may need to adjust for your local shell.

```
$ export PATH=/opt/opendylan-2014.1/bin:$PATH
$ make-dylan-app hello-world
$ cd hello-world
$ dylan-compiler -build hello-world.lid
...lots of output...
$ _build/bin/hello-world
Hello, world!
```

Ta da! Now a quick review of the steps with a little bit of explanation.

First you must set PATH so that `make-dylan-app` and `dylan-compiler` will be found. `./_build/bin` is where dylan-compiler puts the executables it builds.

Note: Some of these differ on Windows, so please be sure to read Notes for Windows Users if you are on Windows.

`make-dylan-app` creates a directory with the same name as the application and three files:

1. hello-world.lid – This says what other files are part of the project. The order in which the files are listed here determines the order in which the code in them is loaded.
2. library.dylan contains simple library and module definitions. These can be extended as your project grows more complex.
3. hello-world.dylan contains the main program.

The first time you build hello-world it builds all used libraries, all the way down to the dylan library itself. Subsequent compiles will only need to recompile hello-world itself and will therefore be much faster.

`dylan-compiler` has both a batch mode and an interactive mode. The `-build` option says to build the project in batch mode. When you pass a .lid file to the compiler it builds the library described by that file. In the next section you'll see that it can also pass the name of the project (without ".lid") and it will use "registries" to find the project sources.

The compiler places its output in the `_build` directory in the current working directory. This includes the libraries and executables that it builds. You can run the executable as noted above from this location.

CHAPTER

THREE

USING SOURCE REGISTRIES

Passing the name of a .lid file to `dylan-compiler` works great when you have a single library that only uses other libraries that are part of Open Dylan, but what if you want to use a second library that you wrote yourself? How will `dylan-compiler` find the sources for that library? The answer is registries. For each Dylan library that isn't part of Open Dylan itself, you create a file in the registry that points to the .lid file for the library. Here's an example for hello-world:

```
$ mkdir -p src/registry/generic
$ echo abstract://dylan/hello-world/hello-world.lid > src/registry/generic/hello-world
$ export OPEN_DYLAN_USER_REGISTRIES=`pwd`/src/registry
```

What's going on here? First of all, the registry mechanism makes it possible to have platform specific libraries. Anything platform-independent can be put under the "generic" directory. Other supported platform names are `x86_64-freebsd`, `x86-linux`, `x86-win32`, etc. For a full list see the Open Dylan registry.

Platform-specific registry directories are searched before the "generic" registry, so if you have a library that has a special case for Windows, you could use two registry entries: one in the "x86-win32" directory and one in the "generic" directory.

Now let's look at the actual content of our hello-world registry file:

```
abstract://dylan/hello-world/hello-world.lid
```

What this is doing is locating a file *relative to the directory that the registry itself is in*. If the "registry" directory is `/home/you/dylan/registry` then this registry file says the hello-world .lid file is in `/home/you/dylan/hello-world/hello-world.lid`. "abstract://dylan/" is just boilerplate.

Once you've set the `OPEN_DYLAN_USER_REGISTRIES` environment variable to point to our new registry, `dylan-compiler` can find the hello-world library source no matter what directory you're currently working in. You only need to specify the library name:

```
$ cd /tmp
$ dylan-compiler -build hello-world
```

You can add more than one registry to `OPEN_DYLAN_USER_REGISTRIES` by separating them with colons:

```
$ export OPEN_DYLAN_USER_REGISTRIES=/my/registry:/their/registry
```

CHAPTER

FOUR

ADDING NEW LIBRARIES

We do not yet have a packaging system, so this document lays out how we currently handle inter-library dependencies.

Adding a Git Submodule

The current way of handling inter-library dependencies is to use git submodules. This allows you to specify a precise version that you rely upon, but assumes that we're all using git.

We tend to keep all git submodules in a top level directory within the repository named `ext`. To add a new submodule:

```
git submodule add <repository url> ext/<name>
```

The *repository url* should be a publicly accessible URL, so it is recommended to use either the git or https protocols (`git://` or `https://`) rather than SSH (`git@`).

The `name` should be the name of the repository.

For example, to add the `tracing` library as a submodule, one would:

```
git submodule add https://github.com/dylan-foundry/tracing.git ext/tracing
```

Updating a Git Submodule

If the submodule has been updated to point at a new revision, after you do a `git pull`, you will want to update your submodules:

```
git submodule update --init --recursive
```

If you want to update the submodule to point to a new revision, then you would:

```
cd ext/<name>
git pull --ff-only origin master
cd ../..
git add ext/<name>
git commit -m 'Updated <name>.'
```

Setting Up Registry Entries

For each library that you add as a submodule, you will need to create a registry entry so that the Open Dylan compiler can find the library. See Using Source Registries for more detail.

In the case of the tracing library, you would create a new file, `registry/generic/tracing-core`, with the contents:

```
abstract://dylan/ext/tracing/tracing-core/tracing-core.lid
```

You can usually get a good idea for what registry entries are needed by looking into the registry directory of the library that you're using.

Transitive Dependencies

The Dylan compiler won't find transitive dependencies, so you will need to create registry entries for them as well.

Sometimes, you will want to create git submodules for them as well, but other times you can just reference them from the version that was pulled in with the existing submodule.

As an example, if you pull in the HTTP library, it has a number of submodules, so you don't need to pull each of those in directly, but can reference them through the `ext/http/` directory. (Note in this case that the `http` library uses a non-standard name for the directory holding its submodules.)

CHAPTER

FIVE

A FEW MORE QUICK TIPS

1. Add `-clean` to the command line to do a clean build:

```
dylan-compiler -build -clean /my/project.lid
```

2. Use `dylan-compiler -help` to see all the options. Options that don't take an argument may be negated by adding "no". e.g. -nologo
3. The `-build` option builds an executable unless you add this line to your .lid file:

```
target-type: dll
```

4. If you miss having rich command line history and similar features, use `rlwrap` with `dylan-compiler`.

You should now have enough information to start working on your Dylan project. The next few sections go into more detail on using `dylan-compiler`, which also has an interactive mode that can make the edit/build/debug cycle a bit faster. Or if you're an Emacs user you may prefer to jump directly to the section on the Dylan Interactor Mode for Emacs (DIME).

CHAPTER

SIX

USING DYLAN-COMPILER INTERACTIVELY

The interactive mode of `dylan-compiler` allows you to carry out multiple development tasks over a period of time without having to restart the console compiler each time. To start the console environment in interactive mode, enter `dylan-compiler` without any arguments at a shell. For example:

```
$ dylan-compiler
Hacker Edition
Version 2014.1
Copyright (c) 1997-2004, Functional Objects, Inc.
Portions Copyright (c) 2004-2014, Dylan Hackers
Portions Copyright (c) 2001-2012, Ravenbrook Ltd.
>
```

If you've used the Open Dylan IDE on Windows, note that using `dylan-compiler` interactively is similar to working in the IDE's interactor.

You can find a list of command groups by entering the command `help`. The command groups in the console compiler are:

Command Group	Description
BASIC	basic commands
BROWSING	browsing commands
BUILD	project building commands
INTERNAL	internal commands
LIBRARY-PACKS	library packs commands
PROJECT	project commands
PROPERTY	property handling commands
REGISTRY	registry commands
REPORTS	report commands
SYSTEM	operating system commands

You can use `help -group group-name` to view the available commands and properties of a specific group. You can also use `help command-name` to view the full documentation of a command. We can see the kind of information available by looking at the help entry for the `help` command:

```
> help help
Usage: :HELP [options*] [command]

If specified with no arguments, HELP shows a list of all commands
with a one line description. Help can display command options by
specifying the name of the command. Additionally, it can display
group or property help by specifying the GROUP or PROPERTY option.

Arguments:
  COMMAND - the command to describe
```

```
Options:
  -GROUP group - the command group to describe
  -PROPERTY property - the property to describe
```

Therefore, to find out what commands exist within the *PROJECT* command group, type:

```
> help -group project

PROJECT:

Commands applying to projects.

Commands:
  CLOSE   closes the specified project
  IMPORT  imports a LID file
  OPEN    opens the specified project

Properties:
  PROJECT   Current project
  PROJECTS  Open projects

For documentation on a group, use:    HELP -GROUP group.
For documentation on a command, use:  HELP command
For a complete list of commands, use: SHOW COMMANDS
```

Then, to examine the `OPEN` command, type:

```
> help open
Usage: OPEN file

Opens the specified project.

Arguments:

  FILE - the filename of the project
```

Properties can be displayed via the `show` command. For example to see the value of the "projects" property listed previously, use `show projects`.

To exit the console environment, use the command `exit`.

CHAPTER

SEVEN

AN EXAMPLE OF DYLAN-ENVIRONMENT INTERACTIVE FUNCTIONALITY

Note: `dylan-environment` is currently only supported on Windows. Unix users may wish to skip this section.

The dylan-environment has a few more options and command groups, which will be presented briefly here:

Options	Description
-ARGUMENTS arguments	Arguments for the project's application
-PLAY	Open and debug the playground project
-START	Start the project's application
-DEBUG	Debug the project's application
-PROFILE	Profile the execution of the application
-SHARE-CONSOLE	Share the console with the application

Command Group	Description
BREAKPOINTS	breakpoint commands
DEBUGGING	debugging commands
MEMORY	memory viewing commands
REMOTE-DEBUGGING	remote debugging commands
STACK	stack commands

The following example demonstrates the console environment's interactive functionality. In the example, the user starts dylan-environment in interactive mode, opens the playground project, performs some arithmetic, defines a method, and then traces it:

```
# dylan-environment
Hacker Edition
Version 2014.1
Copyright (c) 1997-2004, Functional Objects, Inc.
Portions Copyright (c) 2004-2014, Dylan Hackers
Portions Copyright (c) 2001-2012, Ravenbrook Ltd.

> play
Opened project gui-dylan-playground
Starting: gui-dylan-playground
? 1 + 2;
  $0 = 3
? define method factorial (x) if (x < 2) 1 else x * factorial(x - 1) end end;
? factorial(5);
  $1 = 120
? :trace factorial
? :set messages verbose
Messages: verbose
? factorial(6);
0: factorial (<object>): (6)
```

```
  1: factorial (<object>): (5)
    2: factorial (<object>): (4)
      3: factorial (<object>): (3)
        4: factorial (<object>): (2)
          5: factorial (<object>): (1)
          5: factorial (<object>) => (2)
        4: factorial (<object>) => (6)
      3: factorial (<object>) => (24)
    2: factorial (<object>) => (120)
  1: factorial (<object>) => (720)
0: factorial (<object>) => (#[720])
  $2 = 720
? :exit
```

The commands described in this appendix can also be used in the Command Line window within the regular Open Dylan development environment. Choose **File > Command Line...** from the main window and use commands at the *?* prompt.

CHAPTER

EIGHT

EDITOR SUPPORT

Dylan support is available in a broad range of editors.

In alphabetical order:

Atom

See the language-dylan package.

Code Mirror

Support for Dylan will be available out of the box in the near future.

Emacs

See Dylan Interactor Mode for Emacs (DIME).

IntelliJ

See the DeftIDEA plugin.

Light Table

Support will be forthcoming.

Sublime Text

Dylan support is available via Package Control. This is a conversion of the Textmate plugin.

Textmate

See dylan.tmbundle.

Vim

Support for Dylan is available out of the box. However, enhanced support is available in dylan-vim.

CHAPTER

NINE

DYLAN INTERACTOR MODE FOR EMACS (DIME)

DIME and its back-end, dswank, create a link between the Dylan compiler and emacs so that editor commands can leverage everything the compiler knows about your source code. It allows you to view cross references, locate definitions, view argument lists, compile your code, browse class hierarchies, and more. This section will give a brief introduction to using DIME.

The first thing you need to use DIME is the emacs Lisp code for dylan-mode, which can be downloaded from the dylan-mode GitHub repository. If you don't have ready access to git there is a link on that page to download as a .zip file.

Next set up your .emacs file as follows. Adjust the pathnames to match your Open Dylan installation location and the directory where you put dylan-mode.

```
(add-to-list 'load-path "/path/to/dylan-mode")
(setq inferior-dylan-program "/opt/opendylan/bin/dswank")
(require 'dime)
(dime-setup '(dime-dylan dime-repl))
(setenv "OPEN_DYLAN_USER_REGISTRIES" "/path/to/your/registry:...more...")
```

Setting `OPEN_DYLAN_USER_REGISTRIES` is important because that's how DIME finds your projects.

For this tutorial let's use a "dime-test" project created with `make-dylan-app`. See the section Hello World to create the project, and also make sure you have a registry entry for it. See Using Source Registries if you're not sure how to set that up.

Start dime:

```
$ export PATH=/opt/opendylan/bin:$PATH
$ cd ...dir containing registry...
$ echo abstract://dylan/dime-test/dime-test.lid > registry/generic/dime-test
$ make-dylan-app dime-test
$ cd dime-test
$ emacs dime-test.dylan
M-x dime <Enter>
```

You should now have a buffer called `*dime-repl nil*` that looks like this:

```
Welcome to dswank - the Hacker Edition Version 2014.1 SLIME interface
opendylan>
```

This is the Open Dylan compiler interactive shell. You can issue commands directly here if you like, but mostly you'll issue dime commands from your Dylan source buffers.

Change projects: Switch back to the dime-test.dylan buffer and type `C-c M-p dime-test` to tell DIME to switch to the dime-test project. If DIME doesn't let you enter "dime-test" as the project name that means it couldn't find the registry entry. Press <Tab> to see a complete list of available projects.

Compile: To build the project, type `C-c C-k`. You should see something like "Compilation finished: 3 warnings, 18 notes". (The reason there are so many warnings is because there are some warnings in the dylan library itself. This is a bug that should be fixed eventually.)

Edit definition: There's not much code in dime-test.dylan except for a `main` method. Move the cursor onto the call to "format-out" and type `M-.`. It should jump to the format-out definition in the `io-internals` module.

Compiler warnings: Switch back to the dime-test.dylan buffer and make a change that causes a compiler warning, such as removing the semicolon at the end of the `format-out` line. Recompile with `C-c C-k` and you should see something like "Compilation finished: 6 warnings, 18 notes". You can jump to the first warning using the standard for emacs: `C-x `.

Argument lists: Note that when you type an open parenthesis, or comma, or space after a function name dime will display the **argument list** and return values in the emacs minibuffer. e.g., try typing `+ (`.

Cross references: To list cross references (e.g., who calls function F?) move the cursor over the name you want to look up and type `C-c C-w C-c` ('c' for call). DIME will display a list of callers in a `*dime-xref*` buffer. `C-M-.` will take you to the next caller. Use it repeatedly to move to each caller definition in turn. Move the cursor to a particular caller in the `*dime-xref*` buffer and press <Enter> to jump to that caller.

That should be enough to give you the flavor of DIME. Following is a table of useful commands, and you can of course find many more using the standard emacs tools such as `C-h b` and `M-x apropos`.

Keyboard shortcut	Effect
M-x dime	start dime
, change-project	change project (in the repl buffer)
C-c M-p	change project (in Dylan source buffers)
M-.	jump to definition
M-,	jump backwards
C-c C-k	compile project
C-c C-w C-a	who specializes? (or who defines?)
C-c C-w C-r	who references?
C-c C-w C-b	who binds?
C-c C-w C-c	who calls?

CHAPTER

TEN

DEBUGGING WITH GDB OR LLDB

Warning: Some of the facilities described here are only available in builds from April 11, 2013 or later.

Warning: On macOS, we recommend using lldb to debug rather than gdb.

Which compiler backend are you using?

If you are using the C backend, then the information discussed here will be useful for debugging. We use the C backend on 64 bit Linux, 64 bit FreeBSD and all macOS versions.

If you're using the HARP backend (32 bit Linux or 32 bit FreeBSD), then you'll be limited to getting stack traces.

Debugging with the C backend

Debugging with the C backend is not a perfect experience but it is improving.

Finding the generated C files

The C files generated by the compiler will be found under the `_build/build` directory with a directory in there for each library that was built as part of the project.

Finding the corresponding Dylan code

The generated C files contain the file name and line number of the corresponding Dylan source. However, this may be confusing in the presence of inlined functions and macro expansions.

Understanding name mangling

Fill this in, particularly with a link to the Hacker's Guide!

Understanding stack traces

Let's look at part of a representative stack trace:

```
#0  0x92b41b06 in read$NOCANCEL$UNIX2003 ()
#1  0x0042c509 in Kunix_readYio_internalsVioI ()
#2  0x000bc8e7 in xep_4 ()
#3  0x0042ba99 in Kaccessor_read_intoXYstreams_internalsVioMM0I ()
#4  0x000c0805 in key_mep_6 ()
#5  0x000c43a4 in implicit_keyed_single_method_engine_4 ()
#6  0x000c1dd5 in gf_optional_xep_4 ()
#7  0x004139fb in Kload_bufferYstreams_internalsVioI ()
#8  0x0041334a in Kdo_next_input_bufferYstreamsVioMM1I ()
#9  0x000c04ab in key_mep_4 ()
#10 0x000c3eaf in implicit_keyed_single_method_engine_1 ()
#11 0x0040520f in Kread_lineYstreamsVioMM0I ()
#12 0x000c0321 in key_mep_3 ()
#13 0x000c3eaf in implicit_keyed_single_method_engine_1 ()
#14 0x0079dcf6 in Kcommand_line_loopYcommand_linesVenvironment_commandsMM0I ()
#15 0x000bf6b3 in rest_key_xep_5 ()
#16 0x00007abe in Kdo_execute_commandVcommandsMdylan_compilerM0I ()
#17 0x000bb9bb in primitive_engine_node_apply_with_optionals ()
#18 0x0002b9ba in Khandle_missed_dispatchVKgI ()
#19 0x0002aaef in KPgf_dispatch_absentVKgI ()
#20 0x000c25e8 in general_engine_node_n_engine ()
#21 0x004b19a8 in Kexecute_commandVcommandsMM0I ()
#22 0x000bb9bb in primitive_engine_node_apply_with_optionals ()
#23 0x0002b9ba in Khandle_missed_dispatchVKgI ()
#24 0x0002aaef in KPgf_dispatch_absentVKgI ()
#25 0x000c25e8 in general_engine_node_n_engine ()
#26 0x000c11ab in gf_xep_1 ()
#27 0x0000aa9e in KmainYconsole_environmentVdylan_compilerI ()
#28 0x0000abb3 in _Init_dylan_compiler__X_start_for_user ()
```

Some things to notice:

- Method dispatch takes up the bulk of the stack frames with function calls like `gf_xep_1`, `general_engine_node_n_engine`, `rest_key_xep_5`, `key_mep_4`, `xep_4`, `implicit_keyed_single_method_engine_1` and so on.
- Methods representing Dylan code are mangled as discussed in a section above.
- It may seem alarming to see methods like `KPgf_dispatch_absentVKgI` or `Khandle_missed_dispatchVKgI` (where did dispatch go?!?). These function calls indicate that the dispatch data for a method hadn't been set up yet as this is the first invocation of that method. The method dispatch data is set up lazily, so this is normal and expected.
- There isn't any information on arguments or what file and line number contains the corresponding code. This means that you don't have the debug data for the compiler around. An easy way to address this is to build your own copy of the compiler.

Breaking on main

Unfortunately, you can't simply set a breakpoint on `main`. This is because the generated code runs from shared library constructor functions so the entire Dylan application runs and exits prior to `main` being invoked.

A reasonable alternative is to determine the C name of your entry point function and set a breakpoint on that instead.

Inspecting Dylan objects in LLDB

In LLDB, you may import our Dylan support library:

```
command script import /path/to/opendylan/share/opendylan/lldb/dylan
```

Do not import the scripts under that directory directly as that will not work. Once this has been done, variables that are representing Dylan values will automatically be shown with additional data about their actual values.

Inspecting Dylan objects in GDB

We do not yet have support for Dylan in GDB as we do for LLDB.

The C runtime contains a number of helper functions specifically for improving the debugging process. These can be invoked from gdb or lldb and will assist you in analyzing values.

D **dylan_object_class** (D* *instance*)
: Returns the class instance for the given instance object.

bool **dylan_boolean_p** (D *instance*)
: Tests whether instance is a <boolean>.

bool **dylan_true_p** (D *instance*)
: Tests whether instance is #t.

bool **dylan_float_p** (D *instance*)
: Tests whether instance is a <float>.

bool **dylan_single_float_p** (D *instance*)
: Tests whether instance is a <single-float>.

float **dylan_single_float_data** (D *instance*)
: Returns the float data stored in the instance.

bool **dylan_double_float_p** (D *instance*)
: Tests whether instance is a <double-float>.

double **dylan_double_float_data** (D *instance*)
: Returns the double data stored in the instance.

bool **dylan_symbol_p** (D *instance*)
: Tests whether instance is a <symbol>.

D **dylan_symbol_name** (D *instance*)
: Returns the string form of the given symbol.

bool **dylan_pair_p** (D *instance*)
: Tests whether instance is a <pair>.

bool **dylan_empty_list_p** (D *instance*)
: Tests whether instance is an empty list.

D **dylan_head** (D *instance*)
: Returns the head of the given <pair> instance.

D **dylan_tail** (D *instance*)
: Returns the tail of the given <pair> instance.

bool **dylan_vector_p** (D *instance*)
: Tests whether instance is a <vector>.

bool **dylan_string_p** (D *instance*)
Tests whether instance is a <string>.

char* **dylan_string_data** (D *instance*)
Returns the C string data stored in the given instance.

bool **dylan_simple_condition_p** (D *instance*)
Tests whether instance is a <simple-condition>.

D **dylan_simple_condition_format_string** (D *instance*)
Returns the format string stored in the given <simple-condition>.

D **dylan_simple_condition_format_args** (D *instance*)
Returns the format string arguments stored in the given <simple-condition>.

bool **dylan_class_p** (D *instance*)
Tests whether instance is a <class>.

D **dylan_class_debug_name** (D *instance*)
Returns the <string> object containing the class's name.

bool **dylan_function_p** (D *instance*)
Tests whether instance is a <function>.

D **dylan_function_debug_name** (D *instance*)
Returns the <string> object containing the function's name. Note that we do not store the name for all function objects.

void **dylan_print_object** (D *object*)
Print some information about the given object to stdout.

Debugging with the HARP backend

As mentioned previously, this is largely limited to getting stack traces. If you try to run a Dylan application built with the HARP backend under the debugger, you may need to adjust your debugger's signal handling as the Memory Pool System GC that is used employs the SIGSEGV signal.

To do this on Linux and FreeBSD in gdb, use this command:

```
handle SIGSEGV pass nostop noprint
```

Add more notes about this later.

CHAPTER

ELEVEN

NOTES FOR WINDOWS USERS

For users trying out the command line tools under Windows, there are some important differences.

dylan-compiler

On Windows, the `dylan-compiler` executable is called `dylan-compiler-with-tools.exe`.

The IDE is `win32-environment.exe`.

Both are located in `C:\Program Files\Open Dylan\bin\` on a 32 bit Windows version. On 64 bit Windows, they can be found within `C:\Program Files (x86)\Open Dylan\bin\`.

Build Products Location

Instead of placing build products within a `_build` directory, they are stored in a single location by default:

```
%APPDATA%\Open-Dylan\
```

The `APPDATA` defaults to `C:\Users\...\AppData\Roaming\`.

This can be modified by setting the `OPEN_DYLAN_USER_ROOT` environment variable.

Setting Environment Variables

Environment variables are managed differently on Windows. For modifying environment variables permanently, see this guide.

For modifying an environment variable from the command line, use this syntax:

```
set variable=value
```

CHAPTER

TWELVE

ENVIRONMENT VARIABLES

Open Dylan uses a variety of environment variables to control behavior, where to look for files, or where to place output files. In the common case, it is not necessary to set any of these variables.

OPEN_DYLAN_TARGET_PLATFORM: Used in Cross Compilation.

OPEN_DYLAN_USER_REGISTRIES: Controls where to look for registry entries to find libraries. See Using Source Registries.

Defaults to looking for a directory named `registry` in the current directory.

OPEN_DYLAN_USER_ROOT: The directory where build output is placed. This is the 'user root'.

On Windows, see Notes for Windows Users. On all other platforms, this defaults to the directory `_build` in the current directory.

OPEN_DYLAN_USER_BUILD: The directory within the 'user root' where things are built.

Defaults to `build`.

CHAPTER

THIRTEEN

CROSS COMPILATION

For now, see the porting section of the Hacker's Guide.

CHAPTER

FOURTEEN

PLATFORM SPECIFIC PROJECTS

When a project involves platform-specific code, some consideration needs to be made to the lid file and registry, as well as the layout of code. As an example, consider the `io` and system libraries, found in https://github.com/dylan-lang/opendylan/tree/master/sources/io/ and https://github.com/dylan-lang/opendylan/tree/master/sources/system/.

LID File

For further details of the LID file format, see LID file.

1. Library

   The library remains the same across all platforms since it is, after all, a platform-dependent version of that library:

Keyword	unix-io.lid	win32-io.lid
Library:	io	io

2. Files

   Each platform's project may contain files that are the same in each platform, as well as files which are present in one but not in another:

Keyword	unix-io.lid	win32-io.lid
Files:	buffered-format	buffered-format
	format-condition	format-condition
	unix-interface	win32-interface
	unix-file-accessor	win-file-accessor
	unix-standard-io	win32-standard-io
	format-out	format-out
	(etc)	*(etc)*
C-Source-Files:	unix-portability.c	
RC-Files:		version.rc

3. Linked Libraries (from dylan-lang/opendylan/sources/system)

   Each platform's project will probably require a different set of linked libraries:

Keyword	x86_64-linux-system.lid	x86-win32-system.lid
C-Libraries:	-ldl	advapi32.lib
		shell32.lib

Note: An example from the `system` library was used as the `io` library doesn't link directly against any C libraries.

LID File Inheritance

When LID files are almost identical, it can be useful to create a base LID file and inherit it, overriding whatever is necessary for each platform. A per-platform LID file might look like:

```
Library:        uv
LID:            uv-posix.lid
C-libraries:    -framework CoreServices
```

Registry

For further details of setting up the registry entries, see Using Source Registries.

Normally, when a reference to a (platform independent) project is placed in the registry, it is put into the generic directory. Platform dependent projects are placed in the platform-labelled subdirectories. *e.g.*

opendylan/sources/registry/x86_64-linux/io `abstract://dylan/io/unix-io.lid`

opendylan/sources/registry/x86-win32/io `abstract://dylan/io/win32-io.lid`

Code Layout

The `io` library is laid out in the following manner:

1. All platform-specific code is inside a single module (`io-internals`).
2. `*-interface.dylan` contains the low-level functions accessing the platform-specific libraries (*e.g.* `unix-read`, `win32-read`).
3. `*-file-accessor.dylan` uses the functions from (2) to produce a platform-independent API (*e.g.* `accessor-read-into!`).
4. Only those methods from (3) are exported from the module.

Note: Most libraries that are portable aren't as complex in their layout as the `io` library and don't require separate modules.

D

dylan-environment, 15
dylan_boolean_p (C function), 23
dylan_class_debug_name (C function), 24
dylan_class_p (C function), 24
dylan_double_float_data (C function), 23
dylan_double_float_p (C function), 23
dylan_empty_list_p (C function), 23
dylan_float_p (C function), 23
dylan_function_debug_name (C function), 24
dylan_function_p (C function), 24
dylan_head (C function), 23
dylan_object_class (C function), 23
dylan_pair_p (C function), 23
dylan_print_object (C function), 24
dylan_simple_condition_format_args (C function), 24
dylan_simple_condition_format_string (C function), 24
dylan_simple_condition_p (C function), 24
dylan_single_float_data (C function), 23
dylan_single_float_p (C function), 23
dylan_string_data (C function), 24
dylan_string_p (C function), 23
dylan_symbol_name (C function), 23
dylan_symbol_p (C function), 23
dylan_tail (C function), 23
dylan_true_p (C function), 23
dylan_vector_p (C function), 23

P

Platform Specific Projects, 31