

# THE ODIN SYSTEM

## REFERENCE MANUAL

Geoffrey M. Clemm

\*\*\* Odin Version 1.18 \*\*\*

The Odin System is a simpler, more powerful, and more reliable replacement for Make. It computes complete dependency information automatically, making the build scripts shorter and easier to manage. Odin gains efficiency by parallel builds on remote machines, by eliminating most of the file system status queries required by Make, and by sharing from a cache of previously computed derived files.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Derived Object Managers . . . . .	1
1.2	Odin Objects . . . . .	1
1.2.1	Source and Derived Objects . . . . .	1
1.2.2	The Derived Object Cache . . . . .	2
1.3	Why Not Just Use Make? . . . . .	2
1.3.1	Dependency Information . . . . .	2
1.3.2	Specifying Build Variants . . . . .	3
1.3.3	Storing Derived File Variants . . . . .	3
1.3.4	Parallel Builds . . . . .	4
1.3.5	Help . . . . .	4
1.3.6	Errors . . . . .	4
1.3.7	Operations on Lists . . . . .	4
1.3.8	Editing while Building . . . . .	4
1.3.9	Recursive Implicit Rules . . . . .	4
1.3.10	Circular Dependencies . . . . .	5
1.3.11	Efficient Dependency Computation . . . . .	5
1.3.12	Tool Dependencies . . . . .	5
1.3.13	Availability . . . . .	5
<b>2</b>	<b>The Odinfile</b>	<b>6</b>
2.1	Targets . . . . .	6
2.2	Odin-Expressions . . . . .	6
2.2.1	Lexical Conventions . . . . .	6
2.2.2	Selection Expressions . . . . .	7
2.2.3	Derivation Expressions . . . . .	7
2.2.4	Parameterization Expressions . . . . .	8
2.2.5	String Expressions . . . . .	9
2.3	Virtual Targets . . . . .	9
2.4	Text Targets . . . . .	10
2.4.1	Executable Text Targets . . . . .	11
2.5	Command Targets . . . . .	11
2.6	Nested Target Definitions . . . . .	11
<b>3</b>	<b>Getting Started</b>	<b>12</b>
3.1	Systems with Several Files . . . . .	12
3.2	Files in Other Directories . . . . .	13
3.3	Recursive Odin Calls . . . . .	14
3.4	Debugging Your Program . . . . .	14

<b>4</b>	<b>Odin-Commands</b>	<b>16</b>
4.1	Query Odin-Commands	16
4.1.1	Status of Odin Objects	17
4.1.2	Error and Warning Messages	17
4.2	Copy Odin-Commands	17
4.2.1	Copying to Standard Output	18
4.2.2	Editing with the Copy Odin-Command	18
4.3	Execute Odin-Commands	18
4.4	Variable Assignment Odin-Commands	18
4.4.1	Dir	19
4.4.2	BuildHosts, MaxBuilds	19
4.4.3	KeepGoing	19
4.4.4	History	20
4.4.5	ErrLevel, WarnLevel, LogLevel	20
4.4.6	HelpLevel	20
4.4.7	VerifyLevel	20
4.4.8	Size	20
4.4.9	Environment Variables	20
4.5	Help	21
4.5.1	Source Type Help	21
4.5.2	Derivation Help	22
4.5.3	Parameterization Help	22
4.5.4	Variable Help	23
<b>A</b>	<b>Dependency Database Utilities</b>	<b>24</b>
A.1	Testing Host Files for Changes	24
A.2	Redoing a Particular Derivation Step	25
A.3	Browsing the Dependency Database	25
<b>B</b>	<b>Tool Package Libraries</b>	<b>26</b>
B.1	Tool Packages	26
B.1.1	Versioned Tool Packages	26
B.2	Derivation Graphs	27
B.3	Source Declarations	27
B.4	Object Type Declarations	28
B.4.1	Built-In Supertypes	28
B.4.2	Built-In Derived Object Types	29
B.5	Parameter Type Declarations	32
B.6	Environment Variable Declarations	32
B.7	Tool Declarations	32
B.7.1	Input-Expressions	33
B.7.2	Ignoring the Status of a Tool Input	33
B.7.3	Ignoring Changes to the Value of a Tool Input	33
B.7.4	EXEC Tool	34
B.7.5	COLLECT Tool	35
B.7.6	READ-LIST Tool	35
B.7.7	READ-REFERENCE Tool	35
<b>C</b>	<b>Derivation Graph Syntax</b>	<b>36</b>
<b>D</b>	<b>Derivation Graph Example</b>	<b>38</b>

<b>E</b>	<b>Odin Tool Package Library</b>	<b>40</b>
E.1	config . . . . .	40
E.2	odin . . . . .	40
E.2.1	Other transformations . . . . .	41
E.3	vc . . . . .	44
E.3.1	SCCS version control file . . . . .	44
E.3.2	A view-path system model . . . . .	45
E.3.3	RCS version control file . . . . .	47
E.3.4	Other transformations . . . . .	49
E.4	code . . . . .	49
E.4.1	System model of source code files . . . . .	49
E.4.2	Object library archive . . . . .	50
E.4.3	Object module . . . . .	51
E.4.4	Other transformations . . . . .	53
E.5	cc . . . . .	53
E.5.1	System model for a C program . . . . .	53
E.5.2	Executable binary . . . . .	55
E.5.3	C source code . . . . .	57
E.5.4	Other transformations . . . . .	60
E.6	cpp . . . . .	61
E.6.1	Cpp input . . . . .	61
E.7	cxx . . . . .	63
E.7.1	System model for a C++ program . . . . .	64
E.7.2	C++ source code . . . . .	65
E.7.3	Other transformations . . . . .	67
E.8	odi . . . . .	68
E.8.1	System model for an ODI C++ program . . . . .	68
E.8.2	Other transformations . . . . .	70
E.9	f77 . . . . .	70
E.9.1	Fortran77 source code . . . . .	70
E.10	lex . . . . .	71
E.10.1	Scanner grammar . . . . .	71
E.11	yacc . . . . .	72
E.11.1	Yacc input . . . . .	73
E.12	tregrm . . . . .	75
E.12.1	Input grammar for the Tregrm tree-building parser generator . . . . .	76
E.13	dg . . . . .	77
E.13.1	Odin Derivation Graph . . . . .	77
E.14	idl . . . . .	82
E.14.1	C++ Interface Definition Language file . . . . .	83
E.14.2	Other transformations . . . . .	85
E.15	prof . . . . .	85
E.15.1	Other transformations . . . . .	85
E.16	lint . . . . .	85
E.16.1	Other transformations . . . . .	85
E.17	misc . . . . .	86
E.17.1	Other transformations . . . . .	86
E.18	roff . . . . .	87
E.18.1	Nroff/troff input . . . . .	87
E.18.2	Eqn input . . . . .	89
E.18.3	Tbl input . . . . .	90
E.18.4	Other transformations . . . . .	91

E.19	tex . . . . .	92
E.19.1	Basis of a TeX formatted document . . . . .	92
E.19.2	Bibliographic data file . . . . .	92
E.19.3	Bibliographic style file . . . . .	94
E.19.4	Formatting style file . . . . .	95
E.19.5	Portable Document Format file . . . . .	96
E.19.6	Formatting style file . . . . .	97
E.19.7	TeX formatter input . . . . .	99
E.19.8	Postscript file . . . . .	100
E.19.9	Other transformations . . . . .	101
E.20	dbx . . . . .	102
E.20.1	Other transformations . . . . .	103
E.21	run . . . . .	103
E.21.1	Other transformations . . . . .	103
E.22	regsub . . . . .	104
E.22.1	Regsub substitution input . . . . .	104
<b>F</b>	<b>Odin Lua Library</b>	<b>107</b>
F.1	GLib . . . . .	107
F.2	File Commands . . . . .	108
F.3	Odin Support Utilities . . . . .	109
F.4	Configuration Support . . . . .	110
F.5	Miscellaneous . . . . .	110

# Chapter 1

## Introduction

### 1.1 Derived Object Managers

A software environment can be simplified if the user's attention is focused on the information provided by the environment rather than the tools that create this information. A *derived object manager* provides this focus by automating the process of tool invocation. A derived object manager responds to a request for a piece of computed information, or *derived object*, by invoking the minimal number of tools necessary to produce that object. If previously computed objects are automatically stored by the object manager for later re-use, significant improvements in response time can be achieved.

In an extensible environment, the kinds of information potentially provided are extended by adding new tools that manipulate and generate new kinds of information. A few examples of the kind of information in a software environment are program source text, test data, modification histories, attributed syntax trees, compiled object code, data flow analysis, and results of symbolic execution.

A derived object manager must provide a language with which a user or a tool can name any desired object, and a specification language for describing the kinds of objects to be managed and the tools that produce them. In the Odin system, derived objects are named by *odin-expressions*, and the specification language is an extended production system called a *derivation graph* in which each tool is described by a single declaration.

### 1.2 Odin Objects

Each Odin object is either a *file*, a *string*, or a *list* (an ordered sequence of objects). An example of a file is a file containing source code, an executable binary file, or an output file from a test run. An example of a string is a command flag. An example of a list is the source files of a program or the arguments to a command.

#### 1.2.1 Source and Derived Objects

Source objects are files that can be created or modified directly by the user. They can be regular files, directories, or symbolic links. Source objects cannot be automatically recreated by the Odin system, but are the basic building blocks from which the Odin system creates all other objects. Every source object is given a type by the Odin system based on its host filename, and this type determines what derived objects can be produced from the source object.

Derived objects are objects that can be produced from source objects and other derived objects through the invocation of one or more tools. Tools are invoked only as needed to create a specified derived object. The Odin system automatically saves objects from previous requests, so a given object might already exist and therefore be available immediately. Derived objects are created and modified only by the Odin system

itself, not by users. Examples of objects that can be derived from source code are a cross reference listing, executable binary code, or the list of files referenced by `#include` statements.

### 1.2.2 The Derived Object Cache

In the Odin system, all derived objects are stored in a directory called the *derived object cache*, or simply the *cache*. The cache also contains the database that stores the *depends* relationship between the output and input files of a tool run, and the *contains* relationship between a list and its elements.

All host files other than those in the cache are source objects. A derived object can be copied from the cache into a source directory, but this does *not* make the file in the source directory a derived object – it just creates a source object that happens to have the same contents as the derived object.

The default location for the cache is a directory named `.ODIN` in the user's home directory. A non-default cache location can be specified in the `$ODIN` environment variable, or with the `-c` option to the `odin` host-command. The main reasons for specifying a non-default location for the cache are to share a common cache with other users, or to locate the cache on a local disk for more efficient access to derived files.

The default name for a file in the cache is `label.id.type`, where `label` and `id` are the results of `:label` and `:id` derivations (see Section B.4.2), and where `type` is the type of the file. If this filename is too long for your operating system, set the environment variable `$ODIN_SHORTNAMES` and a file in the cache will be named simply `id.type`.

The default inter-process communication mechanism for the `odin` cache manager process is TCP/IP. If TCP/IP is not available, set the environment variable `$ODIN_LOCALIPC` and Unix domain sockets will be used instead.

## 1.3 Why Not Just Use Make?

The most ubiquitous build manager is Stu Feldman's Make program[3], with its many descendents[2][6][5][4]. Although ideal for small projects, Make has two major problems when used for large or complex projects: inaccurate dependency information and poor support for variant builds (such as builds for different architectures or builds with different levels of compiler optimization). Inaccurate dependency information leads to inefficient builds when unnecessary recomputations are performed and, more seriously, incorrect builds when necessary recomputations are not performed. Poor support for variant builds consists primarily of problems with specifying the variants and with storing the intermediate files for the variants. An indication of the difficulty of specifying a complex build variant is that often the only way to find out what a given build description means is to run Make and see what kind of commands it executes.

### 1.3.1 Dependency Information

Inaccurate dependency information results from a variety of causes. If the user maintains the dependency information by hand, it is virtually guaranteed to be incorrect. A "make-depend" tool can automate the production of dependency information, but because of the expense of running such a tool on an entire system, it is usually not automatically invoked before each build, again leading to builds based on incorrect dependency information. If a separate dependency file is created for each source file to allow incremental dependency computation, it becomes feasible to automatically compute dependency information before each build. Unfortunately, the source directories then become cluttered with dependency information files, and the overhead of opening and reading all the dependency information files becomes excessive.

In Odin, tools that automatically gather dependency information are as easy to describe as standard tools like compilers, and Odin takes care of running them incrementally when they are needed. The results of these tools are stored in a persistent database rather than in the user's Makefiles, so that this information can be retrieved efficiently and does not clutter up source directories.

Other dependency problems stem from Make's use of file modification time ordering to determine whether a file is up-to-date. Network clock inconsistencies and the use of tools that restore files with their original modification dates can result in changed sources files whose modification dates are less than those of derived

files that depend on them. In Odin, this problem is avoided by storing the date stamps of the inputs to a build step in the dependency database, so that any change to the modification time of a file (either earlier or later) triggers a re-build.

The use of file modification time ordering can also result in inefficient builds when a change to a source file does not result in a corresponding change to an intermediate derived file. For example, adding a comment to a source file will usually not change the result of compiling that file, which means that there is no reason to relink the unchanged object code to form a new executable. Odin avoids this inefficiency by re-running a tool only if the *value* of one of its inputs has changed. In particular, this allows Odin to build directly from an RCS `xxx,v` or an SCCS `s.xxx` file, without first checking out a version into a source directory (the check-out operation is just a tool like any other tool). This is not feasible in Make because every time a new version is added to the version control file, it would assume everything derived from *any* version of that file must be recomputed.

### 1.3.2 Specifying Build Variants

In Make, a derived file variant is specified by changing the values of the appropriate Make variables. For example, the `CC` variable can be changed to build with a different C compiler, and the `CCFLAGS` variable can be changed to build with different C compiler flags. Unfortunately, the scope of a variable is the entire Makefile, while a build variant often involves different variable values for different intermediate files (for example, most of the files are to be compiled with debugging on, except for certain files containing time critical routines which are to be compiled with optimization on). To work around this problem, in Make you have to introduce a special build rule for each intermediate file build variant.

In Odin, a filename can be extended by an arbitrary number of build *parameters*, that are then used by Odin to customize the way that file is processed in any build step that refers to it. For example,

```
prog.c +debug
```

indicates that the debug flag should be set whenever `prog.c` is used in a build step, while

```
io.c +optimize=2
```

indicates that optimization level 2 should be set whenever `io.c` is used in a build step. If the parameterized filename refers to a list of other files, the parameters are inherited by each of the referenced files. Because of the controlled scoping provided by Odin parameters and the expressive power of Odin implicit build rules, explicit build rules are hardly ever needed in an Odinfile.

### 1.3.3 Storing Derived File Variants

Make usually stores a derived file in the directory containing the source file from which it is derived, since this allows other builds that reference the source file to reuse that derived file. The main disadvantage of this approach is that only one variant of a given derived file can exist at a given time. The build manager can easily mistake which version is currently in the directory, leading to an incorrect build. Other disadvantages are that a user must have write permission to the source directories in order to re-build derived files, a user can mistakenly delete a source file thinking it is a derived file, and a user can mistakenly edit a derived file thinking it is a source file.

The usual approach to this problem is to have the user explicitly allocate different directories to contain the results of different variant builds. Unfortunately, this makes it difficult for build sharing to take place, and unless the version of Make has been extended to write out the variable values used for the builds in a given directory, it is easy to get incorrect builds if different variable values are used for builds in the same directory.

Another approach to this problem used in ClearMake[1] consists of introducing a proprietary *multi-version file system*. Every file opened for reading during the build and every variable value is tracked by the system, and if the same set of files with the same variables are present in a later build request, the previously computed results of the build are re-used. Even this approach, though, requires that the user explicitly manage variant derived file locations if concurrent variant builds of files local to a view are desired.



The alternative approach used by Odin that does not require replacing the native Unix file system is to use a derived file cache. In effect, the file namespace is extended to provide names for every derived file variant. For example,

```
/usr/src/prog.c :exe
```

is the executable computed by compiling and linking `/usr/src/prog.c`,

```
/usr/src/prog.c +debug :exe
```

is the executable with debugging information, and

```
/usr/src/prog.c +optimize=2 :exe
```

is the executable compiled with level-2 optimization. These extended names are parsed by the Odin interpreter, which either maps the name to a source file or to the correct derived file in the derived file cache.

### **1.3.4 Parallel Builds**

Odin provides concurrent parallel builds on local and remote hosts. You just specify a list of build hosts and the maximum number of parallel builds.

### **1.3.5 Help**

Odin provides a powerful help system that uses its knowledge of the currently installed set of implicit rules (including those added by the local site administrator and an individual user). In particular, it can tell you what kinds of file extensions are currently recognized, what kinds of files can be derived from a specific type of file, and what parameters can be specified to produce a variant of a given derived file.

### **1.3.6 Errors**

Odin maintains a persistent database of all error and warning messages associated with each build step. You can request a summary report of all errors and/or warnings associated with all the build steps of a particular derived object. For derived object variants, any valid error information from other variants is reused from the database rather than recomputed.

### **1.3.7 Operations on Lists**

Enhanced versions of Make each provide their own idiosyncratic set of operations for manipulating filenames and lists of filenames. In Odin, these operations are performed by arbitrary Unix tools – this is feasible because Odin caches the results of these tools in the dependency database.

### **1.3.8 Editing while Building**

You can edit files that are currently being used for one or more builds, and Odin will automatically recognize the changes and redo the parts of the builds that are affected by the changes, before letting the builds complete.

### **1.3.9 Recursive Implicit Rules**

Odin allows you to recursively chain implicit rules, so for example, you can specify that the files included by a file are the files directly included by the file plus the files included by the files included by the file. Even enhanced versions of Make that do support chaining of implicit rules do not allow recursive chaining.

### 1.3.10 Circular Dependencies

Odin allows a file to depend on itself. This may seem unreasonable until you start writing tools that generate code, and you find that it would be very useful to have the tool generate part of itself. In this situation, Odin will terminate the computation when the tool produces a file that is identical to the one used as input to the computation (i.e., when the computation reaches a *fixed-point*).

### 1.3.11 Efficient Dependency Computation

In very large systems, even when a desired derived file is up-to-date, simply `fstat()`'ing all the source files upon which it depends can take Make several minutes. Since Odin can broadcast file change information in its dependency database, dependency computation can be made proportional to the number of file dependencies that have *changed* rather than the total number of file dependencies.

### 1.3.12 Tool Dependencies

Build steps automatically depend on the tools and scripts that produce them, so if you change a tool, Odin will recompute anything produced by that tool the next time you request a build. If the tool change did not affect a particular run of that tool, the recomputation is short-circuited, and the rest of the derived files that depend on that output of the tool are marked as still being valid.

### 1.3.13 Availability

Finally, Odin is portable, free, and available in source form. It can be retrieved via anonymous ftp from the compressed tar formatted file `distrib/odin/odin.tar.Z` at `ftp.cs.colorado.edu` (128.138.243.151).

## Chapter 2

# The Odinfile

### 2.1 Targets

A build *target* in a directory is a source file whose value is computed by the derived object manager. Just as Make uses a file named **Makefile** to specify targets for a directory, Odin uses a file named **Odinfile**. When the host-command:

```
odin prog
```

is executed, Odin checks to see if **prog** is a target in **./Odinfile**. If it is a target, but is not up-to-date, Odin invokes whatever tools are necessary to bring it up-to-date. Odin automatically looks in all directories that contain the input files for these tools, and if there is an **Odinfile** that contains a target for an input file, Odin ensures that the input file is up-to-date before using it.

A target definition in an **Odinfile** consists of a filename followed by two equal-signs and an *odin-expression*. For example, the **Odinfile** entry:

```
prog == prog.c +debug :exe
```

declares that the value of the source file **prog** should always be equal to the value of the file specified by the odin-expression **prog.c+debug:exe**.

### 2.2 Odin-Expressions

Just as any source file can be named with a source filename, any derived file can be named with an *odin-expression*. An odin-expression is just like a source filename, except that in addition to the slash operator used in source filenames for selecting elements of directories, there is a plus-sign operator for adding parameters to an object and a colon operator for selecting a derivation from an object. For example, in the odin-expression:

```
src/prog.c+debug:exe
```

**prog.c** is selected from the **src** directory, the parameter **+debug** is associated with **src/prog.c**, and the **:exe** (i.e. executable binary) derivation is selected from **src/prog.c+debug**.

#### 2.2.1 Lexical Conventions

Lexically, an odin-expression is composed of a sequence of *identifier* and *operator* tokens, and is terminated by a new-line character. An odin-expression can be continued on multiple lines by escaping each new-line character with a backslash. This backslash (but not the new-line) is deleted before the expression is parsed. Multiple odin-expressions can be specified on the same line by separating them with semi-colon operators.

An identifier token is just a sequence of characters. The following characters:

: + = ( ) / % ; ? \$ < > ! <space> <tab> <new-line> # \ ' ,

must be escaped to be included in an identifier. A single character can be escaped by preceding it with a backslash (e.g. `lost\+found`). A sequence of characters can be escaped by enclosing them in single quote marks (e.g. `'lost+found'`).

Unescaped *whitespace* characters (spaces, tabs, and new-lines) are ignored during parsing except when they separate adjacent identifiers. A comment is indicated by a sharp and is terminated by a new-line character. For example, the odin-expression:

```
# this is a comment
```

is a comment and is equivalent to whitespace.

An odin-expression can be surrounded by parentheses. Parentheses are required for nested odin-expressions (such as values of parameters) or for the empty expression `()` which represents an immutable empty file.

## 2.2.2 Selection Expressions

A selection expression, indicated by the slash operator, selects a file from a directory. The argument to the slash operator is the *key* of the desired file. For example, the odin-expression:

```
src/prog.c
```

selects `prog.c` from the directory `src`. (The careful reader will note that an odin-expression composed entirely of selection expressions bears an uncanny resemblance to a standard Unix filename.) Any special character in the key must be escaped. For example, `prog.c++` must be escaped, as in `prog.c\+\+ or 'prog.c++'`.

Odin splits each key into two parts: a *root* and a *type-name*. The type-name is the longest prefix of the key that matches one of the declared source types prefixes (see Section B.3). If no prefix match is found, the type-name is the longest suffix of the key that matches one of the declared source type suffixes. If no suffix match is found, the type-name is the empty string.

Assuming that `s.` is the only declared source type prefix and `.c` is the only declared source type suffix, then the following relationships hold:

FILENAME	KEY	ROOT	TYPE-NAME
<code>src/prog.c</code>	<code>prog.c</code>	<code>prog</code>	<code>.c</code>
<code>src/s.prog.c</code>	<code>s.prog.c</code>	<code>prog.c</code>	<code>s.</code>
<code>src/prog</code>	<code>prog</code>	<code>prog</code>	

The type-name is used to determine the object type of the file (see Section B.3), and the root is used to determine the object type of generic objects derived from the file (see Section B.4.1).

## 2.2.3 Derivation Expressions

A derivation expression, indicated by the colon operator, is used to specify a derived object. The argument to the colon operator is an *object type* that has been declared in one of the tool packages (see Section B.4). For example, the odin-expression:

```
prog.c :fmt
```

names a formatted version of `prog.c`, and the odin-expression:

```
prog.c :fmt :exe
```

names the result of compiling and linking the formatted version of `prog.c`.

A derived object can be a directory, in which case it is called a *derived directory*. Elements of a derived directory are selected with the same slash operator used to select elements of source directories. For example, if `src/prog.c:output` is a derived directory containing the output files from a test run of `prog.c`, and this directory contains three files named `DATA`, `source.listing`, and `source.errors`, then these three files are named by the odin-expressions:

```
src/prog.c:output/DATA
src/prog.c:output/source.listing
src/prog.c:output/source.errors
```

An element of a derived directory cannot be another directory, although it can be a symbolic link to another directory.

## 2.2.4 Parameterization Expressions

A parameterization expression, indicated by the plus-sign operator, extends an object with additional information that affects the derived objects produced from that object. The argument to the plus-sign operator is a *parameter type* (see Section B.5), optionally followed by an equal-sign operator and a sequence of one or more identifiers and parenthesized object-expressions. The identifiers specify string objects. For example, in the odin-expression:

```
prog.c +lib=(support.c.sm +debug :a) +lib=termcap
```

the value of the first `+lib` parameter is the object `support.c.sm+debug:a` and the value of the second `+lib` parameter is the string `termcap`. An example of an odin-expression with a parameter value that is a sequence is:

```
prog.c +lib=dbm (support.c.sm +debug :a) socket
```

which specifies that the `+lib` parameter has a value that is a sequence consisting of the string `dbm`, the object `support.c.sm+debug:a` and the string `socket`.

If the parameter value is omitted, it is equivalent to specifying the identifier consisting of a single space ' ' as the value. For example, the following two odin-expressions:

```
prog.c +debug
prog.c +debug=' '
```

are equivalent.

The parameter values of a given parameter type in an odin-expression form an ordered set, where the order of the values is the order specified in the odin-expression. Therefore, the odin-expressions:

```
prog.c +lib=dbm +lib=socket :exe
prog.c +lib=socket +lib=dbm :exe
```

are not equivalent. If multiple copies of the same parameter value appear, only the first of the multiple copies is kept. For example, the odin-expressions:

```
prog.c +lib=socket +lib=dbm +lib=socket :exe
prog.c +lib=socket +lib=dbm :exe
```

are equivalent. If a parameter has a value that is a sequence, that value is only considered the same as another identical sequence. For example, the odin-expressions:

```
prog.c +lib=socket dbm +lib=socket :exe
prog.c +lib=socket +lib=dbm :exe
```

are not equivalent, while the odin-expressions:

```
prog.c +lib=socket dbm +lib=socket +lib=socket dbm :exe
prog.c +lib=socket dbm +lib=socket :exe
```

are equivalent.

The parameter value lists of each parameter type are disjoint, therefore, the order of parameters of different types is not significant. For example, the odin-expressions:

```
prog.c +debug +profile :exe
prog.c +profile +debug :exe
```

are equivalent.

Parameters of a list object are inherited by all the elements of the list. For example, if the odin-expression:

```
prog.c.sm :list
```

named a list whose elements were `main.c` and `subrs.c`, then the odin-expression:

```
prog.c.sm :list +debug
```

associates the `+debug` parameter with `main.c` and `subrs.c`.

Parameters are not inherited forward or backward through derivations. Therefore, in the odin-expression:

```
prog.c +debug :o +optimize=2 :exe
```

the `:o` derivation has debugging, but no optimization, while the `:exe` derivation has optimization, but no debugging.

Frequently, several odin-expressions share a common set of parameters. To support this kind of sharing, a parameterization expression can take the form of a plus-sign followed by a parenthesized object-expression. The common set of parameters are then stored in the file named by the object-expression. For example, if the file `my.prms` contained the text:

```
+debug +lib_sp=(/local/lib) +lib=socket +lib=dbm +gnu
```

then the odin-expressions:

```
prog.c +(my.prms) :exe
prog.c +debug +lib_sp=(/local/lib) +lib=socket +lib=dbm +gnu :exe
```

would be equivalent.

### 2.2.5 String Expressions

A string expression, indicated by the equal-sign operator followed by an identifier, is used to specify arguments to tools that are not filenames. For example, the odin-expression:

```
=cxv
```

names the string `cxv`.

## 2.3 Virtual Targets

In addition to targets, an `Odinfile` can contain *virtual targets* that define a set of *virtual files*. A virtual target is like a target, except that the value is not copied into a host file, but is simply associated with the specified virtual filename. Virtual files provide a mechanism for specifying aliases for complicated odin-expressions, and a mechanism for specifying build-support files without cluttering host directories.

A virtual target definition is the same as a target definition (see Section 2.1), except that the filename begins with an unescaped percent-sign. For example, the `Odinfile` entry:

```
%prog == prog.c +debug :exe
```

binds the odin-expression `prog.c+debug:exe` to the virtual filename `prog`.

The syntax for selecting a virtual file from an `Odinfile` is identical to that for selecting a file from a directory, except that a percent-sign is used instead of a slash. For example, the odin-expression:

```
src/Odinfile%prog
```

selects the virtual file `prog` from `src/Odinfile`.

If the directory name preceding the percent-sign is omitted, the virtual file is selected from the `Odinfile` in the current working directory. For example, the odin-expressions:

```
%prog
./Odinfile%prog
```

are equivalent.

Although only a file named `Odinfile` can specify targets, any file can contain virtual targets. For example, if `src/models` were a host file containing the text:

```
%fastprog == prog.c +optimize=2 :exe
```

then the odin-expression:

```
src/models%fastprog
```

obtains the virtual target for `%fastprog` from `src/models`.

## 2.4 Text Targets

The value of a target can be specified directly as lines of text (a *here document*), instead of as an odin-expression. In this case, the value declaration consists of two left-angle-brackets optionally followed by an arbitrary tag identifier. For example, the `Odinfile` entry:

```
prog.c.sm == << END
    main.c
    routines.c
END
```

declares `prog.c.sm` to be a text target. The value of `prog.c.sm` is then a file containing the text:

```
main.c
routines.c
```

If the tag identifier is omitted, the text value ends at the first line containing only whitespace characters. For example, the `Odinfile` entry:

```
prog.c.sm == <<
    main.c
    routines.c
```

is equivalent to the previous definition.

If the tag identifier begins with a new-line character, the terminating tag identifier does not contain the leading new-line character, and the last new-line character in the text value is ignored. For example, the `Odinfile` entry:

```
vowels == << \\
TAG
aeiou
TAG
```

defines the contents of the file `vowels` to be the five characters: "aeiou".

### 2.4.1 Executable Text Targets

Execute permission can be set for a text target by specifying it to be an *executable text target*. A text target is an executable text target if an exclamation-point is specified immediately before the <<. For example, the `Odinfile` entry:

```
%backup == !<<
    backup_name=$1.back
    mv $1 $backup_name
    cp $backup_name $1
```

specifies `%backup` to be an executable text target.

## 2.5 Command Targets

A target is a *command target* if an exclamation-point is specified following the filename. For example, the `Odinfile` entry:

```
prog ! == prog.c.sm +debug :exe
```

specifies `prog` to be a command target.

Normally, when a filename is specified as an argument to the `odin` host-command, Odin simply ensures that the value of the filename is up-to-date. If the filename is a command target, Odin also executes the file after it is made up-to-date. For example, with the preceding target definition for `prog`, the host-command:

```
odin prog
```

makes sure `prog` is up-to-date with respect to `prog.c.sm+debug:exe`, and then executes it.

## 2.6 Nested Target Definitions

A file containing additional target definitions can be specified in a *nested target definition*. A nested target definition consists of two equal-signs followed by an `odin`-expression. A common use of nested target definitions is to pre-process the text of the target definitions. For example, the `Odinfile` entries:

```
== %nested :cpp

%nested == << EOF
#ifdef linux
    prog == prog.c +gnu +debug :exe
#else
    prog == prog.c +debug :exe
#endif
EOF
```

specify that the `+gnu` parameter should be used on the `linux` platform.



## Chapter 3

# Getting Started

To get started using Odin, go to a directory containing your favorite "hello world" program.

```
1% cd /u/fred
2% cat hello.c
```

```
main()
{ printf("Hello World.\n"); return 0; }
```

Then create an Odinfile containing the line `hello==hello.c:exe`.

```
3% ls
Odinfile  hello.c
4% cat Odinfile
```

```
hello == hello.c :exe
```

You are now ready to build `hello`. Just say:

```
5% odin hello
```

By default, Odin issues a message every time it invokes a tool, for example:

```
scan_for_includes hello.c
cc -c hello.c
cc hello.o
** Copying up-to-date value into /u/fred/hello
```

The first tool scanned the source file for includes; the second generated object code; the third generated an executable; and the fourth copied the executable into the file named `/u/fred/hello`.

You have now built your first executable with Odin. Now to test it:

```
%6 hello
Hello World.
```

### 3.1 Systems with Several Files

Interesting systems will usually consist of more than one file. In Odin, you list the source files that make up the system in a file called a *system model*. The files named in `#include` statements should not be listed in the system model – Odin will discover them by running a tool that scans the source files looking for `#include` statements.

Modify `hello.c` so that it calls a `print()` routine in the file `print.c`.

```

7% ls
Odinfile  hello    hello.c  print.c
8% cat hello.c

main()
{ print("Hello "); print("World.\n"); return 0; }

9% cat print.c

print(s) char *s;
{ printf(s); }

```

Now you need to create a system model, **hello.c.sm**, for the new **hello** system. The system model can be a regular file, but it can equally well be specified as a *virtual file* in the **Odinfile**.

```

10% cat Odinfile

hello == %hello.c.sm :exe

%hello.c.sm == << EOF
    hello.c; print.c
EOF

```

The new version of **hello** can now be built:

```

11% odin hello

```

You will notice although the file **hello** has changed, no new files have appeared in the source directory.

```

12% ls
Odinfile  hello    hello.c  print.c

```

This is because all derived files are stored in the derived file cache directory (**\$HOME/.ODIN** by default – a non-default location can be specified in the **\$ODIN** environment variable).

But the derived files are available for re-use. For example, if you to add a comment to the **print()** routine:

```

13% cat print.c

/* print the character sequence s to standard output */
print(s) char *s;
{ printf(s); }

```

and then ask Odin for **hello** again:

```

14% odin hello

```

you will notice a couple of things. First, the previous results of scanning **hello.c** for includes and compiling it are both re-used, as you would expect. Somewhat more surprising may be that Odin notices that the compilation of the modified **print.c** produced the same object code. Odin then skips the link phase since the old executable is still valid.

## 3.2 Files in Other Directories

The files in a system model can be in any directory, using either absolute or relative pathnames. Odin knows that **/u/fred/print.c** is potentially different from **/u/jane/print.c** so will not be confused if you switch from one to the other in your system model.

```

15% cat ../jane/print.c

print(s) char *s;
{ printf("%s*", s); }

%16 cat Odinfile

hello == %hello.c.sm :exe

%hello.c.sm == << EOF
    hello.c; ../jane/print.c
EOF

%17 odin hello

```

### 3.3 Recursive Odin Calls

Suppose that `/u/jane/print.c` is modified to include a `y.tab.h` file generated by the `yacc` tool from a file named `print.y`. The fact that `y.tab.h` is generated from `print.y` is indicated in `/u/jane/Odinfile`.

```

18% ls ../jane
Odinfile  print.c    print.y
19% cat ../jane/print.c

#include "y.tab.h"
print(s) char *s;
{ printf("%s*", s); }

20% cat ../jane/Odinfile

y.tab.h == print.y :h

```

If you were using `make`, you would have to somehow get `make ../jane` invoked before you tried to `make hello`. In Odin, all these dependencies are computed for you, so a simple `odin hello` is sufficient.

```

21% odin hello

```

### 3.4 Debugging Your Program

Odin uses parameters instead of tool flags to modify the behavior of tools. In particular, `+debug` adds debugging information. Either the entry for `hello` in the `Odinfile` can be modified, or as is done below, a second entry such as `test-hello` can be added.

```

22% cat Odinfile

hello == %hello.c.sm :exe

test-hello == %hello.c.sm +debug :exe

%hello.c.sm == << EOF
    hello.c; print.c
EOF

```

At this point, you can build `test-hello` and run the debugger.

```
23% odin test-hello
```

```
24% dbx test-hello
```

These two steps can be combined by adding an executable `%dbx` entry into the `Odinfile`.

```
25% cat Odinfile
```

```
hello == %hello.c.sm :exe
```

```
test-hello == %hello.c.sm +debug :exe
```

```
%dbx ! == %hello.c.sm +debug :dbx
```

```
%hello.c.sm == << EOF
```

```
    hello.c; print.c
```

```
EOF
```

Then the host-command:

```
26% odin %dbx
```

would build an up-to-date executable from `%hello.c.sm`, build a script for executing `dbx` on this executable, and then because `%dbx` is an executable target, execute the script.

## Chapter 4

# Odin-Commands

In addition to being used as a means for making source files up-to-date, the Odin system can also be used as a command interpreter for *odin-commands*. Odin-commands are either given as arguments to the `odin` host-command, or, if the `odin` host-command is invoked with no arguments, *odin-commands* are read interactively from standard input.

When Odin is used as an interactive command interpreter, it first prints out a banner indicating the version of Odin you are using, and then an arrow indicating it is ready to accept input. You exit the Odin interpreter by typing an end-of-file, commonly the control-D character.

```
1% odin
Odin Version 1.18.0
-> ^D
2%
```

When the logging level is high enough, executing *odin-commands* from a script will print the *odin-command* before executing it. However, some command scripts are expected to execute silently regardless of log level. In this case, each *odin-command* to be silently executed may be prefixed with the at-character (`@`).

There are four kinds of *odin-commands*: *query odin-commands*, *copy odin-commands*, *execute odin-commands*, and *variable assignment odin-commands*. If the Odin interpreter encounters a question-mark in an *odin-command*, it generates a help message describing what could have been typed at the location of the question-mark.

### 4.1 Query Odin-Commands

A *query odin-command* consists simply of an *odin-expression*. A query *odin-command* causes the Odin object named by the *odin-expression* to be brought up to date, and reports information concerning the status of the object (such as error and warning messages produced by tool steps that were run to produce the object).

```
-> print.c:exe
--- </u/fred/print.c :exe> generated errors ---
ld: Undefined symbol
    _main
cc failed
->
```

The level of detail of this information is controlled by the value of the `ErrLevel` and `WarnLevel` variables (see Section 4.4.5).

### 4.1.1 Status of Odin Objects

Associated with each Odin object is a status level, where the status level is one of **OK**, **WARNING**, **ERROR**, **CIRCULAR**, **NOFILE**, and **ABORT**. **OK** is the maximum status level and **ABORT** the minimum.

The status of a given derived object depends on the results of the tools that produced that object. If any tool generated warning messages, the status level of the given object is at most **WARNING**. If any tool generated error messages, the status level of the given object is at most **ERROR**. If an object that was needed to create the given object is the object itself, the status level of the given object is at most **CIRCULAR**. If any object that was needed to generate the given object did not exist, the status level of the given object is at most **NOFILE**. If any object that was needed to generate the given object had status level **ERROR** or less, then the status level of the given object is set to be **ABORT**.

The status of a source object is **NOFILE** if the host file does not exist, the status of the value of its target value (see Section 2.1) if it is a target, and otherwise **OK**.

### 4.1.2 Error and Warning Messages

The warning or error messages produced by all tool invocations are saved by the Odin system. The difference between an error and a warning is that an error prevents the tool from generating its output, while a warning indicates that although output was generated, it might be faulty. An example of an error message from a loader is:

```
Unsatisfied external reference: "proc1".
```

An example of a warning message from a loader is:

```
Multiply defined external: "proc2", first copy loaded.
```

A text file containing a summary of all error messages for an object can be obtained by applying the `:err` derivation to the object. For example, the file named by the odin-expression:

```
prog.c :exe :err
```

contains a summary of all error messages produced by any tool used in the generation of the `prog.c:exe` object. The `:warn` derivation produces a text file containing both warning and error messages for an object.

## 4.2 Copy Odin-Commands

A *copy odin-command* copies the contents of a specified Odin object into another object (the destination object). The copy is performed only if the status level of the specified object is no lower than **WARNING**. The destination object must be a source object since only source objects can be directly modified by a user.

There are two forms of the copy odin-command: *copy-to*, indicated by a right-angle-bracket, and *copy-from*, indicated by a left-angle-bracket. Examples of these two odin-commands are:

```
-> prog.c +debug :exe > prog
-> prog < prog.c +debug :exe
```

If the destination object is a directory, the *label* of the specified object is used to name the new copy. The label of a source file is the last component of the pathname of the source file. The label of a derived object is *source-label.type-name* where *type-name* is the name of the output type of the tool that produced it (see Section B.7). and where *source-label* is the label of the source file from which it is derived. For example, the label of `/usr/src/prog.c` is `prog.c` and the label of `/usr/src/prog.c:exe` is `prog.c.exe`.

If a list is copied into a directory, each element of the list is copied individually into the directory.

### 4.2.1 Copying to Standard Output

If the destination object is omitted from a copy-to odin-command, the specified object is displayed on the current standard output device. For example, the odin-command:

```
-> prog.c >
```

displays the file named `prog.c`.

### 4.2.2 Editing with the Copy Odin-Command

If only the destination object is specified in a copy-from odin-command, the specified object is given to the host-system editor indicated by the `$EDITOR` environment variable, with the `vi` editor the default. For example, if the value of the `$EDITOR` variable is `emacs`, then the odin-command:

```
-> prog.c <
```

invokes the `emacs` editor on the file `prog.c`.

## 4.3 Execute Odin-Commands

An *execute odin-command* consists of an odin-expression followed by an exclamation-point and a host-command line, where either the odin-expression or the host-command can be omitted. Examples of execute odin-commands (with output omitted) are:

```
-> prog.c +debug :stdout !more -s
-> ! ls *.c
-> prog.c :exe !
```

The result of the execute odin-command is to make the Odin object up-to-date, append its filename to the host-command line, and give the resulting extended host-command line to the host system for execution.

The exclamation-point has the special lexical property that if the first non-whitespace character following it is not a colon, a semi-colon, or an equal-sign, then the rest of the line is treated as a single escaped sequence of characters. This avoids the confusion resulting from interactions between host-command and Odin character escape conventions. A leading colon, equal-sign, or whitespace character can be included in the escaped sequence of characters by preceding it with a backslash.

If the host-command begins with the characters `lua!`, the text following that prefix is executed as Lua code. Otherwise, it is executed using the system command interpreter. See Appendix F for a list of additional functions provided by the Lua interpreter. Each Lua host-command is executed in a separate interpreter, but in the same process space as the Odin server. Care must be taken to not invoke C shared libraries which might disrupt Odin.

If the host-command is omitted, the Odin object itself is executed. If execute permission is set for the Odin object, it is given to the host operating system for execution; otherwise, the Odin object is assumed to contain odin-commands that are to be executed by the Odin interpreter.

## 4.4 Variable Assignment Odin-Commands

The behavior of the Odin interpreter can be modified by changing the value of an *Odin variable*. The functions affected by Odin variables are the current working directory, the distributed parallel build facility, the help facility, the error and log facility, the file change notification facility, and the maximum total file system space used by derived objects.

A variable assignment odin-command consists of the name of an Odin variable followed by an equal-sign operator and an odin-expression. For example, the odin-commands:

```
-> dir = ../src
-> warnlevel = 4
```

assign the value `../src` to the `Dir` variable and the value `4` to the `WarnLevel` variable (Odin variable names are case-insensitive).

If the value is omitted from a variable assignment odin-command, Odin displays the current value of the specified variable. For example, after the preceding odin-commands, the odin-command:

```
-> warnlevel =
4
```

can be used to find out the current value of `WarnLevel`.

The Odin variables and their default values are:

```
Dir ..... odin_invocation_directory
MaxBuilds ..... 2
BuildHosts .... LOCAL : LOCAL
Size ..... 0
KeepGoing ..... yes
History ..... yes
LogLevel ..... 2
ErrLevel ..... 3
WarnLevel ..... 2
HelpLevel ..... 1
VerifyLevel ... 2
```

An initial value for an Odin variable can be specified in an environment variable whose name is the the Odin variable name in capital letters preceded by the string `ODIN`. For example, the initial value for `MaxBuilds` is specified in the `ODINMAXBUILDS` environment variable.

#### 4.4.1 Dir

The current working directory can be changed by assigning a new value to the `Dir` variable. The value of the current working directory is especially significant for Odin, since Odin identifies source objects by their absolute pathname, and the current working directory provides the absolute pathname for all relative names given to the Odin interpreter.

#### 4.4.2 BuildHosts, MaxBuilds

The `BuildHosts` variable specifies the list of hosts that are used to execute the tools that generate the derived objects. A tool is executed on the first entry in the `BuildHosts` list that does not have a currently executing tool. The name `LOCAL` refers to the local host. The `MaxBuilds` variable specifies the maximum number of tools to execute in parallel.

The hosts in `BuildHosts` must have the same machine architecture and file namespace as the local host. If builds on non-equivalent machines are desired, this can be achieved by extending the tool packages (see Section B.1).

#### 4.4.3 KeepGoing

When a build step reports errors, Odin will continue with build steps that do not depend on the failed build step. Setting the value of the `KeepGoing` variable to `no` will cause Odin to terminate the build when any build step reports an error (similar to the default behavior of `Make`).



#### 4.4.4 History

The **History** variable specifies whether emacs-like history and command line editing is supported by Odin when it is used as an interactive command interpreter.

#### 4.4.5 ErrLevel, WarnLevel, LogLevel

When an odin-command is executed, Odin indicates any errors or warnings associated with the odin-expressions specified in that odin-command. The **ErrLevel** and **WarnLevel** variables specify how detailed this report is. In particular, the user can choose whether to see final status information, to see messages incrementally as they are produced by tools steps, or to see a summary of all relevant messages (including those from previously cached tool steps).

Odin can also produce a variety of information about the activities it is performing, such as a brief description of each tool that is invoked to satisfy a given request. The **LogLevel** variable specifies how detailed these messages are.

#### 4.4.6 HelpLevel

The **HelpLevel** variable specifies what degree of detail is provided when the user asks for a list of possible file or parameter types (see Section 4.5). Normally, only commonly used types are described, but the **HelpLevel** can be increased to have all possible types described.

#### 4.4.7 VerifyLevel

By default, Odin checks the modification dates of all relevant source files at the beginning of an Odin session and before each interactive odin-command. If all file modifications during the session are performed through copy odin-commands or through an editor that has been upgraded to send a **filename!:test** odin-command to Odin whenever **filename** is modified, the **VerifyLevel** variable can be set to 1 and only the check at the beginning of the session is performed. If all file modifications since the last session have been performed in the above manner, **VerifyLevel** can be set to 0 and the initial check is avoided as well.

#### 4.4.8 Size

The value of the **Size** variable indicates how much disk space (in kilobytes) is currently being used by derived files.

#### 4.4.9 Environment Variables

Environment variables can be used in odin-commands. For example, if the environment variable **\$HOME** has the value **/u/geoff**, then the two odin-commands:

```
-> $HOME/prog.c :exe
-> /u/geoff/prog.c :exe
```

are equivalent. The value of an environment variable can be quoted by immediately preceding it with a quoted identifier. For example, if the value of **\$PATH** is **/bin:/usr/bin**, then the two odin-commands:

```
-> prog.c +path='/etc:'$PATH
-> prog.c +path='/etc:/bin:/usr/bin'
```

are equivalent.

The environment variable **\$ODINCACHE** is set by Odin to be the location of the cache directory (see Section 1.2.2).

An environment variable is given a new value with a variable assignment odin-command of the form:

```
Variable = !Value
```

For example, the odin-command:

```
-> HOME = !/u/clemm
```

sets the value of the environment variable `$HOME` to be `/u/clemm`.

Although any environment variable can be used in odin-commands, only environment variables declared in a tool package (see Section B.6) can be used in `Odinfile` target definitions (see Section 2.1) The value of a declared environment variable is obtained from the environment at the time the current cache was created or reset. If the variable was not set in that environment, the default value specified in the derivation graph declaration of the variable is used.

The expressions `~` and `~name` are treated as if they were environment variables, bound respectively to the login directory of the current user and the login directory of the user with login `name`.

## 4.5 Help

A simple context-sensitive help facility is provided to describe the syntax of odin-commands and the currently available object types and parameter types. If a user types a question-mark anywhere in an odin-command, Odin provides a description of what could appear at that location in the odin-command. For example, the following odin-command provides a syntax summary:

```
-> ?
Control-D
OdinExpr
  HostFile
    + ParameterType
    + ParameterType = Identifier
    + ParameterType = (OdinExpr)
    : FileType
    / Key
    % Key
OdinExpr > OdinExpr
OdinExpr >
OdinExpr < OdinExpr
OdinExpr <
OdinExpr !HostCommand
OdinExpr !
OdinExpr !:test
!HostCommand
!:test
Variable =
Variable = Value
Variable = !Value
```

### 4.5.1 Source Type Help

If a list of the declared source object type-names is desired, a question-mark can be put in place of the extension for a file. For example, the odin-command:

```
-> prog?
Possible Base Types :
.c ..... C source code
.f ..... Fortran77 source code
.c.sm ..... C system model containing a list of filenames
,v ..... RCS version control file
```

describes the currently recognized type suffixes, where the message contents is derived from the installed tool packages (see Section B.3).

### 4.5.2 Derivation Help

If a list of possible derivations is desired, a question-mark can be put in place of the derivation name, and the Odin system responds with a list of the possible object types that can appear at that position. For example, the odin-command:

```
-> prog.c :fmt : ?
Possible Derivations from an Object of Type "fmt":
o ..... object code
exe ..... executable binary
fmt ..... formatted version
xref ..... cross reference listing
```

states that all of the following are acceptable odin-expressions:

```
prog.c :fmt :o
prog.c :fmt :exe
prog.c :fmt :fmt
prog.c :fmt :xref
```

### 4.5.3 Parameterization Help

If a list of the possible parameter types is desired, a question-mark can be put in place of the parameter, and the Odin system responds with a list of the possible parameter types that can appear at that position. For example, the odin-command:

```
-> prog.c :fmt + ?
Possible Parameters:
lib ..... object code library
debug ..... flag to generate debugging information
```

states that any of the following are acceptable odin-expressions:

```
prog.c :fmt +lib=(/usr/lib/network.a)
prog.c :fmt +debug
prog.c :fmt +debug +lib=(/usr/lib/network.a)
```

A more exact form of parameter help can be specified by indicating which derivation you intend to apply to the parameterized object. For example, the odin-command:

```
-> prog.c :fmt + ? :o
Possible Parameters:
debug ..... flag to generate debugging information
```

states that the following is an acceptable odin-expression:

```
prog.c :fmt +debug :o
```

Since the +lib parameter is not relevant to the derivation from :fmt to :o, it is not listed.

#### 4.5.4 Variable Help

A list of the available variable names is generated in response to the `odin-command`,

```
-> ? =  
Dir MaxBuilds BuildHosts Size KeepGoing History  
LogLevel ErrLevel WarnLevel HelpLevel VerifyLevel
```

A description of the possible values that can be assigned to a given variable is generated in response to the `odin-command`:

```
Variable = ?
```

For example, the `odin-command`:

```
-> LogLevel = ?  
0: No log information is generated.  
1: Build commands are echoed.  
2: And Odin commands.  
3: And names of objects with errors.  
4: And names of objects generated by tool scripts.  
5: And names of objects generated by internal tools.  
6: And names of objects deleted.  
7: And names of objects touched by broadcast.
```

describes the possible values of the `LogLevel` variable.

## Appendix A

# Dependency Database Utilities

Odin provides the following utilities for browsing and modifying the dependency database: `:test`, `:redo`, `:inputs`, `:outputs`, `:elements`, `:element-of`, and `:dpath`. These utilities are specified following the exclamation point of an execute odin-command. For example, the odin-command:

```
-> prog.c :o ! :redo
```

would invoke the `:redo` utility with `prog.c:o` as its argument.

Unlike objects named in all other kinds of odin-commands, the object specified as an argument to a database utility is *not* made up-to-date before the database utility is executed. If you wish the database utility to be executed on an up-to-date object, first give the object as an argument to a query odin-command. For example:

```
-> prog.c :o
-> prog.c :o ! :redo
```

### A.1 Testing Host Files for Changes

Normally Odin checks all host files for changes at the beginning of an Odin session and before each interactive odin-command (Odin actually performs these checks lazily, and will `fstat()` a file only when it is needed as input to a tool). It also checks a particular host file for changes after it is used as the destination of a copy odin-command.

Odin's runtime efficiency can be significantly improved by setting the Odin `VerifyLevel` variable to 1 (see Section 4.4.7). In this mode, the user (or a tool such as the user's editor) must notify Odin with the `:test` utility whenever a file changes other than through a copy odin-command.

For example, the odin-command:

```
-> /usr/src/prog.c ! :test
```

causes Odin to check to see if `/usr/src/prog.c` has changed, to determine whether objects derived from this file must be recomputed. If no source is specified for the `:test` utility, i.e. the odin-command is just:

```
-> ! :test
```

then all source files known to Odin are checked.

The Odin distribution includes an `odin.el` file that extends the gnu-emacs editor to issue the appropriate `filename! :test` message whenever a file is saved.

## A.2 Redoing a Particular Derivation Step

The `:redo` utility allows the user to tell Odin that a particular step in a derivation should be recomputed when it is next requested, even if that step has already been computed and inputs to that step have not changed. For example, the odin-commands:

```
-> prog.c +debug :o ! :redo
-> prog.c +debug :o
```

causes Odin to redo the `prog.c+debug:o` step. This utility is useful when a transient error in a derivation step occurs, but the tool that implements the derivation step did not recognize it as a transient error.

## A.3 Browsing the Dependency Database

The `:inputs` utility displays the direct inputs to the specified derived object. The `:outputs` utility displays the direct outputs of the specified object. The `:elements` utility displays the elements of the specified list. The `:element-of` utility displays the lists that contain the specified object as an element. For example, the odin-command:

```
-> prog.c +debug :exe !: inputs
```

displays the input objects used to compute the derived object `prog.c+debug:exe`.

The `:dpath` utility displays the dependency path from one object to another (if there is one). Normally, `:dpath` is applied to an object with a trailing `+depend` parameter. For example:

```
prog.c :exe +depend=(/usr/include/stdio.h) ! :dpath
```

would display the dependency path from `/usr/include/stdio.h` to `prog.c:exe`.

When browsing the dependency database, you may encounter certain internal object types that you have not explicitly declared. For example, if you have declared a tool with multiple outputs, there will be an object whose type ends with `*composite` which represents the set of output objects from a single tool run. Another example is a `:abstract` object which is created for each `:PIPE` and `:GENERIC` object.

# Appendix B

## Tool Package Libraries

In the Odin system, all information about tools, object types, and parameter types is specified in special directories called *tool package libraries*. When a cache (see Section 1.2.2) is created, *tool packages* from the tool package libraries are installed in the cache. A tool package is just a sub-directory of a tool package library, where the name of the package is the name of the sub-directory. The list of tool packages and the order in which they should be loaded is specified in a file named `PKGLST` (one package name per line) in the tool package library directory.

A default tool package library that is provided with the Odin distribution is installed into every cache. This default library is commonly located in `/usr/local/lib/Odin`. Do not add, delete, or modify packages from this library. This ensures that installation of newer versions of Odin will not delete or overwrite any local modifications.

A user can have the packages from additional libraries installed by specifying these libraries (separated by colons) in the `$ODINPATH` environment variable. For example, if a user sets the `$ODINPATH` variable to have the value `/project/lib/Odin:/home/geoff/my-pkg-lib`, then the packages from `/project/lib/Odin`, from `/home/geoff/my-pkg-lib`, and from the default library are installed into every cache the user creates. If a package with a given name occurs in several package libraries, the package from the library that occurs earliest in `$ODINPATH` is used.

The entries in `$ODINPATH` can also be individual tool packages rather than tool package libraries, in which case those packages are added to the list of packages to be installed into the cache.

### B.1 Tool Packages

A tool package is a directory that contains a *derivation graph* file and any support files that implement the tools declared in the derivation graph. If `pkgx` is the name of the tool package, then `pkgx.dg` is the name of the derivation graph file.

The derivation graph is the mechanism for declaring object types, parameter types, environment variables, and tools for the tool package. It is described in detail in Section B.2.

#### B.1.1 Versioned Tool Packages

By default, a tool package is symbolically linked into each cache in which it is installed. Since the Odin system maintains the dependency of each build step on the package scripts used by that build step, a change to a build script will cause the recomputation of all build steps that use that script. In case the script change is an enhancement rather than a bug fix, it is often preferable to allow each user to decide when to upgrade his cache to use the new packages.

To provide this functionality, a tool package library can contain a file named `LIBVER` in addition to the tool package subdirectories. The `LIBVER` file contains text of the form `PKGVER.i`, where `i` is an integer. Each tool package then contains a file named `PKGVER.i` and a set of sub-directories, with one sub-directory

for each version of the tool package. If changes to a package are made in the form of a new version, these changes are not seen by a user until he resets his cache with the `-R` option to the `odin` host-command.

The tool package version installed into a cache is the one specified by the `PKGVER.i` file. For example, if `LIBVER` contained the text `PKGVER.7`, and if a tool package directory named `pkgx` contains three tool version directories named `pkgx/1.0`, `pkgx/1.1-beta`, and `pkgx/1.2`, and if the contents of the `pkgx/PKGVER.7` file is the text `1.1-beta`, then any new cache that installs `pkgx` installs the `1.1-beta` version of `pkgx`.

When a new version of the library is installed, the `i` in `PKGVER.i` is incremented, e.g. to `PKGVER.8`, and a `PKGVER.8` file is created in each of the package subdirectories. The library can be reverted back to an earlier version of the packages by editing the `LIBVER` file.

## B.2 Derivation Graphs

A derivation graph consists of a sequence of source, object type, parameter type, environment variable, and tool declarations. A source declaration associates a filename pattern with an object type. An object type declaration associates an object type with a set of supertypes. A parameter type declaration associates a parameter type with an object type for the parameter values. An environment variable declaration specifies a default value for an environment variable. A tool declaration specifies the inputs and outputs of the tool, where the inputs are object types, parameter types, `odin`-expressions, and identifiers, and where the outputs are object types.

In the derivation graph, the following characters are special:

```
<whitespace> # \ ' : + = ( ) / % ; ? * < > @ & $
```

The quoting and commenting conventions for the derivation graph are the same as those for `odin`-expressions (see Section 2.2).

## B.3 Source Declarations

A source declaration specifies the type for a host file based on the filename of the host file. It consists of a *filename expression*, a right arrow (composed from an equal-sign and a right-angle-bracket), and the name of a declared object type. A filename expression consists either of an identifier followed by an asterisk (i.e. all filenames that begin with that identifier), an asterisk followed by an identifier (i.e. all filenames that end with that identifier), or just an asterisk (i.e. all filenames). For example, the derivation graph entries:

```
*.c      => :c;
*.c.c    => :cplusplus;
s.*      => :sccs;
*        => :FILE;
```

declare that the object type of a filename ending with `.c` is `:c`, the object type of a filename ending with `.c.c` is `:cplusplus`, the object type of a filename beginning with `s.` is `:sccs`, and the object type of any filename is `:FILE`.

The following disambiguation rules are used to ensure that every host file has a single object type: if a host filename matches both a prefix filename expression and a suffix filename expression, then the host file is an instance of the prefix filename expression; if a host filename matches two different prefix filename expressions or two different suffix filename expressions, then the host file is an instance of the longer filename expression. For example, with the previous source declarations, the following object types would be assigned:

FILENAME	OBJECT-TYPE
<code>src/test.c</code>	<code>c</code>
<code>src/test.c.c</code>	<code>cplusplus</code>
<code>src/s.test.c.c</code>	<code>sccs</code>
<code>src/test</code>	<code>FILE</code>



## B.4 Object Type Declarations

An object type declaration consists of the object type being declared, a *help identifier*, a right arrow, and the *direct supertypes* of the type being declared. For example, the derivation graph entries:

```
:c 'C source code'? => :source;
:source 'source code' => :FILE;
:fmt 'formatted source code' => :FILE;
```

declare that the direct supertype of `:c` is `:source`, the direct supertype of `:source` is `:FILE`, and the direct supertype of `:fmt` is `:FILE`.

The *help identifier* is used by the Odin help system to generate messages about the installed types (see Section 4.5). The help identifier is followed by a question-mark to indicate that it should be displayed in help messages. The user can explicitly request to see all help identifiers by increasing the value of the Odin `HelpLevel` variable (see Section 4.4.6).

An object type can be declared with multiple direct supertypes, either in a single declaration or in separate declarations. For example, the derivation graph entries:

```
:fmt.c 'formatted version of C code' => :c;
:fmt.c 'formatted version of C code' => :fmt;
```

and

```
:fmt.c 'formatted version of C code' => :c :fmt;
```

are equivalent.

The *supertypes* of an object type are the transitive closure of the direct supertypes (i.e. the supertypes are the direct supertypes and all supertypes of the direct supertypes). For example, from the preceding object type declarations, the supertypes of `:fmt.c` are `:c`, `:fmt`, `:source`, and `:FILE`.

The supertypes of an object type help determine what objects can be derived from an object of that type. In particular, if the type of the object `obj` is `:o`, then the odin-expression `obj:deriv` is valid if `:deriv` is a supertype of `:o`, if `:deriv` is the output of a tool whose inputs can be derived from `:o`, or if `:deriv` is a supertype of an object that can be derived from `:o`.

In some rare circumstances, a tool produces an object of a specified type, but this object should not be used to provide this type as an intermediate step in derivations. In this case, the output type of the tool should be declared to be a **base subtype** of the specified type. This is indicated by placing angle brackets around the supertype. For example, the derivation graph entry:

```
:dummy_main.c 'dummy main program for a C executable' => <:c>;
```

declares that the direct base supertype of `:dummy_main.c` is `:c`. This states that derivations appropriate to objects of type `:c` can be applied to objects of type `:dummy_main.c`, but that a derived object of type `:dummy_main.c` cannot be used for a derivation that must produce an object of type `:c`.

### B.4.1 Built-In Supertypes

The following object type declarations are built into the Odin system:

```
:OBJECT 'any object'           => :OBJECT;
:FILE 'a file'                 => :OBJECT;
:LIST 'a list'                 => :OBJECT;
:VOID 'an object with no value' => :OBJECT;
:REFERENCE 'a reference'       => :OBJECT;
:EXECUTABLE 'a file that should be executed when queried' => :FILE;
:GENERIC 'a file whose type is based on its label'      => :FILE;
:PIPE 'a file produced by a type-preserving tool'      => :FILE;
:DERIVED-DIRECTORY 'a derived directory'               => :FILE;
```

Every object type other than the built-in object types must be a subtype of exactly one of `:FILE`, `:LIST`, or `:VOID`.

If the type of an object is a subtype of `:VOID`, the the object does not have a value. These types are useful for declaring tools that do not generate an output file, but do have side effects such as printing a file or writing a tape.

If the type of an object is a subtype of `:REFERENCE`, then the object refers to another object. Whenever a reference object is used, such as when it is displayed or when it is given as input to a tool, it is automatically dereferenced by Odin.

If the type of an object is a subtype of `:EXECUTABLE`, when the object is specified in a query `odin-command` (see Section 4.1), the object is automatically executed after it has been made up-to-date. Executable types are analogous to executable targets (see Section 2.4.1).

If the type of an object is a subtype of `:GENERIC`, then the object is a derived file whose type is based on the root of the source file from which it is derived (see Section 2.2.2). A generic type is commonly produced as the result of a checkout operation from a source control file or as the result of an uncompress operation on a compressed file. For example, if the root of `prog.c.sm,v` is `prog.c.sm`, if the type of a file named `prog.c.sm` is `:c.sm`, and if `:checkout` is declared in the derivation graph entry:

```
:checkout 'version checked out of a version control file' => :GENERIC;
```

then the type of `prog.c.sm,v :checkout` is `:c.sm`.

If the type of an object is a subtype of `:PIPE`, then the object is a derived file whose type is the same as the input object to the tool that produced it. For example, if `:stdout` is declared in the derivation graph entry:

```
:stdout 'standard output file of a tool run' => :PIPE;
```

and if the type of `prog.c` is `:c`, then the type of `prog.c +cmd='sed s/a/b/' :stdout` is also `:c`.

If the type of an object is a subtype of `:DERIVED-DIRECTORY`, then the object must be a directory produced by a tool.

## B.4.2 Built-In Derived Object Types

<code>:err</code>	<code>'errors'</code>	<code>=&gt; :FILE;</code>
<code>:warn</code>	<code>'warnings'</code>	<code>=&gt; :FILE;</code>
<code>:targets</code>	<code>'targets in an Odinfile'</code>	<code>=&gt; :FILE;</code>
<code>:name</code>	<code>'name of an object'</code>	<code>=&gt; :FILE;</code>
<code>:names</code>	<code>'names of the elements of a list'</code>	<code>=&gt; :FILE;</code>
<code>:filename</code>	<code>'filename of a file'</code>	<code>=&gt; :FILE;</code>
<code>:ls</code>	<code>'filenames of the elements of a list'</code>	<code>=&gt; :FILE;</code>
<code>:cat</code>	<code>'concatenation of the contents of the elements of a list'</code>	<code>=&gt; :FILE;</code>
<code>:first</code>	<code>'first element of a list'</code>	<code>=&gt; :OBJECT;</code>
<code>:union</code>	<code>'list union'</code>	<code>=&gt; :LIST;</code>
<code>:id</code>	<code>'unique numeric id of an object'</code>	<code>=&gt; :FILE;</code>
<code>:label</code>	<code>'label of an object'</code>	<code>=&gt; :FILE;</code>
<code>:labels</code>	<code>'labels of the elements of a list'</code>	<code>=&gt; :FILE;</code>
<code>:dir</code>	<code>'directory containing a file'</code>	<code>=&gt; :REFERENCE;</code>
<code>:vir_dir</code>	<code>'directory containing virtual files'</code>	<code>=&gt; :DERIVED-DIRECTORY;</code>
<code>:depend</code>	<code>'source dependencies'</code>	<code>=&gt; :LIST;</code>
<code>:source_names</code>	<code>'source components of a system model'</code>	<code>=&gt; :FILE;</code>
<code>:operation_names</code>	<code>'operation components of a system model'</code>	<code>=&gt; :FILE;</code>
<code>:view</code>	<code>'view from a view-path specification'</code>	<code>=&gt; :LIST;</code>
<code>:expand_hooks</code>	<code>'expand hooks'</code>	<code>=&gt; :FILE;</code>

```

:apply      'apply operations'          => :REFERENCE;
:odin_help  'top level help for odin'    => :EXECUTABLE;

```

The `:err` derivation produces a text file from an arbitrary input object. This text file contains all error messages generated by any tool used in the process of creating the input object. The `:warn` derivation produces a text file that contains both warning and error messages.

An `Odinfile` can contain nested target definitions (see Section 2.6). The `:targets` derivation produces a flattened version of the `Odinfile` containing all target definitions.

The `:name` derivation produces a text file that contains an odin-expression that specifies the input object. The `:names` derivation produces a text file from a list, where the text file contains a sequence of odin-expressions corresponding to the elements of the input object.

The `:filename` derivation produces a text file from a file, where the text file contains the host filename of the input object. The `:ls` derivation produces a text file from a list, where the text file contains a sequence of host filenames corresponding to the elements of the input object. The filename of a string object is the identifier that names the string object.

The `:cat` derivation produces a text file from a list, where the text file contains the concatenation of the contents of the elements of the input object. The contents of a string object is the identifier that names the string object.

The `:first` derivation produces an object from a list, by selecting the first element of the list. This derivations is primarily used in parameter type declarations.

The `:union` derivation produces a list from a list, where each element in the input list appears exactly once in the output list.

The `:id` derivation generates a text file containing a unique integer for the input object. The `:label` derivation generates a text file containing the label of the input object. The `:labels` derivation produces a text file from a list, where the text file contains a sequence of labels corresponding to the elements of the input object.

The `:dir` derivation produces a reference object that refers to the directory containing the input object. If the input object is a derived object, then `:dir` produces the directory that contains the source object from which the input object is derived.

The `:vir_dir` derivation produces a derived directory containing all the virtual targets defined in the input file. For the syntax of a virtual target definition, see Section 2.3.

The `:depend` derivation produces a list from an arbitrary input object. This list contains all source objects used in the process of creating the input object.

The `:source_names` derivation produces a text file from a text file. The format of the input text file is a sequence of odin-expressions naming file objects. The output text file contains just the initial source filename component of each odin-expression (i.e. the first derivation or parameterization operator and all subsequent operations are removed). An example of such an input file would be:

```

../prog.c :fmt +debug
test.c +inc_sp=(..)

```

The result of applying `:source_names` to this input file would be:

```

../prog.c
test.c

```

The `:operation_names` derivation produces a text file from a text file. The format of the input text file is a sequence of odin-expressions naming file objects, one odin-expression per line. The output text file contains any text following the initial source filename component of each odin-expression (i.e. everything preceding the first derivation or parameterization operator is removed). An example of such an input file would be:

```

../prog.c :fmt +debug
test.c +inc_sp=(..)

```

The result of applying `:operation.names` to this input file would be:

```
:fmt +debug
+inc_sp=...
```

The `:view` derivation produces a list from a text file. The format of the text file is sequences of odin-expressions naming file objects, where each sequence is terminated by an odin-expression naming a string object. An example of such an input file would be:

```
# the search sequence for test.h
test.h
test.h,v :co
RCS/test.h,v :co
= 'test.h'
# the search sequence for sys.h
sys.h
SCCS/s.sys.h :co
= 'sys.h'
```

The `:view` list contains the first file from each sequence whose status is greater than `ERROR`. If none of the files in a particular sequence have the appropriate status, the status of the `:view` object is set to `ERROR`, unless the sequence is terminated by the empty string odin-expression:

```
= ''
```

in which case that sequence is just ignored. The `:view` derivation depends only on the file selected from a given sequence and on the files preceding the selected file in the given sequence. This means that error messages from these preceding files are included in `:error` reports for the `:view` object, but any derived files following a selected files from a given sequence are ignored when the `:view` object is made up-to-date.

The `:expand_hooks` derivation produces a text file from an input text file by expanding all *hooks* found in the input file. A hook is a way of linking a message back to an appropriate area in a source file that is relevant to understanding the given message.

A hook is text of the form:

```
(| odin-expression | hooktype | hookdata |)
```

where `odin-expression` is the name of the object referenced by the hook, `hooktype` is the type of the hook, and `hookdata` is the actual reference. In order to escape the delimiter `'(|', '|', or '|)'`, precede it with a backslash.

The `:expand_hooks` derivation replaces all constructs of this form with the result of applying `:hooktype` to the object specified by `odin-expression`. The value of `hookdata` is placed in the `+hookvalue` parameter. Hooks appearing in the `hookdata` string (*nested hooks*) are passed unexpanded to the `:hooktype` tool. The result of the `:hooktype` derivation should be text that references the appropriate location in a source file. This text can in turn contain additional hooks that will be expanded (in particular, this allows for the expansion of nested hooks).

If `:hooktype` cannot be applied to the object specified by `odin-expression`, Odin attempts to apply `:hooktype` to any object from which this object can be derived. If all attempts to apply `:hooktype` fail, the result of hook expansion is the input text with the delimiters around the hook deleted.

The `:apply` derivation produces a reference object from an input object and a `+apply` parameter. The value of the `+apply` parameter is a file containing an arbitrary sequence of selection, derivation, and parameterization expressions, which are applied to the input object by the `:apply` derivation. For example, if the file `apply.expr` contained the text:

```
+debug :output /OUT.DAT
```

the two odin-expressions:

```
prog.c +debug :output /OUT.DAT
prog.c +apply=(apply.expr) :apply
```

would be equivalent.

The `:odin_help` derivation produces an executable file from the current working directory. This file is executed whenever the user requests help with the `odin-command`

-> ?

The `:odin_help` derivation can be overridden in a tool package to provide a different response to this help command.

## B.5 Parameter Type Declarations

A parameter type declaration consists of the parameter type being declared, a help identifier, a right arrow, and an object type. When a parameter type is used as the input to a tool, the list of values for that parameter type is first derived to the specified object type. For example, the derivation graph entries:

```
+define 'a macro definition'      => :cat;
+lib    'an object code library' => :ls;
+debug  'debug flag'              => :first;
```

declare a `+define` parameter type whose values are `:cat`'ed, a `+lib` parameter type whose values are `:ls`'ed, and a `+debug` parameter type which is a flag.

## B.6 Environment Variable Declarations

An environment variable declaration consists of the environment variable being declared, a help identifier, an equal-sign, and either an identifier or a parenthesized `odin-expression`. The value following the equal-sign specifies the default value for that variable. For example, the derivation graph entries:

```
$CC 'name of the C compiler' = 'gcc';

$CC_HOME 'directory containing the C compiler tools' = '/usr/local/bin';
```

specify default values for `$CC` and `$CC_HOME`. These values can then be overridden by corresponding values in the environment when a particular cache is created or reset. For example, if the value of `$CC_HOME` is defined to be `/usr/public/bin` in the environment of the process that resets a cache, then the value of `$CC_HOME` is `/usr/public/bin` whenever it is used in `odin-expressions` or `odin-commands` for that cache.

If the default value is a parenthesized `odin-expression`, the expression is evaluated and the resulting filename is used as the default value.

The current environment variables for a cache can be found in `$ODINCACHE/ENV`.

## B.7 Tool Declarations

A tool declaration consists of a tool name, the tool inputs, a right arrow, and the tool outputs. The tool name is either `EXEC`, `COLLECT`, `READ-LIST`, or `READ-REFERENCE`. A tool input is an identifier or an *input-expression* in parentheses (see Section B.7.1). A tool output is an object type in parentheses.

For example, the derivation graph entry:

```
EXEC (/bin/yacc) -dv (:y) => (:y.tab.c) (:y.tab.h) (:y.output);
```

declares an `EXEC` tool that has inputs `(/bin/yacc)`, `-dv`, and `(:y)`, and that has outputs: `(:y.tab.c)`, `(:y.tab.h)`, and `(:y.output)`.

### B.7.1 Input-Expressions

An *input-expression* is just like an *odin-expression*, except that a *derived input* or a *parameter input* can appear wherever a source filename can appear in an *odin-expression*. A *derived input* consists of a declared object type (e.g. `:o`), while a *parameter input* consists of a declared parameter type (e.g. `+debug`).

An input-expression can contain references to declared environment variables. For example, the following derivation graph entries declare and reference the environment variable `$YACC_HOME`:

```
$YACC_HOME 'directory containing yacc' = '/bin';

EXEC ($YACC_HOME/yacc) -dv (:y) => (:y.tab.c) (:y.tab.h) (:y.output);
```

An input-expression can contain *second-order derivations*. A second-order derivation consists of a *second-order object type* followed by an equal sign and a derivation expression. For example, the input-expression:

```
:lookup=:o
```

contains the second-order object type `:lookup` whose argument is the derivation `:o`.

The five second-order object types are `:map`, `:recurse`, `:extract`, `:delete`, and `:lookup`. The derivation `:map=:type` takes a list as input and produces a list containing the application of `:type` to each element of the input list. The derivation `:recurse=:type` takes a list as input and produces a list containing the application of `:type` `:recurse=:type` to each element of the input list to which `:type` can be applied, and just the original element otherwise. The derivation `:recurse=:type` can also take a file as input, in which case the input is treated as if it were a list containing that file as its only member. The derivation `:extract=:type` takes a list as input and produces a list containing each element of the input list that is a subtype of `:type`. The derivation `:delete=:type` takes a list as input and produces a list containing each element of the input list that is not a subtype of `:type`. The derivation `:lookup=:type` takes an object as input and produces a reference to an object which is the application of `:type` to the input object, except that each parameter value that is a reference or list is replaced by the sequence of files contained by the reference or list.

### B.7.2 Ignoring the Status of a Tool Input

By default, a tool is not run if any of its inputs have `ERROR` status or less. In case the tool should ignore the status of an input object, an ampersand is appended to the declaration of that input. For example, the derivation graph entry:

```
EXEC (od.sh) (:FILE)& => (:FILE);
```

declares that the `od.sh` tool does not care what the status of the `:FILE` input object is.

### B.7.3 Ignoring Changes to the Value of a Tool Input

If a tool only uses the name of an input file or only uses the names of the elements of an input list, then that input is marked with an at-sign. In this case Odin ignores changes to the value(s) of the input object, and only reruns the tool if the name(s) change. For example, consider a tool that takes all relative names specified in `#include` statements from an input file, and forms the cross product of those names with a set of directory names in a specified search path. A derivation graph entry for such a tool is:

```
EXEC (c_inc.sh) (:c) (+search_path)@ => (:c_inc);
```

This declares that `c_incs.sh` should be re-executed if the `:c` file or the names in the `+search_path` parameter change, but not if the contents of the directories in the search path change.

### B.7.4 EXEC Tool

The EXEC tool passes its inputs to the Unix `exec1()` system call and expects its outputs to be placed in current working directory. For example, the derivation graph entry:

```
EXEC (/bin/ld) -o exe (:o) => (:exe);
```

declares that a file named `exe` is produced in the current working directory by executing the C statement:

```
status = exec1("/bin/ld", "/bin/ld", "-o", "exe", "/xxx/filename.o");
```

where `/xxx/filename.o` is the name of a file with type `:o`. When the `exec1()` completes, Odin moves all output files into the cache.

If the first argument (the program name) ends with the extension `.lua`, the `exec1()` is replaced by an invocation of the script using a built-in Lua interpreter. See Appendix F for a list of additional functions provided by this interpreter. All references to `exec1()` arguments below refer to the arguments to the Lua script, passed in via `arg`, instead. On UNIX, the script is executed in a separate process. On Windows, each script is executed in a separate interpreter, but the interpreter shares the process space with the Odin server. Care must be taken to not disrupt the operation of the server via externally loaded libraries.

An identifier input is passed to the `exec1()` as a C string.

An input-expression is passed to the `exec1()` as an absolute pathname of the file named by the input-expression. Any relative source filename in the input-expression refers to a file in the same directory as the derivation graph file containing the type expression. For example, if an EXEC declaration in the derivation graph file `/usr/local/Odin/misc/1.0/misc.dg` contains the input `(od.sh)`, then this input is passed to the `exec1()` as `/usr/local/Odin/misc/1.0/od.sh`.

The value of a derived or parameter input in the input-expression depends on the input object to which the tool is being applied. For example, in the odin-expression:

```
prog.c +debug :exe
```

the input object to the `:exe` derivation is `prog.c+debug`.

Each derived input is replaced by a file that is the result of applying the derived input object type to the input object. For example, when the `:exe` object is being computed for `prog.c+debug:exe`, the `:o` derived input is replaced by the file `prog.c+debug:o`.

Each parameter input is replaced by the object resulting from applying the object type specified in the parameter declaration for that parameter type, to the list of values of all parameters of that type associated with the input object. As a special case, if no parameters of that type are associated with the input object, the parameter input is replaced by the empty string.

Each `exec1()` is invoked in an empty directory provided by Odin, and the tool is expected to put its output files in that directory. The name of an output file should be the same as the type-name of the output. For example, the result of the `exec1()` call for the output `(:exe)` should be an output file named `exe`.

In addition to the declared outputs, there are two standard output files named `ERRORS` and `WARNINGS`. Fatal errors should be written to `ERRORS`, while recoverable errors should be written to `WARNINGS`. The `ERRORS` and `WARNINGS` files are used by Odin to determine if error or warning status should be set for the output of the tool.

Normally, the standard input of the `exec1()` is assigned to `/dev/null` and the standard output and error is assigned to a temporary file that is written to the standard output of the Odin client process after the `exec1()` completes. To aid in debugging of an EXEC tool, if the Odin variable `MaxBuilds` is set to 1, the standard input, output, and error of the `exec1()` are those of the Odin client process.

If the exit status of the `exec1()` call is non-zero, Odin ignores the results of the `exec1()`, and aborts the current odin-command. Non-zero exit status should therefore only be used to indicate transient failures (such as memory or disk errors). This means that most standard Unix tools must be wrapped in shell scripts, so that normal error messages are sent to the file `ERRORS` and only transient errors cause a non-zero exit status.

Since the Odin system caches the results of EXEC tools and assumes the results are still valid if none of the input objects change, the tool writer must ensure that all files referenced during a tool run are specified

as inputs to that tool. In addition to explicit input files, many tools use additional implicit input files whose names are specified in the text of the explicit input to the tool. It is necessary in these cases to provide a dependency scanning tool that produces as output a list containing the names of all of these additional input files. These additional inputs are specified in the tool declaration separated from the explicit inputs by the keyword **NEEDS**. For example, the derivation graph entry:

```
EXEC (cc.sh) (:c) (+cc_flags) NEEDS (:all_c_inc) => (:o);
```

declares that the script `cc.sh` uses as inputs an object of type `:c` and a parameter of type `+cc_flags`, and indirectly uses as input an object of type `:all_c_incs`.

### B.7.5 COLLECT Tool

When the output type of a **COLLECT** tool declaration is a list, the **COLLECT** tool produces a list whose elements are the input objects. For example, the derivation graph entry:

```
COLLECT (:c_inc) (:c_inc :map=:all_c_inc) => (:all_c_inc);
:all_c_inc 'all c includes' => :LIST;
```

declares that `:all_c_inc` is a list that contains `:c_inc` and the result of applying `:all_c_inc` to each element of `:c_inc`.

When the output type of a **COLLECT** tool declaration is a reference, the **COLLECT** tool produces a reference to the input object. For example, the derivation graph entry:

```
COLLECT (:c +debug :o) => (:debug.o);
:debug.o 'C compile with debug on' => :REFERENCE;
```

declares that `:debug.o` is a reference to an object that results from applying `+debug:o` to a `:c` file.

### B.7.6 READ-LIST Tool

The **READ-LIST** tool produces a list from an input object. If the input object is a directory, this list contains all the files in the directory. If the input object is a file, this list contains the objects named by the `odin`-expressions in the input file. For example, the derivation graph entry:

```
READ-LIST (:c.sm) => (:c.sm.list) ;
```

declares that `:c.sm.list` is the list specified by the contents of a file of type `:c.sm`.

### B.7.7 READ-REFERENCE Tool

The **READ-REFERENCE** tool produces a reference to the object named by the single `odin`-expression in the input file. For example, the derivation graph entry:

```
READ-REFERENCE (:stub.name) => (:stub.ref) ;
```

declares that `:stub.ref` will be a reference to the object named by the contents of a file of type `:stub.name`.



## Appendix C

# Derivation Graph Syntax

```
DerivationGraph -> DGEEntry ';' DerivationGraph
                -> ;

DGEEntry        -> 'BANNER' "Identifier"
                -> 'INCLUDE' "Identifier"
                -> Pattern '=' '>' ObjectType
                -> ObjectType Description '=' '>' SuperType+
                -> ParameterType Description '=' '>' ObjectType
                -> Variable Description '=' "Identifier"
                -> Tool Arg+ Needs '=' '>' Result+ ;

Pattern         -> "Identifier" '*'
                -> '*' "Identifier"
                -> '*' ;

ObjectType      -> ':' "Identifier" ;

ParameterType   -> '+' "Identifier" ;

Description     -> "Identifier" '?'
                -> "Identifier" ;

Tool            -> 'EXEC'
                -> 'COLLECT'
                -> 'READ-LIST'
                -> 'READ-REFERENCE' ;

SuperType       -> '<' ObjectType '>'
                -> ObjectType ;

Needs           -> 'NEEDS' Arg+
                -> ;

Result          -> '(' ObjectType ')' ;

Arg             -> Word
                -> FileArg
                -> FileArg '@'
```

```

-> FileArg '&' ;

FileArg      -> '(' Root Operation* ')' ;

Root         -> Word
             -> ObjectType
             -> ParameterType
             -> '/'
             -> '/' Word
             -> '%' Word ;

Operation    -> Parameter
             -> ObjectType
             -> ObjectType '=' ObjectType
             -> '/' Word
             -> '/'
             -> '%' Word ;

Parameter    -> ParameterType
             -> ParameterType '=' PrmVal+ ;

PrmVal       -> Word
             -> FileArg ;

Word         -> "Identifier"
             -> Variable;

Variable     -> '$' "Identifier";

```

## Appendix D

# Derivation Graph Example

```
# Source Types

*.c.sm => :c.sm ;

*.c => :c ;

*.o => :o ;

# Object Types

:c.sm 'system model'? => :FILE ;

:c 'C code'? => :FILE ;

:o 'object module'? => :FILE ;

:dir 'directory' => :REFERENCE ;

:c_inc 'potential C-style included files' => :LIST ;

:all_c_inc 'C-style transitively included files' => :LIST ;

:c.list 'c source code list for tools wanting list input' => :c.sm.list ;

:c.sm.list 'list' => :LIST ;

:fmt 'formatted version of C code' => :PIPE ;

:exe 'executable binary'? => :FILE ;

# Parameter Types

+search_path 'name of a directory in an include search path'? => :ls ;

+ignore 'prefix of dependencies to ignore'? => :ls ;
```

```

+debug 'debug switch'? => :first ;

+prof 'profiling switch'? => :first ;

+gnu 'use gnu tools'? => :first ;

+cc_flags 'flags for cc'? => :cat ;

+ld_flags 'flags for ld'? => :cat ;


# Environment Variables

$CC 'name of C compiler' = 'gcc';

$CC_HOME 'directory containing C compiler tools' = '/usr/local/bin';


# Tools

READ-LIST (:c.sm)
  => (:c.sm.list) ;

EXEC (indent.sh) (:c)
  => (:fmt) ;

EXEC (c_inc.sh) (:FILE) (:FILE :dir)@ (+search_path) (+ignore)
  => (:c_inc) ;

COLLECT (:c_inc :map=:all_c_inc) (:c_inc)
  => (:all_c_inc) ;

EXEC (cc.sh) (:c) (:c :dir)@ (+search_path)
  (+debug) (+prof) (+gnu) (+cc_flags) NEEDS (:all_c_inc)& $CC $CC_HOME
  => (:o) ;

COLLECT (:c)
  => (:c.list) ;

EXEC (ld.sh) (:c.sm.list :map=:o) (+debug) (+gnu) (+prof) (+ld_flags)
  => (:exe) ;

```

## Appendix E

# Odin Tool Package Library

The following sections document the tool package library included in the Odin distribution. Each section documents a single package.

### E.1 `config`

The `config` package is at version 1.18.0. This package stores the results of initial configuration as initial values for environment variables:

- `$ODIN_VERSION_STR` — odin-build release number
- `$ODIN_CC` — name of the C compiler
- `$ODIN_CC_FLAGS` — C compiler flags
- `$ODIN_CPP` — name of the C preprocessor
- `$ODIN_CPP_FLAGS` — C preprocessor flags
- `$ODIN_CXX` — name of the C++ compiler
- `$ODIN_CXX_FLAGS` — CC flags
- `$ODIN_YACC` — name of the yacc compiler
- `$ODIN_RANLIB` — archive symbol table program

### E.2 `odin`

The `odin` package is at version 1.19.0. This package includes the main Odin executables. Actions described in this package are built-in to the Odin server. In addition, this package is home to the remote build server. This can be overridden by specifying a different `$ODIN_RBSCMD` (relative to the current cache's package directory). Its arguments are, in order, the remote host name, the Odin cache directory name, the local host name, and the local port number to which to connect. The default command, `odin/rbs.sh`, simply uses `rsh` to execute the `PKGS/odin/rbs.exe` command out of the cache directory of the remote host, with the local host name and port number as arguments.

The following environment variables directly affect Odin operation:

- `$DUMPCORE` — whether to dump core when anomaly detected
- `$ODIN_RBSCMD` — the name of the Odin remote-build-server

- `$ODIN_SHORTNAMES` — use short cache filenames
- `$ODIN_LOCALIPC` — use Unix domain interprocess communication
- `$ODIN_SERVER` — internal use only: non-empty spawns a server
- `$ODIN_SERVER_SHUTDOWN` — internal use only: non-empty causes client to send shutdown command to server
- `$ODINCACHE` — the Odin cache directory. The odin wrapper script normally sets this based on `$ODIN` and `$ODINVIEW`. `$ODIN` is a top-level directory for the cache, and `$ODINVIEW` is the directory under that (by default, these are `$HOME/.ODIN` and the local host name, respectively).

In addition, the following environment variables are available to tool programs:

- `$ODINVERBOSE` — if non-empty, a trace message should be printed to standard output.
- `$ODINRBSHOST` — a prefix to prepend to any trace messages to indicate what host it is running on
- `$ODINCACHE` — the Odin cache directory

### E.2.1 Other transformations

In addition, the following transformations are defined within this package. They are used in the other packages as indicated.

`:targets_ptr` (Pointer to the targets file) from source of type `:OBJECT` (Any object). Given a path or an Odinfile, create a copy in the cache and return the name of the copy.

`:targets_inc` (Directly nested target definitions) from source of type `:FILE` (A file). Given a path or an Odinfile, find all directly nested target definitions.

`:all_targets_inc` (All nested target definitions) from source of type `:targets_inc` (Directly nested target definitions). Given a path or an Odinfile, find all nested target definitions.

`:targets` (Target definitions from an Odinfile) from source of type `:FILE` (A file). Given an input that is an Odinfile, convert all nested target definitions into their constituent text

`:act_tgt.list` (Actual targets) from source of type `:targets_ptr` (Pointer to the targets file). Given a path or an Odinfile, list all targets which map to real files.

`:vir_tgt.list` (Virtual targets) from source of type `:targets_ptr` (Pointer to the targets file). Given a path or an Odinfile, list all targets which are virtual or are in virtual directories.

`:vir_dir` (A virtual directory) from source of type `:vir_tgt.list` (Virtual targets). Produce a virtual directory containing all the given targets. This can be used to isolate targets into a directory for a tool. Since targets rename expressions, they can also be used to rename inputs.

`:copy` (Copy) from source of type `:OBJECT` (Any object) using the following parameters:

`+copy_dest`  
Where to copy

Copy the given file or directory to the target specified in `+copy_dest`.

`:copy_cmd` (Copy command) from source of type `:OBJECT` (Any object) using the following parameters:

`+copy_dest_desc`  
Where to copy

Copying in or out actually uses this. The input object is the input side of the copy, and the `+copy_dest_desc` is the output side of the copy. Internally, the `:copy` transformation is used for each individual copy.

`:nested_hooks, :hook_values.of` (Nested hooks, Odinfile containing hook values) from source of type `:FILE` (A file). This is a helper to retrieve nested hooks. It is normally not used directly, but rather as a side effect of `:expand_hooks`.

`:expand_hooks` (Expand hooks) from source of type `:nested_hooks` (Nested hooks) with additional source of type `:FILE` (A file). Given a file with hooks, expand all hooks into an output file. A hook could be more commonly called a cross-reference, or a mark. A hook may be surrounded by other text. Its syntax is:

(*| odin-expression | hooktype | hookdata |*)

Backslashes can be used to escape the delimiters.

The *odin-expression* is what derives the object being referenced by the hook. The *hooktype* derivation is applied to the expression with the parameter `+hookvalue` set to the *hookdata*, and the entire hook is replaced by the results. Hooks are applied recursively; that is, if the replacement text contains hooks, they are evaluated and replaced as well.

These hooks are normally used to transform error messages from various formats to a standard format.

`:prm_values` (List of parameter values) from source of type `:FILE` (A file). FIXME: document

`:first` (First object in a list) from source of type `:LIST` (A list). Returns the first element of a list. Often used in parameter definitions to select a single value.

`:err` (Errors) from source of type `:OBJECT` (Any object). All error messages produced by any tool used to create this object.

`:warn` (Warnings) from source of type `:OBJECT` (Any object). All error and warning messages produced by any tool used to create this object.

`:dir` (Directory of a file) from source of type `:OBJECT` (Any object). Returns a reference to the directory containing the object, or, if the object is derived, the directory containing the source object from which it was derived. Use `:filename` to manually dereference.

`:name` (Name of a file) from source of type `:OBJECT` (Any object). Given any object, give a concise, unique Odin-expr which produces that object.

`:names` (Names of the elements of a list) from source of type `:LIST` (A list). Given a list of objects, give a concise, unique Odin-expr which produces each object in the list, one expression per line.

`:filename` (Filename of a file) from source of type `:FILE` (A file). Given a derivation which produces a file, give that file's name. For derived files, the name is that of the file in the cache. For literal strings, the string is its own name.

`:ls` (Filenames of the elements of a list) from source of type `:LIST` (A list). Converts a list of derivations to a printed list of file names (as per the `:filename` derivation), one per line.

`:cat` (Concatenation of the contents of the elements of a list) from source of type `:LIST` (A list). Concatenates the contents of all files in the list to produce a single output file. The contents of a literal string are the literal string itself.

`:union` (List union) from source of type `:LIST` (A list). Removes duplicates from a list.

`:label` (Label of a file) from source of type `:OBJECT` (Any object). Give the label (base filename plus type name modifications) of an input object.

**:labels** (Labels of the elements of a list) from source of type **:LIST** (A list). Give the label (**:label**) of each element of a list, one per line.

**:id** (Id of an object) from source of type **:OBJECT** (Any object). Prints an object's unique decimal ID.

**:depend** (Source dependencies) from source of type **:OBJECT** (Any object). List of all source files (including automatically added ones, like the results of header and library scanning) that the given object depends on. If any object in this list changes, the object must be rebuilt.

**:source\_names** (Source components of a system model) from source of type **:FILE** (A file). Given a file with one odin expr per line, each describing operations on a source, return the names of the sources, one per line (i.e., strip all operations from the expressions).

**:operation\_names** (Operation components of a system model) from source of type **:FILE** (A file). Given a file with one odin expr per line, each describing operations on a source, return the the operations, one per line (i.e., strip all source names from the expressions).

**:apply** (Apply operations from a file) from source of type **:OBJECT** (Any object) using the following parameters:

**+apply**

Operations to apply

The **+apply** parameter is required. This derivation applies the operations given by the **+apply** parameter.

**:odin\_help** (Top level help for odin) from source (options only). This produces an executable which is run when general help is requested by the user. By default, it is a simple program which prints the Odin command syntax. Override the one in odin if different help is desired.

See also **:prefix\_help**, **:suffix\_help**, **:drv\_help**, and **:prm\_help**. There is no way to override the help text for plain strings, type names in mid-derivation, copy arguments, utility invocation, built-in variables, path elements, secondary parameters, parameter arguments, or second-order derivation types.

**:prefix\_help** (List of known prefixes) from source of type **:OBJECT** (Any object). This derivation is invoked to print the list of all prefix file name patterns. Override the one in odin if different help is desired. The input object is always **()** and may be ignored.

**:suffix\_help** (List of known suffixes) from source of type **:OBJECT** (Any object). This derivation is invoked to print the list of all file name suffix patterns. Override the one in odin if different help is desired. The input parameter is always **()** and may be ignored.

**:drv\_help** (Applicable derivation types) from source of type **:OBJECT** (Any object). This derivation is invoked to print the list of all possible derivations in response to **:?**. If an object preceeds the help request, it is passed in.

**:prm\_help** (Applicable parameter types) from source of type **:OBJECT** (Any object) using the following parameters:

**+prm\_help**

Derivation for which parameter types are applicable

This derivation is invoked to print the list of all possible derivations in response to **+?**. If an object preceeds the help request, it is passed in. If a derivation type follows the help request, it is passed in using the **+prm\_help** parameter.



## E.3 vc

The `vc` package is at version 1.12.0. This package supports files in revision control systems (only RCS and SCCS are supported). The `:try.co` type in particular can be parented by other version control systems to hook in. However, it is likely that the `:vc*view.source.spec` derivation needs to be overridden as well.

### E.3.1 SCCS version control file

Files matching the pattern `'s.*'` are of type `:sccs` (SCCS version control file). They may be used whenever a rule calls for a supertype of `:sccs`:

Type	Description
<code>:FILE</code>	A file
<code>:OBJECT</code>	Any object
<code>:try.co</code>	Co if possible

They may be transformed to any of the following:

`:sccs.co` (Version from a version control file) using the following parameters:

`+date`  
Revision date  
`+rev`  
Revision number

Check out a version from SCCS. The parameters specify a version other than the top.

`:sccs.log` (Log file) using the following parameters:

`+date`  
Revision date  
`+rev`  
Revision number

Get the SCCS information for a revision control file. The parameters specify information for a particular revision.

As the type `:FILE` (A file), they may be transformed to any of the following:

`:targets_inc` (Directly nested target definitions). See section E.2 (odin).

`:targets` (Target definitions from an Odinfile). See section E.2 (odin).

`:nested_hooks`, `:hook_values.of` (Nested hooks, Odinfile containing hook values). See section E.2 (odin).

`:expand_hooks` (Expand hooks). See section E.2 (odin).

`:filename` (Filename of a file). See section E.2 (odin).

`:source_names` (Source components of a system model). See section E.2 (odin).

`:operation_names` (Operation components of a system model). See section E.2 (odin).

`:c_inc.view_desc` (View description of C-style included files). See section E.5 (cc).

`:cpp` (Output from cpp pre-processor). See section E.6 (cpp).

`:cxx_inc.view_desc` (View description of C++ style included files). See section E.7 (cxx).

`:idl_inc.view_desc` (View description of IDL-style included files). See section E.14 (idl).

`:boot_name` (Name of the boot-strap file). See section E.17 (misc).

`:diff` (Differences between two files). See section E.17 (misc).

`:rcp` (Archive). See section E.17 (misc).

`:so_inc_spec` (Names of Nroff-style included files). See section E.18 (roff).

`:eqn` (Output from eqn). See section E.18 (roff).

`:run` (Execution of a command line). See section E.21 (run).

`:stdout`, `:output` (Standard output from a test run, Output files from a test run). See section E.21 (run).

`:run.lua` (Execution of Lua code). See section E.21 (run).

`:lua_stdout`, `:lua_output` (Standard output from running Lua code, Output files from running Lua code). See section E.21 (run).

`:regsub` (Output from regsub substitution). See section E.22 (regsub).

As the type `:OBJECT` (Any object), they may be transformed to any of the following:

`:targets_ptr` (Pointer to the targets file). See section E.2 (odin).

`:copy` (Copy). See section E.2 (odin).

`:copy_cmd` (Copy command). See section E.2 (odin).

`:err` (Errors). See section E.2 (odin).

`:warn` (Warnings). See section E.2 (odin).

`:dir` (Directory of a file). See section E.2 (odin).

`:depend` (Source dependencies). See section E.2 (odin).

`:drv_help` (Applicable derivation types). See section E.2 (odin).

`:prm_help` (Applicable parameter types). See section E.2 (odin).

`:redo` (Redo this object step). See section E.17 (misc).

### E.3.2 A view-path system model

Files matching the pattern `*,vw` are of type `:,vw` (A view-path system model). They may be used whenever a rule calls for a supertype of `:,vw`:

Type	Description
<code>:FILE</code>	A file
<code>:OBJECT</code>	Any object
<code>:try.co</code>	Co if possible

They may be transformed to any of the following:

`:view.source.spec` (Specification of all possible source files in the view) using the following parameters:

`+view_sp`

Name of a directory in an view search path (may be given more than once)

`$ODIN_VC_RCS`

Look for RCS files in viewpath (default 'yes')

`$ODIN_VC_SCCS`

Look for SCCS files in viewpath (default 'yes')

Convert a list of odin expressions into a view spec that includes potential RCS and SCCS sources.

`:view.names` (Names of files in the view) with additional source of type `:view.source.spec` (Specification of all possible source files in the view). Convert any file names in a list of odin expressions into one which references RCS and SCCS sources, if present.

`:vw.of` (Odinfile containing the view versions) with additional source of type `:view.names` (Names of files in the view). Convert a list of odin expressions into a virtual directory spec where RCS and SCCS sources have been accounted for.

As the type `:FILE` (A file), they may be transformed to any of the following:

`:targets_inc` (Directly nested target definitions). See section E.2 (odin).

`:targets` (Target definitions from an Odinfile). See section E.2 (odin).

`:nested_hooks`, `:hook.values.of` (Nested hooks, Odinfile containing hook values). See section E.2 (odin).

`:expand_hooks` (Expand hooks). See section E.2 (odin).

`:filename` (Filename of a file). See section E.2 (odin).

`:source_names` (Source components of a system model). See section E.2 (odin).

`:operation_names` (Operation components of a system model). See section E.2 (odin).

`:c_inc.view_desc` (View description of C-style included files). See section E.5 (cc).

`:cpp` (Output from cpp pre-processor). See section E.6 (cpp).

`:cxx_inc.view_desc` (View description of C++ style included files). See section E.7 (cxx).

`:idl_inc.view_desc` (View description of IDL-style included files). See section E.14 (idl).

`:boot_name` (Name of the boot-strap file). See section E.17 (misc).

`:diff` (Differences between two files). See section E.17 (misc).

`:rcp` (Archive). See section E.17 (misc).

`:so_inc.spec` (Names of Nroff-style included files). See section E.18 (roff).

`:eqn` (Output from eqn). See section E.18 (roff).

`:run` (Execution of a command line). See section E.21 (run).

`:stdout`, `:output` (Standard output from a test run, Output files from a test run). See section E.21 (run).

`:run.lua` (Execution of Lua code). See section E.21 (run).

`:lua_stdout`, `:lua_output` (Standard output from running Lua code, Output files from running Lua code). See section E.21 (run).

`:regsub` (Output from regsub substitution). See section E.22 (regsub).

As the type `:OBJECT` (Any object), they may be transformed to any of the following:

**:targets\_ptr** (Pointer to the targets file). See section E.2 (odin).  
**:copy** (Copy). See section E.2 (odin).  
**:copy\_cmd** (Copy command). See section E.2 (odin).  
**:err** (Errors). See section E.2 (odin).  
**:warn** (Warnings). See section E.2 (odin).  
**:dir** (Directory of a file). See section E.2 (odin).  
**:depend** (Source dependencies). See section E.2 (odin).  
**:drv\_help** (Applicable derivation types). See section E.2 (odin).  
**:prm\_help** (Applicable parameter types). See section E.2 (odin).  
**:redo** (Redo this object step). See section E.17 (misc).

### E.3.3 RCS version control file

Files matching the pattern ‘\*,v’ are of type **:v** (RCS version control file). They may be used whenever a rule calls for a supertype of **:v**:

Type	Description
<b>:FILE</b>	A file
<b>:OBJECT</b>	Any object
<b>:try.co</b>	Co if possible

They may be transformed to any of the following:

**:log** (Log file) using the following parameters:

```
+date
    Revision date
+rev
    Revision number
+state
    Revision state
+who
    Revision author
```

Get the RCS information for a revision control file. The parameters specify information for a particular revision.

**:co** (Version from a version control file) using the following parameters:

```
+date
    Revision date
+rev
    Revision number
+state
    Revision state
+who
    Revision author
```

Check out a version from RCS. The parameters specify a version other than the top.

As the type `:FILE` (A file), they may be transformed to any of the following:

- `:targets_inc` (Directly nested target definitions). See section E.2 (odin).
- `:targets` (Target definitions from an Odinfile). See section E.2 (odin).
- `:nested_hooks`, `:hook_values.of` (Nested hooks, Odinfile containing hook values). See section E.2 (odin).
- `:expand_hooks` (Expand hooks). See section E.2 (odin).
- `:filename` (Filename of a file). See section E.2 (odin).
- `:source_names` (Source components of a system model). See section E.2 (odin).
- `:operation_names` (Operation components of a system model). See section E.2 (odin).
- `:c_inc.view_desc` (View description of C-style included files). See section E.5 (cc).
- `:cpp` (Output from cpp pre-processor). See section E.6 (cpp).
- `:cxx_inc.view_desc` (View description of C++ style included files). See section E.7 (cxx).
- `:idl_inc.view_desc` (View description of IDL-style included files). See section E.14 (idl).
- `:boot_name` (Name of the boot-strap file). See section E.17 (misc).
- `:diff` (Differences between two files). See section E.17 (misc).
- `:rcp` (Archive). See section E.17 (misc).
- `:so_inc_spec` (Names of Nroff-style included files). See section E.18 (roff).
- `:eqn` (Output from eqn). See section E.18 (roff).
- `:run` (Execution of a command line). See section E.21 (run).
- `:stdout`, `:output` (Standard output from a test run, Output files from a test run). See section E.21 (run).
- `:run.lua` (Execution of Lua code). See section E.21 (run).
- `:lua_stdout`, `:lua_output` (Standard output from running Lua code, Output files from running Lua code). See section E.21 (run).
- `:regsub` (Output from regsub substitution). See section E.22 (regsub).

As the type `:OBJECT` (Any object), they may be transformed to any of the following:

- `:targets_ptr` (Pointer to the targets file). See section E.2 (odin).
- `:copy` (Copy). See section E.2 (odin).
- `:copy_cmd` (Copy command). See section E.2 (odin).
- `:err` (Errors). See section E.2 (odin).
- `:warn` (Warnings). See section E.2 (odin).
- `:dir` (Directory of a file). See section E.2 (odin).
- `:depend` (Source dependencies). See section E.2 (odin).
- `:drv_help` (Applicable derivation types). See section E.2 (odin).
- `:prm_help` (Applicable parameter types). See section E.2 (odin).
- `:redo` (Redo this object step). See section E.17 (misc).

### E.3.4 Other transformations

In addition, the following transformations are defined within this package. They are used in the other packages as indicated.

**:vw.dir** (A directory containing the view versions) from source of type **:vw.of** (Odinfile containing the view versions). Convert a virtual directory spec where RCS and SCCS sources have been accounted for into a virtual directory.

## E.4 code

The `code` package is at version 1.12.0. This package supports object file management.

### E.4.1 System model of source code files

Files matching the pattern `*.sm` are of type **:sm** (System model of source code files). They may be used whenever a rule calls for a supertype of **:sm**:

Type	Description
<b>:FILE</b>	A file
<b>:OBJECT</b>	Any object
<b>:try.co</b>	Co if possible

As the type **:FILE** (A file), they may be transformed to any of the following:

**:targets\_inc** (Directly nested target definitions). See section E.2 (odin).

**:targets** (Target definitions from an Odinfile). See section E.2 (odin).

**:nested\_hooks**, **:hook\_values.of** (Nested hooks, Odinfile containing hook values). See section E.2 (odin).

**:expand\_hooks** (Expand hooks). See section E.2 (odin).

**:filename** (Filename of a file). See section E.2 (odin).

**:source\_names** (Source components of a system model). See section E.2 (odin).

**:operation\_names** (Operation components of a system model). See section E.2 (odin).

**:c\_inc.view\_desc** (View description of C-style included files). See section E.5 (cc).

**:cpp** (Output from cpp pre-processor). See section E.6 (cpp).

**:cxx\_inc.view\_desc** (View description of C++ style included files). See section E.7 (cxx).

**:idl\_inc.view\_desc** (View description of IDL-style included files). See section E.14 (idl).

**:boot\_name** (Name of the boot-strap file). See section E.17 (misc).

**:diff** (Differences between two files). See section E.17 (misc).

**:rcp** (Archive). See section E.17 (misc).

**:so\_inc.spec** (Names of Nroff-style included files). See section E.18 (roff).

**:eqn** (Output from eqn). See section E.18 (roff).

**:run** (Execution of a command line). See section E.21 (run).

**:stdout**, **:output** (Standard output from a test run, Output files from a test run). See section E.21 (run).

**:run.lua** (Execution of Lua code). See section E.21 (run).

**:lua\_stdout**, **:lua\_output** (Standard output from running Lua code, Output files from running Lua code). See section E.21 (run).

**:regsub** (Output from regsub substitution). See section E.22 (regsub).

As the type **:OBJECT** (Any object), they may be transformed to any of the following:

**:targets\_ptr** (Pointer to the targets file). See section E.2 (odin).

**:copy** (Copy). See section E.2 (odin).

**:copy\_cmd** (Copy command). See section E.2 (odin).

**:err** (Errors). See section E.2 (odin).

**:warn** (Warnings). See section E.2 (odin).

**:dir** (Directory of a file). See section E.2 (odin).

**:depend** (Source dependencies). See section E.2 (odin).

**:drv\_help** (Applicable derivation types). See section E.2 (odin).

**:prm\_help** (Applicable parameter types). See section E.2 (odin).

**:redo** (Redo this object step). See section E.17 (misc).

## E.4.2 Object library archive

Files matching the pattern `*.a` are of type **:a** (Object library archive). They may be used whenever a rule calls for a supertype of **:a**:

Type	Description
<b>:FILE</b>	A file
<b>:OBJECT</b>	Any object
<b>:try.co</b>	Co if possible

As the type **:FILE** (A file), they may be transformed to any of the following:

**:targets\_inc** (Directly nested target definitions). See section E.2 (odin).

**:targets** (Target definitions from an Odinfile). See section E.2 (odin).

**:nested\_hooks**, **:hook\_values.of** (Nested hooks, Odinfile containing hook values). See section E.2 (odin).

**:expand\_hooks** (Expand hooks). See section E.2 (odin).

**:filename** (Filename of a file). See section E.2 (odin).

**:source\_names** (Source components of a system model). See section E.2 (odin).

**:operation\_names** (Operation components of a system model). See section E.2 (odin).

**:c.inc.view\_desc** (View description of C-style included files). See section E.5 (cc).

**:cpp** (Output from cpp pre-processor). See section E.6 (cpp).

`:cxx_inc.view_desc` (View description of C++ style included files). See section E.7 (cxx).  
`:idl_inc.view_desc` (View description of IDL-style included files). See section E.14 (idl).  
`:boot_name` (Name of the boot-strap file). See section E.17 (misc).  
`:diff` (Differences between two files). See section E.17 (misc).  
`:rcp` (Archive). See section E.17 (misc).  
`:so_inc.spec` (Names of Nroff-style included files). See section E.18 (roff).  
`:eqn` (Output from eqn). See section E.18 (roff).  
`:run` (Execution of a command line). See section E.21 (run).  
`:stdout`, `:output` (Standard output from a test run, Output files from a test run). See section E.21 (run).  
`:run.lua` (Execution of Lua code). See section E.21 (run).  
`:lua_stdout`, `:lua_output` (Standard output from running Lua code, Output files from running Lua code).  
 See section E.21 (run).  
`:regsub` (Output from regsub substitution). See section E.22 (regsub).

As the type `:OBJECT` (Any object), they may be transformed to any of the following:

`:targets_ptr` (Pointer to the targets file). See section E.2 (odin).  
`:copy` (Copy). See section E.2 (odin).  
`:copy_cmd` (Copy command). See section E.2 (odin).  
`:err` (Errors). See section E.2 (odin).  
`:warn` (Warnings). See section E.2 (odin).  
`:dir` (Directory of a file). See section E.2 (odin).  
`:depend` (Source dependencies). See section E.2 (odin).  
`:drv_help` (Applicable derivation types). See section E.2 (odin).  
`:prm_help` (Applicable parameter types). See section E.2 (odin).  
`:redo` (Redo this object step). See section E.17 (misc).

### E.4.3 Object module

Files matching the pattern `*.o` are of type `:o` (Object module). They may be used whenever a rule calls for a supertype of `:o`:

Type	Description
<code>:FILE</code>	A file
<code>:OBJECT</code>	Any object
<code>:o_src</code>	Source code for object module
<code>:try.co</code>	Co if possible

As the type `:FILE` (A file), they may be transformed to any of the following:

`:targets_inc` (Directly nested target definitions). See section E.2 (odin).



`:targets` (Target definitions from an Odinfile). See section E.2 (odin).  
`:nested_hooks, :hook_values.of` (Nested hooks, Odinfile containing hook values). See section E.2 (odin).  
`:expand_hooks` (Expand hooks). See section E.2 (odin).  
`:filename` (Filename of a file). See section E.2 (odin).  
`:source_names` (Source components of a system model). See section E.2 (odin).  
`:operation_names` (Operation components of a system model). See section E.2 (odin).  
`:c_inc.view_desc` (View description of C-style included files). See section E.5 (cc).  
`:cpp` (Output from cpp pre-processor). See section E.6 (cpp).  
`:cxx_inc.view_desc` (View description of C++ style included files). See section E.7 (cxx).  
`:idl_inc.view_desc` (View description of IDL-style included files). See section E.14 (idl).  
`:boot_name` (Name of the boot-strap file). See section E.17 (misc).  
`:diff` (Differences between two files). See section E.17 (misc).  
`:rcp` (Archive). See section E.17 (misc).  
`:so_inc_spec` (Names of Nroff-style included files). See section E.18 (roff).  
`:eqn` (Output from eqn). See section E.18 (roff).  
`:run` (Execution of a command line). See section E.21 (run).  
`:stdout, :output` (Standard output from a test run, Output files from a test run). See section E.21 (run).  
`:run.lua` (Execution of Lua code). See section E.21 (run).  
`:lua_stdout, :lua_output` (Standard output from running Lua code, Output files from running Lua code).  
 See section E.21 (run).  
`:regsub` (Output from regsub substitution). See section E.22 (regsub).

As the type `:OBJECT` (Any object), they may be transformed to any of the following:

`:targets_ptr` (Pointer to the targets file). See section E.2 (odin).  
`:copy` (Copy). See section E.2 (odin).  
`:copy_cmd` (Copy command). See section E.2 (odin).  
`:err` (Errors). See section E.2 (odin).  
`:warn` (Warnings). See section E.2 (odin).  
`:dir` (Directory of a file). See section E.2 (odin).  
`:depend` (Source dependencies). See section E.2 (odin).  
`:drv_help` (Applicable derivation types). See section E.2 (odin).  
`:prm_help` (Applicable parameter types). See section E.2 (odin).  
`:redo` (Redo this object step). See section E.17 (misc).

### E.4.4 Other transformations

In addition, the following transformations are defined within this package. They are used in the other packages as indicated.

`:code.list` (List of code files) from source of type `:sm.list` (List of files in a system model). Flatten any system model file references in a system model list, recursively.

`:o.of` (Odinfile containing the object modules) from source of type `:code.list` (List of code files). Turn a list of code files into an Odinfile to map the code files which can produce object files from their full type name to their abbreviated name used elsewhere.

`:a` (Object library archive) from source of type `:o.of` (Odinfile containing the object modules) using the following parameters:

`$ODIN_RANLIB`

Archive symbol table program (default from initial Odin configuration)

Generate an archive of the specified object files, using `ar`. If `$ODIN_RANLIB` is non-empty, run it on the archive as well.

`:libraries.view_desc`, `:library_flags` (Library view description, Library command line flags) from source (options only) using the following parameters:

`+lib`

A library name (may be given more than once)

`+lib.sp`

Name of a directory in an library search path (may be given more than once)

`$ODIN_LIB.SP`

Default library search path (default `'/lib /usr/lib'`)

Generate dependency information and library flags from `+lib` and `+lib.sp` arguments. The `+lib` arguments can be command-line options (preceded by dashes), absolute paths to library files (which are added to the dependencies), and library base names (for use with the `-l` option). The latter are converted to dependencies based on the library path and added as options with `-l`. The file associated with their dependency is assumed to be `lib<name>.<ext>`, where `<ext>` is one of `a`, `so`, or `sl`, or one of those with two or more numeric extensions.

`:libraries` (Library references) from source of type `:libraries.view_desc` (Library view description). Converts view descriptor's valid entries into the list of libraries.

## E.5 cc

The `cc` package is at version 1.13.0. This package supports compiling C code. It includes the C dependency scanner and pretty printer, as well.

### E.5.1 System model for a C program

Files matching the pattern `*.c.sm` are of type `:c.sm` (System model for a C program). They may be used whenever a rule calls for a supertype of `:c.sm`:

Type	Description
<code>:FILE</code>	A file
<code>:OBJECT</code>	Any object
<code>:try.co</code>	Co if possible

As the type `:FILE` (A file), they may be transformed to any of the following:

`:c_inc.view_desc` (View description of C-style included files) using the following parameters:

- `+home`  
Home directory for relative pathnames
- `+inc_sp`  
Name of a directory in an include search path (may be given more than once)
- `+ignore`  
Egrep expression for includes to ignore (may be given more than once)
- `$ODIN_CC_I`  
Directory containing default C include files (default `'/usr/include'`)
- `$ODIN_IGNORE`  
Egrep expression for includes to ignore (default `'/usr/include'`)

Convert a file to a list of possible include files. This is for anything that preprocess with `cpp`. Note that it does not support filtering by definitions or commented-out lines. The output format is compatible with the built-in `:view` transformation; all groups are terminated by the empty string, so if an include file cannot be found, it is ignored.

Any filenames which match the `egrep` pattern in the `+ignore` parameter(s) or the `$ODIN_IGNORE` variable are filtered out. Generally, `ODIN_IGNORE` should include the system directories, like `/usr/include`.

The `+inc_sp` parameter specifies search paths; multiple options add more paths in the order given. Spaces are not supported in search paths. The `$ODIN_CC_I` variable is intended to be `cpp`'s built-in search path, which is appended to the search list. `+home` is simply an alternate, single-entry option to prepend to the path. For local includes (using `""` instead of `<>`), the file's source directory is prepended to the path as well.

`:targets_inc` (Directly nested target definitions). See section E.2 (odin).

`:targets` (Target definitions from an Odinfile). See section E.2 (odin).

`:nested_hooks`, `:hook_values.of` (Nested hooks, Odinfile containing hook values). See section E.2 (odin).

`:expand_hooks` (Expand hooks). See section E.2 (odin).

`:filename` (Filename of a file). See section E.2 (odin).

`:source_names` (Source components of a system model). See section E.2 (odin).

`:operation_names` (Operation components of a system model). See section E.2 (odin).

`:cpp` (Output from `cpp` pre-processor). See section E.6 (cpp).

`:cxx_inc.view_desc` (View description of C++ style included files). See section E.7 (cxx).

`:idl_inc.view_desc` (View description of IDL-style included files). See section E.14 (idl).

`:boot_name` (Name of the boot-strap file). See section E.17 (misc).

`:diff` (Differences between two files). See section E.17 (misc).

`:rcp` (Archive). See section E.17 (misc).

`:so_inc_spec` (Names of Nroff-style included files). See section E.18 (roff).

`:eqn` (Output from `eqn`). See section E.18 (roff).

**:run** (Execution of a command line). See section E.21 (run).

**:stdout, :output** (Standard output from a test run, Output files from a test run). See section E.21 (run).

**:run.lua** (Execution of Lua code). See section E.21 (run).

**:lua\_stdout, :lua\_output** (Standard output from running Lua code, Output files from running Lua code). See section E.21 (run).

**:regsub** (Output from regsub substitution). See section E.22 (regsub).

As the type **:OBJECT** (Any object), they may be transformed to any of the following:

**:targets\_ptr** (Pointer to the targets file). See section E.2 (odin).

**:copy** (Copy). See section E.2 (odin).

**:copy\_cmd** (Copy command). See section E.2 (odin).

**:err** (Errors). See section E.2 (odin).

**:warn** (Warnings). See section E.2 (odin).

**:dir** (Directory of a file). See section E.2 (odin).

**:depend** (Source dependencies). See section E.2 (odin).

**:drv\_help** (Applicable derivation types). See section E.2 (odin).

**:prm\_help** (Applicable parameter types). See section E.2 (odin).

**:redo** (Redo this object step). See section E.17 (misc).

### E.5.2 Executable binary

Files matching the pattern **\*.exe** are of type **:exe** (Executable binary). They may be used whenever a rule calls for a supertype of **:exe**:

Type	Description
<b>:FILE</b>	A file
<b>:OBJECT</b>	Any object
<b>:try.co</b>	Co if possible

They may be transformed to any of the following:

**:profile** (Execution profile). See section E.15 (prof).

**:dbx** (Symbolic debugger run). See section E.20 (dbx).

As the type **:FILE** (A file), they may be transformed to any of the following:

**:c\_inc.view\_desc** (View description of C-style included files) using the following parameters:

**+home**  
Home directory for relative pathnames

**+inc.sp**  
Name of a directory in an include search path (may be given more than once)

**+ignore**  
Egrep expression for includes to ignore (may be given more than once)

`$ODIN_CC_I`

Directory containing default C include files (default `'/usr/include'`)

`$ODIN_IGNORE`

Egrep expression for includes to ignore (default `'/usr/include'`)

Convert a file to a list of possible include files. This is for anything that preprocess with `cpp`. Note that it does not support filtering by definitions or commented-out lines. The output format is compatible with the built-in `:view` transformation; all groups are terminated by the empty string, so if an include file cannot be found, it is ignored.

Any filenames which match the `egrep` pattern in the `+ignore` parameter(s) or the `$ODIN_IGNORE` variable are filtered out. Generally, `ODIN_IGNORE` should include the system directories, like `/usr/include`.

The `+inc_sp` parameter specifies search paths; multiple options add more paths in the order given. Spaces are not supported in search paths. The `$ODIN_CC_I` variable is intended to be `cpp`'s built-in search path, which is appended to the search list. `+home` is simply an alternate, single-entry option to prepend to the path. For local includes (using `""` instead of `<>`), the file's source directory is prepended to the path as well.

`:targets_inc` (Directly nested target definitions). See section E.2 (odin).

`:targets` (Target definitions from an Odinfile). See section E.2 (odin).

`:nested_hooks`, `:hook_values.of` (Nested hooks, Odinfile containing hook values). See section E.2 (odin).

`:expand_hooks` (Expand hooks). See section E.2 (odin).

`:filename` (Filename of a file). See section E.2 (odin).

`:source_names` (Source components of a system model). See section E.2 (odin).

`:operation_names` (Operation components of a system model). See section E.2 (odin).

`:cpp` (Output from `cpp` pre-processor). See section E.6 (cpp).

`:cxx_inc.view_desc` (View description of C++ style included files). See section E.7 (cxx).

`:idl_inc.view_desc` (View description of IDL-style included files). See section E.14 (idl).

`:boot_name` (Name of the boot-strap file). See section E.17 (misc).

`:diff` (Differences between two files). See section E.17 (misc).

`:rcp` (Archive). See section E.17 (misc).

`:so_inc_spec` (Names of Nroff-style included files). See section E.18 (roff).

`:eqn` (Output from `eqn`). See section E.18 (roff).

`:run` (Execution of a command line). See section E.21 (run).

`:stdout`, `:output` (Standard output from a test run, Output files from a test run). See section E.21 (run).

`:run.lua` (Execution of Lua code). See section E.21 (run).

`:lua_stdout`, `:lua_output` (Standard output from running Lua code, Output files from running Lua code). See section E.21 (run).

`:regsub` (Output from `regsub` substitution). See section E.22 (regsub).

As the type `:OBJECT` (Any object), they may be transformed to any of the following:

**:targets\_ptr** (Pointer to the targets file). See section E.2 (odin).  
**:copy** (Copy). See section E.2 (odin).  
**:copy\_cmd** (Copy command). See section E.2 (odin).  
**:err** (Errors). See section E.2 (odin).  
**:warn** (Warnings). See section E.2 (odin).  
**:dir** (Directory of a file). See section E.2 (odin).  
**:depend** (Source dependencies). See section E.2 (odin).  
**:drv\_help** (Applicable derivation types). See section E.2 (odin).  
**:prm\_help** (Applicable parameter types). See section E.2 (odin).  
**:redo** (Redo this object step). See section E.17 (misc).

### E.5.3 C source code

Files matching the pattern `*.c` are of type `:c` (C source code). They may be used whenever a rule calls for a supertype of `:c`:

Type	Description
<b>:FILE</b>	A file
<b>:OBJECT</b>	Any object
<b>:o_src</b>	Source code for object module
<b>:try.co</b>	Co if possible

They may be transformed to any of the following:

**:o** (Object module) using the following parameters:

```
+inc_sp
    Name of a directory in an include search path (may be given more than once)
+cc_home
    Directory containing the C compiler
+gnu
    Use gnu tools
+debug
    Debug switch
+prof
    Profiling flag
+optimize
    Optimization flag
+define
    Macro definition (may be given more than once)
+cc
    Name of the C compiler
+cc_flags
    Flags for cc (may be given more than once)
```

`$ODIN_CC`

Name of the C compiler (default from initial Odin configuration)

`$ODIN_CC_DEBUGF`

Debug flag for C compiler (default `'-g'`)

`$ODIN_CC_FLAGS`

C compiler flags (default from initial Odin configuration)

`$ODIN_CC_HOME`

Directory containing the C compiler (default `"`)

This compiles C code into object code. Its options are passed to the C compiler as follows:

- `+gnu`, `+cc`, and `$ODIN_CC` may specify the compiler, in that order. `+gnu` is merely a shortcut for `+cc=gcc`. The compiler is parsed into command-line elements like a shell command. The search path is `+cc_home`, `$ODIN_CC_HOME`, and the path from the last cache reset, in that order. The default value of `ODIN_CC` is set in the config package when Odin is first configured.
- Specifying `+debug` adds the debugger option specified by `$ODIN_CC_DEBUGF`, which defaults to `-g`.
- Specifying `+prof` adds the profiling option `-pg`.
- Specifying `+optimize=<type>` adds the optimization option `-O<type>`.
- For each `+define` parameter, a `-D<parm>` option is added. Note that spaces are not supported: they separated multiple parameters. This is not a problem when only defining the symbol, but may require work for defining its value.
- For each `+inc_sp` parameter, a `-I<parm>` option is added. Once again, spaces are not supported.
- Finally, the parameter `+cc.flags` and the variable `$ODIN_CC_FLAGS` specify any other options, in that order. These are parsed in a shell-like manner. Any flags which could be specified using the more specific options should be used instead for compatibility.
- Actually, there is one more set of parameters: `-c` to specify only an object file, and `-o <output>` to create the appropriate output file.

When the compiler is run, all standard output is printed directly to the user. Any standard error messages are collected as warnings or errors. Which of the two they are is determined by the compiler's return code: success produces warnings, and failure produces errors.

`:fmt` (Formatted version). This runs the C code through indent, with no special options.

`:c.h` (C function declarations) using the following parameters:

`+inc_sp`

Name of a directory in an include search path (may be given more than once)

`+define`

Macro definition (may be given more than once)

This is an obsolete transformation that will be removed soon. It attempts to produce prototypes, but it does not use the C definitions and include path, and its pattern for finding functions only extracts functions with just a type name followed by a function name on a separate line with no space between the name and the opening parentheses.

`:ln`, `:lint1.log` (Local static semantic analysis of C code, Messages from first phase of lint). See section E.16 (lint).

As the type `:FILE` (A file), they may be transformed to any of the following:

`:c_inc.view_desc` (View description of C-style included files) using the following parameters:

`+home`  
Home directory for relative pathnames

`+inc_sp`  
Name of a directory in an include search path (may be given more than once)

`+ignore`  
Egrep expression for includes to ignore (may be given more than once)

`$ODIN_CC_I`  
Directory containing default C include files (default `'/usr/include'`)

`$ODIN_IGNORE`  
Egrep expression for includes to ignore (default `'/usr/include'`)

Convert a file to a list of possible include files. This is for anything that preprocess with `cpp`. Note that it does not support filtering by definitions or commented-out lines. The output format is compatible with the built-in `:view` transformation; all groups are terminated by the empty string, so if an include file cannot be found, it is ignored.

Any filenames which match the `egrep` pattern in the `+ignore` parameter(s) or the `$ODIN_IGNORE` variable are filtered out. Generally, `ODIN_IGNORE` should include the system directories, like `/usr/include`.

The `+inc_sp` parameter specifies search paths; multiple options add more paths in the order given. Spaces are not supported in search paths. The `$ODIN_CC_I` variable is intended to be `cpp`'s built-in search path, which is appended to the search list. `+home` is simply an alternate, single-entry option to prepend to the path. For local includes (using `""` instead of `<>`), the file's source directory is prepended to the path as well.

`:targets_inc` (Directly nested target definitions). See section E.2 (odin).

`:targets` (Target definitions from an Odinfile). See section E.2 (odin).

`:nested_hooks, :hook_values.of` (Nested hooks, Odinfile containing hook values). See section E.2 (odin).

`:expand_hooks` (Expand hooks). See section E.2 (odin).

`:filename` (Filename of a file). See section E.2 (odin).

`:source_names` (Source components of a system model). See section E.2 (odin).

`:operation_names` (Operation components of a system model). See section E.2 (odin).

`:cpp` (Output from `cpp` pre-processor). See section E.6 (cpp).

`:cxx_inc.view_desc` (View description of C++ style included files). See section E.7 (cxx).

`:idl_inc.view_desc` (View description of IDL-style included files). See section E.14 (idl).

`:boot_name` (Name of the boot-strap file). See section E.17 (misc).

`:diff` (Differences between two files). See section E.17 (misc).

`:rcp` (Archive). See section E.17 (misc).

`:so_inc.spec` (Names of Nroff-style included files). See section E.18 (roff).

`:eqn` (Output from `eqn`). See section E.18 (roff).

`:run` (Execution of a command line). See section E.21 (run).



**:stdout**, **:output** (Standard output from a test run, Output files from a test run). See section E.21 (run).

**:run.lua** (Execution of Lua code). See section E.21 (run).

**:lua\_stdout**, **:lua\_output** (Standard output from running Lua code, Output files from running Lua code). See section E.21 (run).

**:regsub** (Output from regsub substitution). See section E.22 (regsub).

As the type **:OBJECT** (Any object), they may be transformed to any of the following:

**:targets\_ptr** (Pointer to the targets file). See section E.2 (odin).

**:copy** (Copy). See section E.2 (odin).

**:copy\_cmd** (Copy command). See section E.2 (odin).

**:err** (Errors). See section E.2 (odin).

**:warn** (Warnings). See section E.2 (odin).

**:dir** (Directory of a file). See section E.2 (odin).

**:depend** (Source dependencies). See section E.2 (odin).

**:drv\_help** (Applicable derivation types). See section E.2 (odin).

**:prm\_help** (Applicable parameter types). See section E.2 (odin).

**:redo** (Redo this object step). See section E.17 (misc).

### E.5.4 Other transformations

In addition, the following transformations are defined within this package. They are used in the other packages as indicated.

**:c.one.list** (List of a single C file) from source of type **:c** (C source code). Convert a C file name into a list of filenames

**:c.inc** (List of potential C-style included files) from source of type **:c.inc.view\_desc** (View description of C-style included files). This processes the **:c.inc.view\_desc** entries, filtering them for existence and making them into a list.

**:all\_c.inc** (List of C-style transitively included files) from source of type **:c.inc** (List of potential C-style included files). This recursively scans all entries of the list, producing more **:c.inc** and appending them to the list.

**:exe** (Executable binary) from source of type **:o\_of** (Odinfile containing the object modules) with additional source of type **:library\_flags** (Library command line flags) using the following parameters:

**+cc\_home**  
Directory containing the C compiler

**+gnu**  
Use gnu tools

**+purify**  
Purify flag (may be given more than once)

**+debug**  
Debug switch

**+prof**  
 Profiling flag  
**+eprof**  
 Profiling flag for linking  
**+cc**  
 Name of the C compiler  
**+ld.flags**  
 Flags for ld (may be given more than once)  
**\$ODIN\_CC**  
 Name of the C compiler (default from initial Odin configuration)  
**\$ODIN\_CC\_DEBUGF**  
 Debug flag for C compiler (default '-g')  
**\$ODIN\_CC\_FLAGS**  
 C compiler flags (default from initial Odin configuration)  
**\$ODIN\_CC\_HOME**  
 Directory containing the C compiler (default '')

This runs the linker on C code. Note that the linker in this case is the C compiler without the `-c` option. Its options are passed to the C compiler as follows:

- `+gnu`, `+cc`, and `$ODIN_CC` may specify the compiler, as with `:c` to `:o`. In addition, `purify` may be run by selecting `+purify`; the multiple `+purify` options are concatenated and prepended to the linker command. When done, any extra result files from `purify` are removed. As with individual C compilation, the search path is determined by the `+cc.home` parameter, the `$ODIN_CC_HOME` variable, and the path from the last cache reset, in that order.
- Specifying `+debug` and `+prof` work the same way as with `:c` to `:o`. The `+eprof` flag is unused.
- Then, any generic options are added from `+ld.flags` and `ODIN_CC_FLAGS`, respectively. Spaces are interpreted as separators regardless for these two as well.
- Finally, any `+libs` parameters are turned into appropriate `-l` parameters and tacked on, in the order they were found.

## E.6 cpp

The `cpp` package is at version 1.13.0. This package supports running arbitrary files through the C preprocessor.

### E.6.1 Cpp input

Files matching the pattern `'*,c'` are of type `:cpp_in` (Cpp input). They may be used whenever a rule calls for a supertype of `:cpp_in`:

Type	Description
<code>:FILE</code>	A file
<code>:OBJECT</code>	Any object
<code>:try.co</code>	Co if possible

They may be transformed to any of the following:

`:gnc.cpp` (Output from `cpp` pre-processor) using the following parameters:

**+inc\_sp**  
 Name of a directory in an include search path (may be given more than once)

**+define**  
 Macro definition (may be given more than once)

**\$ODIN\_CPP**  
 Name of the C preprocessor (default from initial Odin configuration)

**\$ODIN\_CPP\_FLAGS**  
 C preprocessor flags (default from initial Odin configuration)

Run the C preprocessor on a file, changing its type according to its file name pattern after stripping the .c extension. The preprocessor is defined by the \$ODIN\_CPP environment variable. Options are passed as follows:

- For each +define parameter, a -D<parm> option is added. Note that spaces are not supported: they separated multiple parameters. This is not a problem when only defining the symbol, but may require work for defining its value.
- For each +inc\_sp parameter, a -I<parm> option is added. Once again, spaces are not supported.
- The environment variable \$ODIN\_CPP\_FLAGS is then appended.

As the type :FILE (A file), they may be transformed to any of the following:

:cpp (Output from cpp pre-processor) using the following parameters:

**+inc\_sp**  
 Name of a directory in an include search path (may be given more than once)

**+define**  
 Macro definition (may be given more than once)

**\$ODIN\_CPP**  
 Name of the C preprocessor (default from initial Odin configuration)

**\$ODIN\_CPP\_FLAGS**  
 C preprocessor flags (default from initial Odin configuration)

Run the C preprocessor on a file, retaining its type. The C preprocessor is defined by the \$ODIN\_CPP environment variable. Options are passed as follows:

- For each +define parameter, a -D<parm> option is added. Note that spaces are not supported: they separated multiple parameters. This is not a problem when only defining the symbol, but may require work for defining its value.
- For each +inc\_sp parameter, a -I<parm> option is added. Once again, spaces are not supported.
- The environment variable \$ODIN\_CPP\_FLAGS is then appended.

:targets\_inc (Directly nested target definitions). See section E.2 (odin).

:targets (Target definitions from an Odinfile). See section E.2 (odin).

:nested\_hooks, :hook\_values.of (Nested hooks, Odinfile containing hook values). See section E.2 (odin).

:expand\_hooks (Expand hooks). See section E.2 (odin).

:filename (Filename of a file). See section E.2 (odin).

:source\_names (Source components of a system model). See section E.2 (odin).

:operation\_names (Operation components of a system model). See section E.2 (odin).

`:c_inc.view_desc` (View description of C-style included files). See section E.5 (cc).  
`:cxx_inc.view_desc` (View description of C++ style included files). See section E.7 (cxx).  
`:idl_inc.view_desc` (View description of IDL-style included files). See section E.14 (idl).  
`:boot_name` (Name of the boot-strap file). See section E.17 (misc).  
`:diff` (Differences between two files). See section E.17 (misc).  
`:rcp` (Archive). See section E.17 (misc).  
`:so_inc.spec` (Names of Nroff-style included files). See section E.18 (roff).  
`:eqn` (Output from eqn). See section E.18 (roff).  
`:run` (Execution of a command line). See section E.21 (run).  
`:stdout`, `:output` (Standard output from a test run, Output files from a test run). See section E.21 (run).  
`:run.lua` (Execution of Lua code). See section E.21 (run).  
`:lua_stdout`, `:lua_output` (Standard output from running Lua code, Output files from running Lua code).  
 See section E.21 (run).  
`:regsub` (Output from regsub substitution). See section E.22 (regsub).

As the type `:OBJECT` (Any object), they may be transformed to any of the following:

`:targets_ptr` (Pointer to the targets file). See section E.2 (odin).  
`:copy` (Copy). See section E.2 (odin).  
`:copy_cmd` (Copy command). See section E.2 (odin).  
`:err` (Errors). See section E.2 (odin).  
`:warn` (Warnings). See section E.2 (odin).  
`:dir` (Directory of a file). See section E.2 (odin).  
`:depend` (Source dependencies). See section E.2 (odin).  
`:drv_help` (Applicable derivation types). See section E.2 (odin).  
`:prm_help` (Applicable parameter types). See section E.2 (odin).  
`:redo` (Redo this object step). See section E.17 (misc).

## E.7 cxx

The `cxx` package is at version 1.13.0. This package supports compiling C++ code. It includes the C++ dependency scanner as well.

### E.7.1 System model for a C++ program

Files matching the pattern `*.c++.sm`, `*.c++.sm`, `*.c++.sm`, `*.c++.sm` or, `*.c++.sm` are of type `:cxx.sm` (System model for a C++ program). They may be used whenever a rule calls for a supertype of `:cxx.sm`:

Type	Description
<code>:FILE</code>	A file
<code>:OBJECT</code>	Any object
<code>:try.co</code>	Co if possible

As the type `:FILE` (A file), they may be transformed to any of the following:

`:cxx.inc.view_desc` (View description of C++ style included files) using the following parameters:

`+home`  
Home directory for relative pathnames

`+inc.sp`  
Name of a directory in an include search path (may be given more than once)

`+ignore`  
Egrep expression for includes to ignore (may be given more than once)

`$ODIN_CXX_I`  
Directory containing default C++ include files (default `'/usr/lib/CC/include'`)

`$ODIN_IGNORE`  
Egrep expression for includes to ignore (default `'/usr/include'`)

Convert a file to a list of possible include files. This is for anything that preprocesses with `cpp`. Note that it does not support filtering by definitions of commented-out lines. The output format is compatible with the built-in `:view` transformation; all groups are terminated by the empty string, so if an include file cannot be found, it is ignored.

Any filenames which match the `egrep` pattern in the `+ignore` parameter(s) or the `$ODIN_IGNORE` variable are filtered out. Generally, `ODIN_IGNORE` should include the system directories, like `/usr/include`.

The `+inc.sp` parameter specifies search paths; multiple options add more paths in the order given. Spaces are not supported in search paths. The `$ODIN_CXX_I` variable is intended to be `cpp`'s built-in search path, which is appended to the search list. `+home` is simply an alternate, single-entry option to prepend to the path. For local includes (using `""` instead of `<>`), the file's source directory is prepended to the path as well.

`:targets_inc` (Directly nested target definitions). See section E.2 (odin).

`:targets` (Target definitions from an Odinfile). See section E.2 (odin).

`:nested_hooks, :hook_values.of` (Nested hooks, Odinfile containing hook values). See section E.2 (odin).

`:expand_hooks` (Expand hooks). See section E.2 (odin).

`:filename` (Filename of a file). See section E.2 (odin).

`:source_names` (Source components of a system model). See section E.2 (odin).

`:operation_names` (Operation components of a system model). See section E.2 (odin).

`:c.inc.view_desc` (View description of C-style included files). See section E.5 (cc).

`:cpp` (Output from `cpp` pre-processor). See section E.6 (cpp).

`:idl_inc.view_desc` (View description of IDL-style included files). See section E.14 (idl).

`:boot_name` (Name of the boot-strap file). See section E.17 (misc).

`:diff` (Differences between two files). See section E.17 (misc).

`:rcp` (Archive). See section E.17 (misc).

`:so_inc_spec` (Names of Nroff-style included files). See section E.18 (roff).

`:eqn` (Output from eqn). See section E.18 (roff).

`:run` (Execution of a command line). See section E.21 (run).

`:stdout`, `:output` (Standard output from a test run, Output files from a test run). See section E.21 (run).

`:run.lua` (Execution of Lua code). See section E.21 (run).

`:lua_stdout`, `:lua_output` (Standard output from running Lua code, Output files from running Lua code). See section E.21 (run).

`:regsub` (Output from regsub substitution). See section E.22 (regsub).

As the type `:OBJECT` (Any object), they may be transformed to any of the following:

`:targets_ptr` (Pointer to the targets file). See section E.2 (odin).

`:copy` (Copy). See section E.2 (odin).

`:copy_cmd` (Copy command). See section E.2 (odin).

`:err` (Errors). See section E.2 (odin).

`:warn` (Warnings). See section E.2 (odin).

`:dir` (Directory of a file). See section E.2 (odin).

`:depend` (Source dependencies). See section E.2 (odin).

`:drv_help` (Applicable derivation types). See section E.2 (odin).

`:prm_help` (Applicable parameter types). See section E.2 (odin).

`:redo` (Redo this object step). See section E.17 (misc).

### E.7.2 C++ source code

Files matching the pattern `*.c++`, `*.c++`, `*.c++` or `*.c++` are of type `:cxx` (C++ source code). They may be used whenever a rule calls for a supertype of `:cxx`:

Type	Description
<code>:o_src</code>	Source code for object module

They may be transformed to any of the following:

`:o` (Object module) using the following parameters:

`+inc_sp`

Name of a directory in an include search path (may be given more than once)

**+gnu**  
 Use gnu tools  
**+debug**  
 Debug switch  
**+prof**  
 Profiling flag  
**+optimize**  
 Optimization flag  
**+define**  
 Macro definition (may be given more than once)  
**+cxx**  
 Name of the C++ compiler  
**+cxx.flags**  
 Flags for C++ compilation (may be given more than once)  
**\$ODIN\_CXX**  
 Name of the C++ compiler (default from initial Odin configuration)  
**\$ODIN\_CXX\_DEBUGF**  
 Debug flag for C++ compiler (default '-g')  
**\$ODIN\_CXX\_FLAGS**  
 CC flags (default from initial Odin configuration)  
**\$ODIN\_CXX\_HOME**  
 Directory containing the C++ compiler (default '')  
**\$ODIN\_CXX\_LD\_LIBRARY\_PATH**  
 Load library path for C++ (default '/usr/lib/CC')  
**\$ODIN\_CXX\_PTR**  
 Template repository directory (default '1')

This compiles C++ code into object code. Its options are passed to the C++ compiler as follows:

- The `ODIN_CXX_LD_LIBRARY_PATH` environment variable, if non-empty, is added to `LD_LIBRARY_PATH` before calling the compiler. This should only be set if the compiler requires it.
- `+gnu`, `+cxx`, and `$ODIN_CXX` may specify the compiler, in that order. `+gnu` is merely a shortcut for `+cxx=g++`. The compiler is parsed into command-line elements like a shell command. The search path is `$ODIN_CXX_HOME`, and the path from the last cache reset, in that order. The default value of `ODIN_CXX` is set in the config package when Odin is first configured.
- Setting `$ODIN_CXX_PTR` to 1 will enable passing the obsolete Sun CC compiler's `-ptr` option with a directory created for this compilation. It should not be set for newer Sun C compilers or any other compilers, for that matter.
- Specifying `+debug` adds the debugger option specified by `$ODIN_CXX_DEBUGF`, which defaults to `-g`.
- Specifying `+prof` adds the profiling option `-pg`.
- Specifying `+optimize=<type>` adds the optimization option `-O<type>`.
- For each `+define` parameter, a `-D<parm>` option is added. Note that spaces are not supported: they separated multiple parameters. This is not a problem when only defining the symbol, but may require work for defining its value.
- For each `+inc.sp` parameter, a `-I<parm>` option is added. Once again, spaces are not supported.

- Finally, the parameter `+cc-flags` and the variable `$ODIN_CXX_FLAGS` specify any other options, in that order. These are parsed in a shell-like manner. Any flags which could be specified using the more specific options should be used instead for compatibility.
- Actually, there is one more set of parameters: `-c` to specify only an object file, and `-o <output>` to create the appropriate output file.

When the compiler is run, all standard output is printed directly to the user. Any standard error messages are collected as warnings or errors. Which of the two they are is determined by the compiler's return code: success produces warnings, and failure produces errors. A few error messages related to potentially transient memory errors will result in a transient error that can be retried later without forcing it. Messages without the substring "arning" are not considered warnings, and will not set the warning status.

### E.7.3 Other transformations

In addition, the following transformations are defined within this package. They are used in the other packages as indicated.

- `:cxx.one.list` (List of a single C++ file) from source of type `:cxx` (C++ source code). Convert a C++ file name into a list of filenames
- `:cxx.inc` (List of potential C++ style included files) from source of type `:cxx.inc.view_desc` (View description of C++ style included files). This processes the `:cxx.inc.view_desc` entries, filtering them for existence and making them into a list.
- `:all_cxx.inc` (List of C++ style transitively included files) from source of type `:cxx.inc` (List of potential C++ style included files). This recursively scans all entries of the list, producing more `:cxx.inc` and appending them to the list.
- `:exe` (Executable binary) from source of type `:o_of` (Odin file containing the object modules) with additional source of type `:library_flags` (Library command line flags) using the following parameters:

```
+define
    Macro definition (may be given more than once)
+inc.sp
    Name of a directory in an include search path (may be given more than once)
+gnu
    Use gnu tools
+purify
    Purify flag (may be given more than once)
+debug
    Debug switch
+prof
    Profiling flag
+eprof
    Profiling flag for linking
+cxx
    Name of the C++ compiler
+ld.flags
    Flags for ld (may be given more than once)
$ODIN_CXX
    Name of the C++ compiler (default from initial Odin configuration)
```



**\$ODIN.CXX\_DEBUGF**  
 Debug flag for C++ compiler (default '-g')

**\$ODIN.CXX\_FLAGS**  
 CC flags (default from initial Odin configuration)

**\$ODIN.CXX\_HOME**  
 Directory containing the C++ compiler (default '')

**\$ODIN.CXX\_IGNORE\_ERR**  
 Egrep expression for errors to ignore (default '')

**\$ODIN.CXX\_LD\_LIBRARY\_PATH**  
 Load library path for C++ (default '/usr/lib/CC')

**\$ODIN.CXX\_PTR**  
 Template repository directory (default '1')

This runs the linker on C++ code. Note that the linker in this case is the C++ compiler without the -c option. Its options are passed to the C++ compiler as follows:

- The `ODIN.CXX_LD_LIBRARY_PATH` environment variable, if non-empty, is added to `LD_LIBRARY_PATH` before calling the compiler. This should only be set if the compiler requires it. This may also affect the compiler's library search path, but the dependency checker will not notice.
- `+gnu`, `+cc`, and `$ODIN.CXX` may specify the compiler, as with `:cxx` to `:o`. In addition, `purify` may be run by selecting `+purify`; the multiple `+purify` options are concatenated and prepended to the linker command. When done, any extra result files from `purify` are removed. As with individual C++ compilation, the search path is determined by the `+cc_home` parameter, the `$ODIN.CXX_HOME` variable, and the path from the last cache reset, in that order.
- Specifying `+debug` and `+prof` work the same way as with `:c` to `:o`. The `+eprof` flag is unused.
- Then, any generic options are added from `+ld.flags` and `ODIN.CXX_FLAGS`, respectively. Spaces are interpreted as separators regardless for these two as well.
- Finally, any `+libs` parameters are turned into appropriate `-l` parameters and tacked on, in the order they were found.

## E.8 odi

The `odi` package is at version 1.12.0. FIXME: document

### E.8.1 System model for an ODI C++ program

Files matching the pattern `*odi.SM` are of type `:odi.cxx.sm` (System model for an ODI C++ program). They may be used whenever a rule calls for a supertype of `:odi.cxx.sm`:

Type	Description
<code>:FILE</code>	A file
<code>:OBJECT</code>	Any object
<code>:try.co</code>	Co if possible

As the type `:FILE` (A file), they may be transformed to any of the following:

`:targets_inc` (Directly nested target definitions). See section E.2 (odin).

`:targets` (Target definitions from an Odinfile). See section E.2 (odin).

`:nested_hooks`, `:hook_values.of` (Nested hooks, Odinfile containing hook values). See section E.2 (odin).

`:expand_hooks` (Expand hooks). See section E.2 (odin).  
`:filename` (Filename of a file). See section E.2 (odin).  
`:source_names` (Source components of a system model). See section E.2 (odin).  
`:operation_names` (Operation components of a system model). See section E.2 (odin).  
`:c_inc.view_desc` (View description of C-style included files). See section E.5 (cc).  
`:cpp` (Output from cpp pre-processor). See section E.6 (cpp).  
`:cxx_inc.view_desc` (View description of C++ style included files). See section E.7 (cxx).  
`:idl_inc.view_desc` (View description of IDL-style included files). See section E.14 (idl).  
`:boot_name` (Name of the boot-strap file). See section E.17 (misc).  
`:diff` (Differences between two files). See section E.17 (misc).  
`:rcp` (Archive). See section E.17 (misc).  
`:so_inc.spec` (Names of Nroff-style included files). See section E.18 (roff).  
`:eqn` (Output from eqn). See section E.18 (roff).  
`:run` (Execution of a command line). See section E.21 (run).  
`:stdout, :output` (Standard output from a test run, Output files from a test run). See section E.21 (run).  
`:run.lua` (Execution of Lua code). See section E.21 (run).  
`:lua_stdout, :lua_output` (Standard output from running Lua code, Output files from running Lua code).  
 See section E.21 (run).  
`:regsub` (Output from regsub substitution). See section E.22 (regsub).

As the type `:OBJECT` (Any object), they may be transformed to any of the following:

`:targets_ptr` (Pointer to the targets file). See section E.2 (odin).  
`:copy` (Copy). See section E.2 (odin).  
`:copy_cmd` (Copy command). See section E.2 (odin).  
`:err` (Errors). See section E.2 (odin).  
`:warn` (Warnings). See section E.2 (odin).  
`:dir` (Directory of a file). See section E.2 (odin).  
`:depend` (Source dependencies). See section E.2 (odin).  
`:drv_help` (Applicable derivation types). See section E.2 (odin).  
`:prm_help` (Applicable parameter types). See section E.2 (odin).  
`:redo` (Redo this object step). See section E.17 (misc).

## E.8.2 Other transformations

In addition, the following transformations are defined within this package. They are used in the other packages as indicated.

`:com_schema_standin` (Compiled ObjectStore schema) from source of type `:com_schema_standin` (Compiled ObjectStore schema) using the following parameters:

`+schema_src`  
ObjectStore schema source file (may be given more than once)  
`+schema_dir`  
ObjectStore schema directory  
`+inc_sp`  
Name of a directory in an include search path (may be given more than once)  
`$OS_ROOTDIR`  
ObjectStore root directory (default `'/net/xlendi/usr2/ostore-r3.0'`)

FIXME: document

`:schema.C` (ObjectStore schema source module) from source of type `:odi.cxx.sm.list` (List of files in a C++ system model) with additional source of type `:library_flags` (Library command line flags) using the following parameters:

`+schema_dir`  
ObjectStore schema directory  
`$OS_ROOTDIR`  
ObjectStore root directory (default `'/net/xlendi/usr2/ostore-r3.0'`)

FIXME: document

`:cxx.sm.list` (List of files in a C++ system model) from source of type `:odi.cxx.sm.list` (List of files in a C++ system model) with additional source of type `:schema.C` (ObjectStore schema source module).

FIXME: document

## E.9 f77

The `f77` package is at version 1.12.0. This package enables compilation of Fortran programs.

### E.9.1 Fortran77 source code

Files matching the pattern `*.f` are of type `:f` (Fortran77 source code). They may be used whenever a rule calls for a supertype of `:f`:

Type	Description
<code>:o_src</code>	Source code for object module

They may be transformed to any of the following:

`:o` (Object module) using the following parameters:

`+debug`  
Debug switch  
`+prof`  
Profiling flag

**+f77\_home**  
 Directory containing the C compiler

**+f77**  
 Name of the C compiler

**+f77\_flags**  
 Flags for f77 (may be given more than once)

Compile Fortran code. Its options are passed to the Fortran compiler as follows:

- +f77 may specify the compiler; the default is f77. The compiler is parsed into command-line elements like a shell command. The search path is +f77\_home, followed by the path from the last cache reset.
- +prof adds the profiling option -pg.
- +debug adds the debugging option -g.
- +f77\_flags may be used to add arbitrary command line options
- The only other options are -c and the source file name.

When the compiler is run, all standard output is printed directly to the user. Any standard error messages are collected as warnings or errors. Which of the two they are is determined by the compiler's return code: success produces warnings, and failure produces errors.

## E.10 lex

The **lex** package is at version 1.12.0. This package supports automatically generated lexical analyzers.

### E.10.1 Scanner grammar

Files matching the pattern `*.1` are of type `:1` (Scanner grammar). They may be used whenever a rule calls for a supertype of `:1`:

Type	Description
<code>:FILE</code>	A file
<code>:OBJECT</code>	Any object
<code>:try.co</code>	Co if possible

They may be transformed to any of the following:

`:c` (C source code) using the following parameters:

**+yaccid**  
 Entry point for lexical analyzer

This produces a C file using **lex**. If the +yaccid parameter is specified, all occurrences of yy in resulting output are turned into the given ID, and all occurrences of YY are turned into the ID converted to upper-case. Note that this includes yy and YY within any literal C code or other text inserted by **lex**.

As the type `:FILE` (A file), they may be transformed to any of the following:

`:targets_inc` (Directly nested target definitions). See section E.2 (odin).

`:targets` (Target definitions from an Odinfile). See section E.2 (odin).

`:nested_hooks, :hook_values.of` (Nested hooks, Odinfile containing hook values). See section E.2 (odin).

- `:expand_hooks` (Expand hooks). See section E.2 (odin).
- `:filename` (Filename of a file). See section E.2 (odin).
- `:source_names` (Source components of a system model). See section E.2 (odin).
- `:operation_names` (Operation components of a system model). See section E.2 (odin).
- `:c_inc.view_desc` (View description of C-style included files). See section E.5 (cc).
- `:cpp` (Output from cpp pre-processor). See section E.6 (cpp).
- `:cxx_inc.view_desc` (View description of C++ style included files). See section E.7 (cxx).
- `:idl_inc.view_desc` (View description of IDL-style included files). See section E.14 (idl).
- `:boot_name` (Name of the boot-strap file). See section E.17 (misc).
- `:diff` (Differences between two files). See section E.17 (misc).
- `:rcp` (Archive). See section E.17 (misc).
- `:so_inc_spec` (Names of Nroff-style included files). See section E.18 (roff).
- `:eqn` (Output from eqn). See section E.18 (roff).
- `:run` (Execution of a command line). See section E.21 (run).
- `:stdout`, `:output` (Standard output from a test run, Output files from a test run). See section E.21 (run).
- `:run.lua` (Execution of Lua code). See section E.21 (run).
- `:lua_stdout`, `:lua_output` (Standard output from running Lua code, Output files from running Lua code). See section E.21 (run).
- `:regsub` (Output from regsub substitution). See section E.22 (regsub).

As the type `:OBJECT` (Any object), they may be transformed to any of the following:

- `:targets_ptr` (Pointer to the targets file). See section E.2 (odin).
- `:copy` (Copy). See section E.2 (odin).
- `:copy_cmd` (Copy command). See section E.2 (odin).
- `:err` (Errors). See section E.2 (odin).
- `:warn` (Warnings). See section E.2 (odin).
- `:dir` (Directory of a file). See section E.2 (odin).
- `:depend` (Source dependencies). See section E.2 (odin).
- `:drv_help` (Applicable derivation types). See section E.2 (odin).
- `:prm_help` (Applicable parameter types). See section E.2 (odin).
- `:redo` (Redo this object step). See section E.17 (misc).

## E.11 yacc

The `yacc` package is at version 1.16.0. This package supports automatically generated LALR(1) parsers.

### E.11.1 Yacc input

Files matching the pattern ‘\*.y’ are of type :y (Yacc input). They may be used whenever a rule calls for a supertype of :y:

Type	Description
:FILE	A file
:OBJECT	Any object
:try.co	Co if possible

They may be transformed to any of the following:

:c, :h, :yacc.log (C source code, Header file, Yacc table listing) using the following parameters:

**+yaccid**  
Entry point for lexical analyzer

**+conflict\_ok**  
Reduce/reduce and shift/reduce conflicts are OK

**+sys5**  
Use Unix System V conventions

**+gnu**  
Use gnu tools

**+yacc**  
Name of the yacc compiler

**+yacc\_flags**  
Flags for yacc (may be given more than once)

**\$ODIN\_SYS5**  
Use Unix System V conventions (default ”)

**\$ODIN\_YACC**  
Name of the yacc compiler (default from initial Odin configuration)

**\$ODIN\_YACC\_FLAGS**  
Yacc flags (default ‘-dv’)

This produces a C file using **yacc**. If the **+yaccid** parameter is specified, all occurrences of yy in resulting output are turned into the given ID, and all occurrences of YY are turned into the ID converted to upper-case. Note that this includes yy and YY within any literal C code or other text inserted by **yacc**.

The parser builder used is **yacc** by default, or the value of the **\$ODIN\_YACC** variable. If the **+gnu** option is specified, **bison** in **yacc** compatibility mode (**-y**) is used instead. The **+yacc** parameter overrides all of these. Flags passed into the parser builder are **\$ODIN\_YACC\_FLAGS**, followed by **+yacc\_flags**, followed by the **-y** inserted by the **+gnu** option. Note that this is added even if the parser builder was overridden using **+yacc**.

If the **+conflict\_ok** parameter is not specified, any conflicts flagged by the parser builder become errors. Otherwise, they are warnings. Note that there is no way to distinguish between types of conflicts; in fact, the presence of any log message containing the word conflicts will trigger an error.

The misnamed **\$ODIN\_SYS5** and **+sys5** flag parameters, if non-blank/specified, transform any obsolete type definition of the **malloc** C function to use the correct pointer type.

As the type :FILE (A file), they may be transformed to any of the following:

:targets\_inc (Directly nested target definitions). See section E.2 (odin).

`:targets` (Target definitions from an Odinfile). See section E.2 (odin).  
`:nested_hooks, :hook_values.of` (Nested hooks, Odinfile containing hook values). See section E.2 (odin).  
`:expand_hooks` (Expand hooks). See section E.2 (odin).  
`:filename` (Filename of a file). See section E.2 (odin).  
`:source_names` (Source components of a system model). See section E.2 (odin).  
`:operation_names` (Operation components of a system model). See section E.2 (odin).  
`:c_inc.view_desc` (View description of C-style included files). See section E.5 (cc).  
`:cpp` (Output from cpp pre-processor). See section E.6 (cpp).  
`:cxx_inc.view_desc` (View description of C++ style included files). See section E.7 (cxx).  
`:idl_inc.view_desc` (View description of IDL-style included files). See section E.14 (idl).  
`:boot_name` (Name of the boot-strap file). See section E.17 (misc).  
`:diff` (Differences between two files). See section E.17 (misc).  
`:rcp` (Archive). See section E.17 (misc).  
`:so_inc_spec` (Names of Nroff-style included files). See section E.18 (roff).  
`:eqn` (Output from eqn). See section E.18 (roff).  
`:run` (Execution of a command line). See section E.21 (run).  
`:stdout, :output` (Standard output from a test run, Output files from a test run). See section E.21 (run).  
`:run.lua` (Execution of Lua code). See section E.21 (run).  
`:lua_stdout, :lua_output` (Standard output from running Lua code, Output files from running Lua code). See section E.21 (run).  
`:regsub` (Output from regsub substitution). See section E.22 (regsub).

As the type `:OBJECT` (Any object), they may be transformed to any of the following:

`:targets_ptr` (Pointer to the targets file). See section E.2 (odin).  
`:copy` (Copy). See section E.2 (odin).  
`:copy_cmd` (Copy command). See section E.2 (odin).  
`:err` (Errors). See section E.2 (odin).  
`:warn` (Warnings). See section E.2 (odin).  
`:dir` (Directory of a file). See section E.2 (odin).  
`:depend` (Source dependencies). See section E.2 (odin).  
`:drv_help` (Applicable derivation types). See section E.2 (odin).  
`:prm_help` (Applicable parameter types). See section E.2 (odin).  
`:redo` (Redo this object step). See section E.17 (misc).

## E.12 tregrm

The **tregrm** package is at version 1.16.0. This package supports generating C code for LALR(1) grammars which produce tree structures as the sole side effect of parsing. This is likely too engrained into the Odin source code to be useful outside of Odin.

The input file consists of sequences of tokens, separated by whitespace if necessary. Comments are text from a hash mark (#) to the end of the line, and are ignored as whitespace. Quoted strings are any text between quotes, with backslash to escape quotes and backslashes. The quotes in this case are either both single-quotes or both double-quotes. Names are any sequence of letters, digits, dashes, underscores, dots, slashes, and commas.

The first section of the input describes C include files. For each file to include, **INCLUDE** *name* will generate a **#include** directive. The *name* should be enclosed in double-quotes for local includes and single-quotes for system includes.

The next section describes the lexical analyzer output. It begins with the word **SCANNER**. For each terminal token, the grammar name, followed by an equals sign, followed by the lexical analyzer name, followed by the node type, if required, is given. For end-of-file detection, the grammar name is the name **EOF**. For errors in lexical analysis, the grammar name is the name **ERR**. For tokens which carry no additional value, the grammar name is a quoted string in single quotes. For tokens which carry a value, the grammar name is a quoted string in double quotes. The value will be passed in a node of the type given in the node type, which is a double-quoted string following the => symbol. The node types used here must be declared in the **NODES** section, just like the grammar nodes. The value is stored on the top of **SymStack** by the lexical analyzer; see the source code for details.

The next section lists all node types used by the grammar. It is introduced by the word **NODES**. Each type is a double-quoted string.

The next section gives the grammar rules, and is introduced by the word **RULES**. Each rule consists of a terminal name, followed by one or more productions, followed by a semicolon. Each production consists of the symbol →, followed by a sequence expression, optionally followed by a node type for this sequence, as the symbol =>, followed by the node type in double quotes. The node type may be followed by a question mark to indicate that no node will be created if there are no children, or it may be enclosed in parentheses to indicate that creating a node of this type also ends the parsing. Each sequence expression consists of a sequence of terms. A term is either an element, an optional element (indicated by a trailing ?), an element which may appear zero or more times (indicated by a trailing \*), an element which may appear one or more times (indicated by a trailing +), or an element which may appear in a list of one or more elements, separated by a separator element (indicated by the list element, followed by //, followed by the separator element). Each element is either a terminal name (i.e, a quoted string), a non-terminal name, or a sequence of terms in parentheses.

The first rule is the start rule for the grammar.

A production's non-terminals and terminals with values are linked together as siblings. If the production has a node type, a new node will be created with those siblings as children; otherwise, the collected children are the value for that non-terminal.

In order to use the resulting grammar, some support routines are needed. The lexical analyzer needs **Init\_Lex** to initialize itself, **EndLex** to finish lexical analysis, and **YY\_Lex** to return the next token. The **:tok.h** derivation contains the identifiers to be returned as token values. An error in parsing calls **ParseError(char \*)** and increments the global variable **num.ParseErrors**.

In order to create and use the resulting tree, **odin/if-nod.c** or something equivalent needs to be used. The grammar in particular calls the global function **Init\_ConstructTree()** to start, and **End\_ConstructTree()** to finish. The **End** routine must return the finished tree. For every node to be added, **Action(int, int)** is called. The first parameter is the node type, and the second is the number of children. The type's absolute value is from the **:nod.h** derivation, if non-zero. If zero, the type indicates that the children should simply be linked and passed along. If positive, a node of the given type should be created, with the given number of children. If the number is positive, the children are previously created nodes. If the number is negative, there is no child, but instead a value from the lexical analyzer. If the type and child count are negative, the effect is the same as with a positive type and child count, except that the lexical analyzer should be



terminated after creating the node. If the type is negative and the child count is not, then a node should only be created if the number of children is non-zero.

### E.12.1 Input grammar for the Tregm tree-building parser generator

Files matching the pattern `*.ygi` are of type `:ygi` (Input grammar for the Tregm tree-building parser generator). They may be used whenever a rule calls for a supertype of `:ygi`:

Type	Description
<code>:FILE</code>	A file
<code>:OBJECT</code>	Any object
<code>:try.co</code>	Co if possible

They may be transformed to any of the following:

`:y`, `:nod.h`, `:tok.h`, `:ygi.log` (Yacc input, Parser node types, Parser token types, Tregm listing) using the following parameters:

`+yaccid`  
Entry point for lexical analyzer

As described in the section introduction, a `:ygi` file contains a simplified grammar description for building parse trees. This tool generates the parser generator input and header files needed for parsing.

As the type `:FILE` (A file), they may be transformed to any of the following:

`:targets_inc` (Directly nested target definitions). See section E.2 (odin).  
`:targets` (Target definitions from an Odinfile). See section E.2 (odin).  
`:nested_hooks`, `:hook_values.of` (Nested hooks, Odinfile containing hook values). See section E.2 (odin).  
`:expand_hooks` (Expand hooks). See section E.2 (odin).  
`:filename` (Filename of a file). See section E.2 (odin).  
`:source_names` (Source components of a system model). See section E.2 (odin).  
`:operation_names` (Operation components of a system model). See section E.2 (odin).  
`:c_inc.view_desc` (View description of C-style included files). See section E.5 (cc).  
`:cpp` (Output from cpp pre-processor). See section E.6 (cpp).  
`:cxx_inc.view_desc` (View description of C++ style included files). See section E.7 (cxx).  
`:idl_inc.view_desc` (View description of IDL-style included files). See section E.14 (idl).  
`:boot_name` (Name of the boot-strap file). See section E.17 (misc).  
`:diff` (Differences between two files). See section E.17 (misc).  
`:rcp` (Archive). See section E.17 (misc).  
`:so_inc.spec` (Names of Nroff-style included files). See section E.18 (roff).  
`:eqn` (Output from eqn). See section E.18 (roff).  
`:run` (Execution of a command line). See section E.21 (run).

**:stdout**, **:output** (Standard output from a test run, Output files from a test run). See section E.21 (run).  
**:run.lua** (Execution of Lua code). See section E.21 (run).  
**:lua\_stdout**, **:lua\_output** (Standard output from running Lua code, Output files from running Lua code). See section E.21 (run).  
**:regsub** (Output from regsub substitution). See section E.22 (regsub).

As the type **:OBJECT** (Any object), they may be transformed to any of the following:

**:targets\_ptr** (Pointer to the targets file). See section E.2 (odin).  
**:copy** (Copy). See section E.2 (odin).  
**:copy\_cmd** (Copy command). See section E.2 (odin).  
**:err** (Errors). See section E.2 (odin).  
**:warn** (Warnings). See section E.2 (odin).  
**:dir** (Directory of a file). See section E.2 (odin).  
**:depend** (Source dependencies). See section E.2 (odin).  
**:drv\_help** (Applicable derivation types). See section E.2 (odin).  
**:prm\_help** (Applicable parameter types). See section E.2 (odin).  
**:redo** (Redo this object step). See section E.17 (misc).

## E.13 dg

The **dg** package is at version 1.18. This package contains the derivation graph parser.

### E.13.1 Odin Derivation Graph

Files matching the pattern **\*.dg** are of type **:dg** (Odin Derivation Graph). They may be used whenever a rule calls for a supertype of **:dg**:

Type	Description
<b>:FILE</b>	A file
<b>:OBJECT</b>	Any object
<b>:try.co</b>	Co if possible

They may be transformed to any of the following:

**:dg.log**, **:c**, **:dg.tab**, **:dg.env** (Derivation Graph log, C source code, Derivation Graph tables, Derivation Graph environment variable definitions) using the following parameters:

**+pkg**  
 Name of a package or a file containing package names (may be given more than once)

Given a directed graph file, find its parent directory and from there compile all packages named in the **+pkg** parameter. This parameter must either be a package name or (by enclosing the file name in parentheses) a file listing all package names to include. Note that only those specified packages will be compiled; not even the built-in types will be added. Therefore, it is best to make the parameter a member of a virtual directory containing all desired packages, and to make the package list also contain all of these, in the order they are to be included in the final graph. The results are intended for Odin internal use only, but include:

- `:dg.log` — A verbose description of the created graph
- `:c` — A C file containing the tables.
- `:dg.tab` — A text file containing the tables for easy parsing.
- `:dg.env` — The environment variables defined by the graph, and their current values.

Normally, the default derivation graph's `:c` is compiled into the Odin executable. The `:dg.tab` of the current graph, if different from the default, is stored in `$ODINCACHE/DG`. The `:dg.env` of the current graph is stored in `$ODINCACHE/ENV`.

The information in these tables is as follows:

The `:dg.env` file has as its first line a decimal number indicating the number of entries. Following this is one line for each variable: the variable name, followed by an equals sign, followed by the current value, followed by an ASCII SOH character (1, ctrl-A). No attempt is made to escape ASCII SOH characters in the names or values, so hopefully values will not cause problems. For multi-line values, the above description considers them part of the line they started on.

In the `:dg.tab` file, strings are encoded as a dot, followed by the string, followed by an ASCII SOH character (1, ctrl-A). No attempt is made to escape ASCII SOH characters in the strings, so don't use them in your derivation graph. If a string spans more than one line, it is still considered to be part of the same line that it started on. Numbers are simply decimal digits. Fields are separated by spaces.

The first line of the `:dg.tab` file is the version number of the executable which created this file. If there is any difference from 1.16.10, the following description is likely wrong. In the C code, this is stored in the string variable `DG_Version`.

The next line of the file is a banner string, if defined by any package's derivation graph via the `BANNER` command. In the C code, this is stored in the string variable `Banner`.

The next line in the file is a space-separated sequence of counts, one for each table:

1. `SrcTyp` — stored in `num_SrcTypS` in the C code
2. `FilTyp` — stored in `num_FilTypS` in the C code
3. `PrmTyp` — stored in `num_PrmTypS` in the C code
4. `EnvVar` — stored in `num_EnvVarS` in the C code
5. `EnvVarLst` — stored in `num_EnvVarLstS` in the C code
6. `Tool` — stored in `num_ToolS` in the C code
7. `InpEdg` — stored in `num_InpEdgS` in the C code
8. `MemEdg` — stored in `num_MemEdgS` in the C code
9. `EqvEdg` — stored in `num_EqvEdgS` in the C code
10. `CastEdg` — stored in `num_CastEdgS` in the C code
11. `DrvEdg` — stored in `num_DrvEdgS` in the C code
12. `PrmTypLst` — stored in `num_PrmTypLstS` in the C code
13. `InpSpc` — stored in `num_InpSpcS` in the C code

Next is the contents of each table, one line per entry. The structure of each table entry is defined in `inc/<tabletype>.h`; for each element, the structure element name in the C code is provided. Pointers to other table elements are stored as table index numbers in the text file, with -1 indicating NULL.

1. `SrcTyp` (C table `tps_SrcTyp_SrcTypS[]`): Source Types
  - (a) `Pattern` — The suffix or prefix string
  - (b) `IsPrefix` — Non-zero if prefix pattern; otherwise suffix pattern

- (c) FilTyp — Pointer to FilTyp table entry for type of matching names
- 2. FilTyp (C table tps.FilTyp .FilTypS[]): Derivation types (all declared derivation types, as well as any types that were artificially generated).
  - (a) FTCClass — general classification dervied from parent or self:
    - FTC\_None (1) — :OBJECT, :noinput
    - FTC\_Atmc (2) — :FILE
    - FTC\_List (3) — :LIST
    - FTC\_Void (4) — :VOID
    - FTC\_Pntr (5) — :REFERENCE
    - FTC\_Exec (6) — :EXECUTABLE
    - FTC\_Generic (7) — :GENERIC
    - FTC\_Pipe (8) — :PIPE
    - FTC\_DrvDir (9) — :DERIVED-DIRECTORY
    - FTC\_Struct (10) — pseudo type for describing multiple outputs
    - FTC\_ViewSpec (11) — :view
  - (b) FTName — type name without leading colon
  - (c) Desc — on-line help for declared types
  - (d) HelpLevel — 1 for help text with a ?; 2 for normal help text, and 3 (impossible to print) for artificially generated types.
  - (e) ArgFilTyp — If this is a second-order type, this field points to the FilTyp table entry for the argument type
  - (f) Tool — A pointer to the Tool table for the tool which generates this type, if applicable. Note that parents of this type can be generated implicitly by this tool as well.
  - (g) MemEdg — If this type is FTC\_Struct, the multiple outputs it describes are linked via entries in the MemEdg table starting with the entry pointed to by this field.
  - (h) EqvEdg — If this type has full super types, they are linked via entries in the EqvEdg table starting with the entry pointed to by this field.
  - (i) CastEdg — If this type has blocking super types, they are linked via entries in the CastEdg table starting with the entry pointed to by this field.
  - (j) DrvEdg — All types derivable from this type are linked via entries in the DrvEdg table starting with the entry pointed to by this field.
  - (k) MapPrmTypLst — ? (pointer to PrmTypLst)
  - (l) IsCopy — non-zero if subtype of built-in :copy
  - (m) IsGrouping — non-zero if subtype of a type whose tool is one of PRMVALUES, APPLY, COLLECT, READ-LIST, NAME, NAMES, or EXPANDHOOKS
  - (n) IsGroupingInput — non-zero if subtype of a type whose tool is NESTEDHOOKS, or one of the second order types :map, :lookup, or :recurse
- 3. GenericFilTyp — constant pointer to FilTyp entry for :GENERIC
- 4. PipeFilTyp — constant pointer to FilTyp entry for :PIPE
- 5. ListFilTyp — constant pointer to FilTyp entry for :LIST
- 6. FileFilTyp — constant pointer to FilTyp entry for :FILE
- 7. ObjectFilTyp — constant pointer to FilTyp entry for :OBJECT
- 8. NoInputFilTyp — constant pointer to FilTyp entry for :noinput
- 9. ExpandHooksFilTyp — constant pointer to FilTyp entry for :expand.hooks in odin package
- 10. TargetsPtrFilTyp — constant pointer to FilTyp entry for :targets\_ptr in odin package

11. TargetsFilTyp — constant pointer to FilTyp entry for :targets in odin package
12. ActTargetsFilTyp — constant pointer to FilTyp entry for :act\_tgt.list in odin package
13. VirTargetsFilTyp — constant pointer to FilTyp entry for :vir\_tgt.list in odin package
14. VirDirFilTyp — constant pointer to FilTyp entry for :vir\_dir in odin package
15. CopyFilTyp — constant pointer to FilTyp entry for :copy in odin package
16. PrmValuesFilTyp — constant pointer to FilTyp entry for :prm\_values in odin package
17. ViewSpecFilTyp — constant pointer to FilTyp entry for :view\_spec in odin package
18. FirstFilTyp — constant pointer to FilTyp entry for :first
19. ApplyFilTyp — constant pointer to FilTyp entry for :apply in odin package
20. FatherFilTyp — constant pointer to FilTyp entry for :abstract in odin package
21. PrmTyp (C table tps\_PrmTyp \_PrmTypS[]): All parameter declarations
  - (a) PTName — Parameter name without leading plus.
  - (b) Desc — Help text
  - (c) HelpLevel — Help level (1 for those with ?, and 2 for those without)
  - (d) FilTyp — Derivation to apply to parameters (pointer to FilTyp)
22. NullPrmTyp — constant pointer to PrmTyp entry for +null
23. HookValPrmTyp — constant pointer to PrmTyp entry for +hookvalue
24. CopyDestPrmTyp — constant pointer to PrmTyp entry for +copy\_dest
25. ApplyPrmTyp — constant pointer to PrmTyp entry for +apply
26. PrmTypLst (C table tps\_PrmTypLst \_PrmTypLstS[]): Lists of parameter declarations used by other tables
  - (a) PrmTyp — A pointer to this list element in PrmTyp
  - (b) Next — A pointer to the next list entry in PrmTypLst
27. EnvVar (C table tps\_EnvVar \_EnvVarS[]): All environment variable declarations
  - (a) Name — Variable name without the leading dollar sign
  - (b) Desc — Help text
  - (c) HelpLevel — Help level (1 for those with ?, and 2 for those without)
  - (d) Default — Default value (string)
  - (e) IsFile — non-zero if Default is an odin-expression of a file to read for actual default value; otherwise Default is the actual default value
28. EnvVarLst (C table tps\_EnvVarLst \_EnvVarLstS[]): Lists of environment variables used by other tables
  - (a) EnvVar — Pointer to EnvVar entry which is this list element
  - (b) Next — Pointer to EnvVarLst entry containing the next element, or NULL/-1 for end-of-list.
29. Tool (C table tps\_Tool \_ToolS[]): All derivation tools.
  - (a) TClass
  - (b) InpEdg
  - (c) BasePrmTypLst
  - (d) EnvVarLst
  - (e) Package
30. SourceTool — constant pointer to Tool entry for hidden tool TC\_Source (1).
31. StrTool — constant pointer to Tool entry for hidden tool TC\_Str (12).

32. `DrvDirElmTool` — constant pointer to Tool entry for hidden tool `TC_DrvDirElm` (2).
33. `VirDirElmTool` — constant pointer to Tool entry for hidden tool `TC_VirDirElm` (47).
34. `PntrHoTool` — constant pointer to Tool entry for hidden tool `TC_PntrHo` (8).
35. `PntrElmTool` — constant pointer to Tool entry for hidden tool `TC_PntrElm` (37).
36. `InternalPntrTool` — constant pointer to Tool entry for hidden tool `TC_InternalPntr` (10).
37. `TextDefTool` — constant pointer to Tool entry for hidden tool `TC_TextDef` (18).
38. `InpEdge`
  - (a) `InpSpc`
  - (b) `InpKind`
  - (c) `IsUserArg`
  - (d) `Next`
39. `MemEdg`
  - (a) `FilTyp`
  - (b) `Next`
40. `EqvEdg`
  - (a) `FilTyp`
  - (b) `Next`
41. `CastEdg`
  - (a) `FilTyp`
  - (b) `Next`
42. `DrvEdg`
  - (a) `FilTyp`
  - (b) `PrmTypLst`
  - (c) `Next`
43. `InpSpc`
  - (a) `ISKind`
  - (b) `FilTyp`
  - (c) `PrmTyp`
  - (d) `Str`
  - (e) `IsEnvVar`
  - (f) `InpSpc`
  - (g) `Next`

As the type `:FILE` (A file), they may be transformed to any of the following:

- `:targets_inc` (Directly nested target definitions). See section E.2 (odin).
- `:targets` (Target definitions from an Odinfile). See section E.2 (odin).
- `:nested_hooks, :hook_values.of` (Nested hooks, Odinfile containing hook values). See section E.2 (odin).
- `:expand_hooks` (Expand hooks). See section E.2 (odin).
- `:filename` (Filename of a file). See section E.2 (odin).
- `:source_names` (Source components of a system model). See section E.2 (odin).
- `:operation_names` (Operation components of a system model). See section E.2 (odin).

`:c_inc.view_desc` (View description of C-style included files). See section E.5 (cc).

`:cpp` (Output from cpp pre-processor). See section E.6 (cpp).

`:cxx_inc.view_desc` (View description of C++ style included files). See section E.7 (cxx).

`:idl_inc.view_desc` (View description of IDL-style included files). See section E.14 (idl).

`:boot_name` (Name of the boot-strap file). See section E.17 (misc).

`:diff` (Differences between two files). See section E.17 (misc).

`:rcp` (Archive). See section E.17 (misc).

`:so_inc_spec` (Names of Nroff-style included files). See section E.18 (roff).

`:eqn` (Output from eqn). See section E.18 (roff).

`:run` (Execution of a command line). See section E.21 (run).

`:stdout`, `:output` (Standard output from a test run, Output files from a test run). See section E.21 (run).

`:run.lua` (Execution of Lua code). See section E.21 (run).

`:lua_stdout`, `:lua_output` (Standard output from running Lua code, Output files from running Lua code). See section E.21 (run).

`:regsub` (Output from regsub substitution). See section E.22 (regsub).

As the type `:OBJECT` (Any object), they may be transformed to any of the following:

`:targets_ptr` (Pointer to the targets file). See section E.2 (odin).

`:copy` (Copy). See section E.2 (odin).

`:copy_cmd` (Copy command). See section E.2 (odin).

`:err` (Errors). See section E.2 (odin).

`:warn` (Warnings). See section E.2 (odin).

`:dir` (Directory of a file). See section E.2 (odin).

`:depend` (Source dependencies). See section E.2 (odin).

`:drv_help` (Applicable derivation types). See section E.2 (odin).

`:prm_help` (Applicable parameter types). See section E.2 (odin).

`:redo` (Redo this object step). See section E.17 (misc).

## E.14 idl

The `idl` package is at version 1.14.0. FIXME: document

### E.14.1 C++ Interface Definition Language file

Files matching the pattern `*.IDL` or `*.IDL` are of type `:IDL` (C++ Interface Definition Language file). They may be used whenever a rule calls for a supertype of `:IDL`:

Type	Description
<code>:FILE</code>	A file
<code>:OBJECT</code>	Any object
<code>:try.co</code>	Co if possible

They may be transformed to any of the following:

`:idl_h_dir`, `:client.C`, `:epv.C`, `:sstub.c`, `:cstub.c` (Header files, Client side class implementation, Entry point vector for server, Server stubs, Client stubs) using the following parameters:

`+inc.sp`  
Name of a directory in an include search path (may be given more than once)

`+idl.flags`  
Flags for idl (may be given more than once)

`$ODIN_IDL`  
Name of the idl compiler (default `'idl++'`)

`$ODIN_IDL_FLAGS`  
Idl flags (default `'-keep source'`)

`$ODIN_IDL_HOME`  
Directory containing the idl compiler (default `"`)

FIXME: document

`:idl_targets` (Header file and library targets for an idl file) with additional source of type `:idl_h_dir` (Header files) and additional source of type `:idl.a` (Library of idl generated routines). FIXME: document

As the type `:FILE` (A file), they may be transformed to any of the following:

`:idl_inc.view_desc` (View description of IDL-style included files) using the following parameters:

`+home`  
Home directory for relative pathnames

`+inc.sp`  
Name of a directory in an include search path (may be given more than once)

`+ignore`  
Egrep expression for includes to ignore (may be given more than once)

`$ODIN_IDL_I`  
Directory containing default idl include files (default `"`)

`$ODIN_IGNORE`  
Egrep expression for includes to ignore (default `'/usr/include'`)

FIXME: document

`:targets_inc` (Directly nested target definitions). See section E.2 (odin).

`:targets` (Target definitions from an Odinfile). See section E.2 (odin).

`:nested_hooks`, `:hook_values.of` (Nested hooks, Odinfile containing hook values). See section E.2 (odin).



`:expand_hooks` (Expand hooks). See section E.2 (odin).  
`:filename` (Filename of a file). See section E.2 (odin).  
`:source_names` (Source components of a system model). See section E.2 (odin).  
`:operation_names` (Operation components of a system model). See section E.2 (odin).  
`:c_inc.view_desc` (View description of C-style included files). See section E.5 (cc).  
`:cpp` (Output from cpp pre-processor). See section E.6 (cpp).  
`:cxx_inc.view_desc` (View description of C++ style included files). See section E.7 (cxx).  
`:boot_name` (Name of the boot-strap file). See section E.17 (misc).  
`:diff` (Differences between two files). See section E.17 (misc).  
`:rcp` (Archive). See section E.17 (misc).  
`:so_inc.spec` (Names of Nroff-style included files). See section E.18 (roff).  
`:eqn` (Output from eqn). See section E.18 (roff).  
`:run` (Execution of a command line). See section E.21 (run).  
`:stdout`, `:output` (Standard output from a test run, Output files from a test run). See section E.21 (run).  
`:run.lua` (Execution of Lua code). See section E.21 (run).  
`:lua_stdout`, `:lua_output` (Standard output from running Lua code, Output files from running Lua code).  
 See section E.21 (run).  
`:regsub` (Output from regsub substitution). See section E.22 (regsub).

As the type `:OBJECT` (Any object), they may be transformed to any of the following:

`:targets_ptr` (Pointer to the targets file). See section E.2 (odin).  
`:copy` (Copy). See section E.2 (odin).  
`:copy_cmd` (Copy command). See section E.2 (odin).  
`:err` (Errors). See section E.2 (odin).  
`:warn` (Warnings). See section E.2 (odin).  
`:dir` (Directory of a file). See section E.2 (odin).  
`:depend` (Source dependencies). See section E.2 (odin).  
`:drv_help` (Applicable derivation types). See section E.2 (odin).  
`:prm_help` (Applicable parameter types). See section E.2 (odin).  
`:redo` (Redo this object step). See section E.17 (misc).

## E.14.2 Other transformations

In addition, the following transformations are defined within this package. They are used in the other packages as indicated.

- `:idl_inc` (List of potential IDL-style included files) from source of type `:idl_inc.view_desc` (View description of IDL-style included files). FIXME: document
- `:all_idl_inc` (List of IDL-style transitively included files) from source of type `:idl_inc` (List of potential IDL-style included files). FIXME: document
- `:idl_lib.list` (Idl generated routines) from source of type `:client.C` (Client side class implementation) with additional source of type `:epv.C` (Entry point vector for server) and additional source of type `:sstub.c` (Server stubs) and additional source of type `:cstub.c` (Client stubs). FIXME: document
- `:idl.a` (Library of idl generated routines) from source of type `:idl_lib.list` (Idl generated routines). FIXME: document

## E.15 prof

The `prof` package is at version 1.14.0. This package supports running the standard profiler, `gprof`.

### E.15.1 Other transformations

In addition, the following transformations are defined within this package. They are used in the other packages as indicated.

- `:profile` (Execution profile) from source of type `:exe` (Executable binary) using the following parameters:

- `+prof_data`  
Trace file (may be given more than once)
- `+prof_flags`  
Prof flags (may be given more than once)

Run the profiler, `gprof`. The profiling output data file should be specified using `+prof_data`. Any additional flags to `gprof` may be given in `+gprof_flags`.

## E.16 lint

The `lint` package is at version 1.13.0. FIXME: document

### E.16.1 Other transformations

In addition, the following transformations are defined within this package. They are used in the other packages as indicated.

- `:ln`, `:lint1.log` (Local static semantic analysis of C code, Messages from first phase of lint) from source of type `:c` (C source code) using the following parameters:

- `+inc_sp`  
Name of a directory in an include search path (may be given more than once)
- `+define`  
Macro definition (may be given more than once)

```

+lint_flags
    Lint flags (may be given more than once)
+ignore_mesg
    Message to ignore (may be given more than once)
$ODIN.LINT.FLAGS
    Lint flags (default ")
$ODIN.LINT.PASS1.FLAG
    Pass-1 only lint flag (default '-c')

```

FIXME: document

**:lint** (Static semantic analysis of C code) from source of type **:code.list** (List of code files) using the following parameters:

```

+stub
    Stub to keep Lint quiet about unused functions (may be given more than once)
+ln
    A lint library (may be given more than once)
+lint_flags
    Lint flags (may be given more than once)
+ignore_mesg
    Message to ignore (may be given more than once)
+global_ignore_mesg
    Message to ignore in global second lint pass (may be given more than once)
$ODIN.LINT.FLAGS
    Lint flags (default ")

```

FIXME: document

## E.17 misc

The **misc** package is at version 1.13.0. This package contains miscellaneous support utilities.

### E.17.1 Other transformations

In addition, the following transformations are defined within this package. They are used in the other packages as indicated.

**:redo** (Redo this object step) from source of type **:OBJECT** (Any object). This is an alias for **!redo**

**:redo\_errs** (Redo all steps with errors) from source of type **:err** (Errors). This parses the errors produced for a target, and redoes any step which apparently produced errors.

**:boot\_name** (Name of the boot-strap file) from source of type **:FILE** (A file) using the following parameters:

```

+default
    Default value

```

This checks if the target file exists. If so, the result is the name of the file. If not, the result is the name of the file specified by **+default**. The **+default** parameter may be any odin expression.

**:diff** (Differences between two files) from source of type **:FILE** (A file) using the following parameters:

**+other**  
Another file

This returns the difference between the input file and the file specified by **+other**.

**:rcp** (Archive) from source of type **:FILE** (A file) using the following parameters:

**+f\_dest**  
File destination  
**+d\_dest**  
Directory destination

This copies the source file to the destination specified by **+f\_dest** or **+d\_dest**. **+f\_dest** overrides **+d\_dest**. **+d\_dest** explicitly forces the destination to be a directory; **+f\_dest** copies directly to the target, assuming it is a file.

## E.18 roff

The **roff** package is at version 1.13.0. This package supports documentation written for UNIX **roff**.

### E.18.1 Nroff/troff input

Files matching the pattern **\*.roff** are of type **:roff** (Nroff/troff input). They may be used whenever a rule calls for a supertype of **:roff**:

Type	Description
<b>:FILE</b>	A file
<b>:OBJECT</b>	Any object
<b>:try.co</b>	Co if possible

They may be transformed to any of the following:

**:nroff** (Output from **nroff**) using the following parameters:

**+mp**  
Macro package

Run **nroff** to produce terminal formatted output from **roff** source. The **+mp** parameter may be used to specify a single package to use when formatting (e.g. **+mp='an'** for man pages).

As the type **:FILE** (A file), they may be transformed to any of the following:

**:so\_inc\_spec** (Names of Nroff-style included files) using the following parameters:

**+home**  
Home directory for relative pathnames

Create a **view\_spec** of files included by **roff** source. This is used for automatic dependency calculation.

**:eqn** (Output from **eqn**). Format equations for **roff**. The output retains its file type.

**:targets\_inc** (Directly nested target definitions). See section E.2 (odin).

**:targets** (Target definitions from an Odinfile). See section E.2 (odin).

**:nested\_hooks, :hook\_values.of** (Nested hooks, Odinfile containing hook values). See section E.2 (odin).

`:expand_hooks` (Expand hooks). See section E.2 (odin).  
`:filename` (Filename of a file). See section E.2 (odin).  
`:source_names` (Source components of a system model). See section E.2 (odin).  
`:operation_names` (Operation components of a system model). See section E.2 (odin).  
`:c_inc.view_desc` (View description of C-style included files). See section E.5 (cc).  
`:cpp` (Output from cpp pre-processor). See section E.6 (cpp).  
`:cxx_inc.view_desc` (View description of C++ style included files). See section E.7 (cxx).  
`:idl_inc.view_desc` (View description of IDL-style included files). See section E.14 (idl).  
`:boot_name` (Name of the boot-strap file). See section E.17 (misc).  
`:diff` (Differences between two files). See section E.17 (misc).  
`:rcp` (Archive). See section E.17 (misc).  
`:run` (Execution of a command line). See section E.21 (run).  
`:stdout`, `:output` (Standard output from a test run, Output files from a test run). See section E.21 (run).  
`:run.lua` (Execution of Lua code). See section E.21 (run).  
`:lua_stdout`, `:lua_output` (Standard output from running Lua code, Output files from running Lua code).  
 See section E.21 (run).  
`:regsub` (Output from regsub substitution). See section E.22 (regsub).

As the type `:OBJECT` (Any object), they may be transformed to any of the following:

`:targets_ptr` (Pointer to the targets file). See section E.2 (odin).  
`:copy` (Copy). See section E.2 (odin).  
`:copy_cmd` (Copy command). See section E.2 (odin).  
`:err` (Errors). See section E.2 (odin).  
`:warn` (Warnings). See section E.2 (odin).  
`:dir` (Directory of a file). See section E.2 (odin).  
`:depend` (Source dependencies). See section E.2 (odin).  
`:drv_help` (Applicable derivation types). See section E.2 (odin).  
`:prm_help` (Applicable parameter types). See section E.2 (odin).  
`:redo` (Redo this object step). See section E.17 (misc).

## E.18.2 Eqn input

Files matching the pattern ‘\*,eqn’ are of type `:eqn.in` (Eqn input). They may be used whenever a rule calls for a supertype of `:eqn.in`:

Type	Description
<code>:FILE</code>	A file
<code>:OBJECT</code>	Any object
<code>:try.co</code>	Co if possible

They may be transformed to any of the following:

`:gnc.eqn` (Output from eqn). Format equations for roff. The output gets its type from the file name pattern after stripping off the ,eqn.

As the type `:FILE` (A file), they may be transformed to any of the following:

`:so.inc.spec` (Names of Nroff-style included files) using the following parameters:

`+home`  
Home directory for relative pathnames

Create a `view_spec` of files included by roff source. This is used for automatic dependency calculation.

`:eqn` (Output from eqn). Format equations for roff. The output retains its file type.

`:targets.inc` (Directly nested target definitions). See section E.2 (odin).

`:targets` (Target definitions from an Odinfile). See section E.2 (odin).

`:nested_hooks`, `:hook_values.of` (Nested hooks, Odinfile containing hook values). See section E.2 (odin).

`:expand_hooks` (Expand hooks). See section E.2 (odin).

`:filename` (Filename of a file). See section E.2 (odin).

`:source_names` (Source components of a system model). See section E.2 (odin).

`:operation_names` (Operation components of a system model). See section E.2 (odin).

`:c.inc.view_desc` (View description of C-style included files). See section E.5 (cc).

`:cpp` (Output from cpp pre-processor). See section E.6 (cpp).

`:cxx.inc.view_desc` (View description of C++ style included files). See section E.7 (cxx).

`:idl.inc.view_desc` (View description of IDL-style included files). See section E.14 (idl).

`:boot_name` (Name of the boot-strap file). See section E.17 (misc).

`:diff` (Differences between two files). See section E.17 (misc).

`:rcp` (Archive). See section E.17 (misc).

`:run` (Execution of a command line). See section E.21 (run).

`:stdout`, `:output` (Standard output from a test run, Output files from a test run). See section E.21 (run).

`:run.lua` (Execution of Lua code). See section E.21 (run).

`:lua_stdout`, `:lua_output` (Standard output from running Lua code, Output files from running Lua code).  
See section E.21 (run).

`:regsub` (Output from regsub substitution). See section E.22 (regsub).

As the type `:OBJECT` (Any object), they may be transformed to any of the following:

`:targets_ptr` (Pointer to the targets file). See section E.2 (odin).

`:copy` (Copy). See section E.2 (odin).

`:copy_cmd` (Copy command). See section E.2 (odin).

`:err` (Errors). See section E.2 (odin).

`:warn` (Warnings). See section E.2 (odin).

`:dir` (Directory of a file). See section E.2 (odin).

`:depend` (Source dependencies). See section E.2 (odin).

`:drv_help` (Applicable derivation types). See section E.2 (odin).

`:prm_help` (Applicable parameter types). See section E.2 (odin).

`:redo` (Redo this object step). See section E.17 (misc).

### E.18.3 Tbl input

Files matching the pattern `*.tbl` are of type `:tbl_in` (Tbl input). They may be used whenever a rule calls for a supertype of `:tbl_in`:

Type	Description
<code>:FILE</code>	A file
<code>:OBJECT</code>	Any object
<code>:try.co</code>	Co if possible

They may be transformed to any of the following:

`:tbl` (Output from tbl). Format tables for roff.

As the type `:FILE` (A file), they may be transformed to any of the following:

`:so_inc_spec` (Names of Nroff-style included files) using the following parameters:

`+home`  
Home directory for relative pathnames

Create a `view_spec` of files included by roff source. This is used for automatic dependency calculation.

`:eqn` (Output from eqn). Format equations for roff. The output retains its file type.

`:targets_inc` (Directly nested target definitions). See section E.2 (odin).

`:targets` (Target definitions from an Odinfile). See section E.2 (odin).

`:nested_hooks`, `:hook_values.of` (Nested hooks, Odinfile containing hook values). See section E.2 (odin).

`:expand_hooks` (Expand hooks). See section E.2 (odin).

`:filename` (Filename of a file). See section E.2 (odin).

`:source_names` (Source components of a system model). See section E.2 (odin).

`:operation_names` (Operation components of a system model). See section E.2 (odin).

`:c_inc.view_desc` (View description of C-style included files). See section E.5 (cc).

`:cpp` (Output from cpp pre-processor). See section E.6 (cpp).

`:cxx_inc.view_desc` (View description of C++ style included files). See section E.7 (cxx).

`:idl_inc.view_desc` (View description of IDL-style included files). See section E.14 (idl).

`:boot_name` (Name of the boot-strap file). See section E.17 (misc).

`:diff` (Differences between two files). See section E.17 (misc).

`:rcp` (Archive). See section E.17 (misc).

`:run` (Execution of a command line). See section E.21 (run).

`:stdout`, `:output` (Standard output from a test run, Output files from a test run). See section E.21 (run).

`:run.lua` (Execution of Lua code). See section E.21 (run).

`:lua_stdout`, `:lua_output` (Standard output from running Lua code, Output files from running Lua code). See section E.21 (run).

`:regsub` (Output from regsub substitution). See section E.22 (regsub).

As the type `:OBJECT` (Any object), they may be transformed to any of the following:

`:targets_ptr` (Pointer to the targets file). See section E.2 (odin).

`:copy` (Copy). See section E.2 (odin).

`:copy_cmd` (Copy command). See section E.2 (odin).

`:err` (Errors). See section E.2 (odin).

`:warn` (Warnings). See section E.2 (odin).

`:dir` (Directory of a file). See section E.2 (odin).

`:depend` (Source dependencies). See section E.2 (odin).

`:drv_help` (Applicable derivation types). See section E.2 (odin).

`:prm_help` (Applicable parameter types). See section E.2 (odin).

`:redo` (Redo this object step). See section E.17 (misc).

#### E.18.4 Other transformations

In addition, the following transformations are defined within this package. They are used in the other packages as indicated.

`:all_so_inc` (List of Nroff-style transitively included files) from source of type `:so_inc` (List of Nroff-style included files). Collect all includes recursively from roff-generated include lists. This is used for automatic dependency calculation.



## E.19 tex

The `tex` package is at version 3.1.0. This package supports  $\text{\TeX}$  and  $\text{\LaTeX}$  documentation.

### E.19.1 Basis of a $\text{\TeX}$ formatted document

Files matching the pattern `*.tex.vtd` are of type `:tex.vtd` (Basis of a  $\text{\TeX}$  formatted document). They may be used whenever a rule calls for a supertype of `:tex.vtd`:

Type	Description
<code>:vtd</code>	Virtual target definition file

They may be transformed to any of the following:

`:rootName`, `:rootFileName` (Base name (no `.tex` suffix) of the root file, Simple name of the  $\text{\TeX}$  file to be used as the root) using the following parameters:

`+texroot`  
Name of the  $\text{\TeX}$  document root

Return the file name and base name of the root document. If there is only one document, its name is used. Otherwise, the root document should be specified using `+texroot`.

`:texsp` (Search path for the document basis) with additional source of type `:dir` (Directory of a file) using the following parameters:

`+texsearch`  
Directories to be searched first (may be given more than once)

Return a file containing directories, one per line, to search for  $\text{\TeX}$  inputs. This includes the directory containing the virtual inputs, as well as any path specified by `+texsearch`. If unspecified, the original source directory for the input file is used.

`:dvi`, `:texpdf`, `:tex.log`, `:citations`, `:indexntry`, `:texauxout` (Device independent format file, PDF created from  $\text{\TeX}$  or  $\text{\LaTeX}$ ,  $\text{\TeX}$  log file, Citation and bibliographic data commands, Indexentry commands with page numbers, Auxiliary files potentially modified by  $\text{\TeX}$ ) with additional source of type `:fmtcmd` (Formatting command) and additional source of type `:rootName` (Base name (no `.tex` suffix) of the root file) and additional source of type `:texsp` (Search path for the document basis). Run  $\text{\TeX}$  on inputs until auxiliary files settle. The actual command to run is obtained by the `:fmtcmd` derivation. The `:dvi` and `:texpdf` outputs are optional, and depend on the formatter type (`+usepf`).

### E.19.2 Bibliographic data file

Files matching the pattern `*.bib` are of type `:bib` (Bibliographic data file). They may be used whenever a rule calls for a supertype of `:bib`:

Type	Description
<code>:FILE</code>	A file
<code>:OBJECT</code>	Any object
<code>:try.co</code>	Co if possible

As the type `:FILE` (A file), they may be transformed to any of the following:

`:targets_inc` (Directly nested target definitions). See section E.2 (odin).

`:targets` (Target definitions from an Odinfile). See section E.2 (odin).

`:nested_hooks`, `:hook_values.of` (Nested hooks, Odinfile containing hook values). See section E.2 (odin).  
`:expand_hooks` (Expand hooks). See section E.2 (odin).  
`:filename` (Filename of a file). See section E.2 (odin).  
`:source_names` (Source components of a system model). See section E.2 (odin).  
`:operation_names` (Operation components of a system model). See section E.2 (odin).  
`:c_inc.view_desc` (View description of C-style included files). See section E.5 (cc).  
`:cpp` (Output from cpp pre-processor). See section E.6 (cpp).  
`:cxx_inc.view_desc` (View description of C++ style included files). See section E.7 (cxx).  
`:idl_inc.view_desc` (View description of IDL-style included files). See section E.14 (idl).  
`:boot_name` (Name of the boot-strap file). See section E.17 (misc).  
`:diff` (Differences between two files). See section E.17 (misc).  
`:rcp` (Archive). See section E.17 (misc).  
`:so_inc.spec` (Names of Nroff-style included files). See section E.18 (roff).  
`:eqn` (Output from eqn). See section E.18 (roff).  
`:run` (Execution of a command line). See section E.21 (run).  
`:stdout`, `:output` (Standard output from a test run, Output files from a test run). See section E.21 (run).  
`:run.lua` (Execution of Lua code). See section E.21 (run).  
`:lua_stdout`, `:lua_output` (Standard output from running Lua code, Output files from running Lua code).  
 See section E.21 (run).  
`:regsub` (Output from regsub substitution). See section E.22 (regsub).

As the type `:OBJECT` (Any object), they may be transformed to any of the following:

`:targets_ptr` (Pointer to the targets file). See section E.2 (odin).  
`:copy` (Copy). See section E.2 (odin).  
`:copy_cmd` (Copy command). See section E.2 (odin).  
`:err` (Errors). See section E.2 (odin).  
`:warn` (Warnings). See section E.2 (odin).  
`:dir` (Directory of a file). See section E.2 (odin).  
`:depend` (Source dependencies). See section E.2 (odin).  
`:drv_help` (Applicable derivation types). See section E.2 (odin).  
`:prm_help` (Applicable parameter types). See section E.2 (odin).  
`:redo` (Redo this object step). See section E.17 (misc).

### E.19.3 Bibliographic style file

Files matching the pattern `*.bst` are of type `:bst` (Bibliographic style file). They may be used whenever a rule calls for a supertype of `:bst`:

Type	Description
<code>:FILE</code>	A file
<code>:OBJECT</code>	Any object
<code>:try.co</code>	Co if possible

As the type `:FILE` (A file), they may be transformed to any of the following:

`:targets_inc` (Directly nested target definitions). See section E.2 (odin).  
`:targets` (Target definitions from an Odinfile). See section E.2 (odin).  
`:nested_hooks`, `:hook_values.of` (Nested hooks, Odinfile containing hook values). See section E.2 (odin).  
`:expand_hooks` (Expand hooks). See section E.2 (odin).  
`:filename` (Filename of a file). See section E.2 (odin).  
`:source_names` (Source components of a system model). See section E.2 (odin).  
`:operation_names` (Operation components of a system model). See section E.2 (odin).  
`:c_inc.view_desc` (View description of C-style included files). See section E.5 (cc).  
`:cpp` (Output from cpp pre-processor). See section E.6 (cpp).  
`:cxx_inc.view_desc` (View description of C++ style included files). See section E.7 (cxx).  
`:idl_inc.view_desc` (View description of IDL-style included files). See section E.14 (idl).  
`:boot_name` (Name of the boot-strap file). See section E.17 (misc).  
`:diff` (Differences between two files). See section E.17 (misc).  
`:rcp` (Archive). See section E.17 (misc).  
`:so_inc.spec` (Names of Nroff-style included files). See section E.18 (roff).  
`:eqn` (Output from eqn). See section E.18 (roff).  
`:run` (Execution of a command line). See section E.21 (run).  
`:stdout`, `:output` (Standard output from a test run, Output files from a test run). See section E.21 (run).  
`:run.lua` (Execution of Lua code). See section E.21 (run).  
`:lua_stdout`, `:lua_output` (Standard output from running Lua code, Output files from running Lua code). See section E.21 (run).  
`:regsub` (Output from regsub substitution). See section E.22 (regsub).

As the type `:OBJECT` (Any object), they may be transformed to any of the following:

`:targets_ptr` (Pointer to the targets file). See section E.2 (odin).  
`:copy` (Copy). See section E.2 (odin).  
`:copy_cmd` (Copy command). See section E.2 (odin).

`:err` (Errors). See section E.2 (odin).  
`:warn` (Warnings). See section E.2 (odin).  
`:dir` (Directory of a file). See section E.2 (odin).  
`:depend` (Source dependencies). See section E.2 (odin).  
`:drv_help` (Applicable derivation types). See section E.2 (odin).  
`:prm_help` (Applicable parameter types). See section E.2 (odin).  
`:redo` (Redo this object step). See section E.17 (misc).

## E.19.4 Formatting style file

Files matching the pattern ‘`*.cls`’ are of type `:cls` (Formatting style file). They may be used whenever a rule calls for a supertype of `:cls`:

Type	Description
<code>:FILE</code>	A file
<code>:OBJECT</code>	Any object
<code>:try.co</code>	Co if possible

As the type `:FILE` (A file), they may be transformed to any of the following:

`:targets_inc` (Directly nested target definitions). See section E.2 (odin).  
`:targets` (Target definitions from an Odinfile). See section E.2 (odin).  
`:nested_hooks, :hook_values.of` (Nested hooks, Odinfile containing hook values). See section E.2 (odin).  
`:expand_hooks` (Expand hooks). See section E.2 (odin).  
`:filename` (Filename of a file). See section E.2 (odin).  
`:source_names` (Source components of a system model). See section E.2 (odin).  
`:operation_names` (Operation components of a system model). See section E.2 (odin).  
`:c_inc.view_desc` (View description of C-style included files). See section E.5 (cc).  
`:cpp` (Output from cpp pre-processor). See section E.6 (cpp).  
`:cxx_inc.view_desc` (View description of C++ style included files). See section E.7 (cxx).  
`:idl_inc.view_desc` (View description of IDL-style included files). See section E.14 (idl).  
`:boot_name` (Name of the boot-strap file). See section E.17 (misc).  
`:diff` (Differences between two files). See section E.17 (misc).  
`:rcp` (Archive). See section E.17 (misc).  
`:so_inc_spec` (Names of Nroff-style included files). See section E.18 (roff).  
`:eqn` (Output from eqn). See section E.18 (roff).  
`:run` (Execution of a command line). See section E.21 (run).  
`:stdout, :output` (Standard output from a test run, Output files from a test run). See section E.21 (run).

`:run.lua` (Execution of Lua code). See section E.21 (run).

`:lua_stdout`, `:lua_output` (Standard output from running Lua code, Output files from running Lua code). See section E.21 (run).

`:regsub` (Output from regsub substitution). See section E.22 (regsub).

As the type `:OBJECT` (Any object), they may be transformed to any of the following:

`:targets_ptr` (Pointer to the targets file). See section E.2 (odin).

`:copy` (Copy). See section E.2 (odin).

`:copy_cmd` (Copy command). See section E.2 (odin).

`:err` (Errors). See section E.2 (odin).

`:warn` (Warnings). See section E.2 (odin).

`:dir` (Directory of a file). See section E.2 (odin).

`:depend` (Source dependencies). See section E.2 (odin).

`:drv_help` (Applicable derivation types). See section E.2 (odin).

`:prm_help` (Applicable parameter types). See section E.2 (odin).

`:redo` (Redo this object step). See section E.17 (misc).

## E.19.5 Portable Document Format file

Files matching the pattern `*.pdf` are of type `:pdf` (Portable Document Format file). They may be used whenever a rule calls for a supertype of `:pdf`:

Type	Description
<code>:FILE</code>	A file
<code>:OBJECT</code>	Any object
<code>:try.co</code>	Co if possible

As the type `:FILE` (A file), they may be transformed to any of the following:

`:targets_inc` (Directly nested target definitions). See section E.2 (odin).

`:targets` (Target definitions from an Odinfile). See section E.2 (odin).

`:nested_hooks`, `:hook_values.of` (Nested hooks, Odinfile containing hook values). See section E.2 (odin).

`:expand_hooks` (Expand hooks). See section E.2 (odin).

`:filename` (Filename of a file). See section E.2 (odin).

`:source_names` (Source components of a system model). See section E.2 (odin).

`:operation_names` (Operation components of a system model). See section E.2 (odin).

`:c_inc.view_desc` (View description of C-style included files). See section E.5 (cc).

`:cpp` (Output from cpp pre-processor). See section E.6 (cpp).

`:cxx_inc.view_desc` (View description of C++ style included files). See section E.7 (cxx).

`:idl_inc.view_desc` (View description of IDL-style included files). See section E.14 (idl).

`:boot_name` (Name of the boot-strap file). See section E.17 (misc).

`:diff` (Differences between two files). See section E.17 (misc).

`:rcp` (Archive). See section E.17 (misc).

`:so_inc_spec` (Names of Nroff-style included files). See section E.18 (roff).

`:eqn` (Output from eqn). See section E.18 (roff).

`:run` (Execution of a command line). See section E.21 (run).

`:stdout`, `:output` (Standard output from a test run, Output files from a test run). See section E.21 (run).

`:run.lua` (Execution of Lua code). See section E.21 (run).

`:lua_stdout`, `:lua_output` (Standard output from running Lua code, Output files from running Lua code). See section E.21 (run).

`:regsub` (Output from regsub substitution). See section E.22 (regsub).

As the type `:OBJECT` (Any object), they may be transformed to any of the following:

`:targets_ptr` (Pointer to the targets file). See section E.2 (odin).

`:copy` (Copy). See section E.2 (odin).

`:copy_cmd` (Copy command). See section E.2 (odin).

`:err` (Errors). See section E.2 (odin).

`:warn` (Warnings). See section E.2 (odin).

`:dir` (Directory of a file). See section E.2 (odin).

`:depend` (Source dependencies). See section E.2 (odin).

`:drv_help` (Applicable derivation types). See section E.2 (odin).

`:prm_help` (Applicable parameter types). See section E.2 (odin).

`:redo` (Redo this object step). See section E.17 (misc).

### E.19.6 Formatting style file

Files matching the pattern `*.sty` are of type `:sty` (Formatting style file). They may be used whenever a rule calls for a supertype of `:sty`:

Type	Description
<code>:FILE</code>	A file
<code>:OBJECT</code>	Any object
<code>:try.co</code>	Co if possible

As the type `:FILE` (A file), they may be transformed to any of the following:

`:targets_inc` (Directly nested target definitions). See section E.2 (odin).

`:targets` (Target definitions from an Odinfile). See section E.2 (odin).

`:nested_hooks`, `:hook_values.of` (Nested hooks, Odinfile containing hook values). See section E.2 (odin).  
`:expand_hooks` (Expand hooks). See section E.2 (odin).  
`:filename` (Filename of a file). See section E.2 (odin).  
`:source_names` (Source components of a system model). See section E.2 (odin).  
`:operation_names` (Operation components of a system model). See section E.2 (odin).  
`:c_inc.view_desc` (View description of C-style included files). See section E.5 (cc).  
`:cpp` (Output from cpp pre-processor). See section E.6 (cpp).  
`:cxx_inc.view_desc` (View description of C++ style included files). See section E.7 (cxx).  
`:idl_inc.view_desc` (View description of IDL-style included files). See section E.14 (idl).  
`:boot_name` (Name of the boot-strap file). See section E.17 (misc).  
`:diff` (Differences between two files). See section E.17 (misc).  
`:rcp` (Archive). See section E.17 (misc).  
`:so_inc.spec` (Names of Nroff-style included files). See section E.18 (roff).  
`:eqn` (Output from eqn). See section E.18 (roff).  
`:run` (Execution of a command line). See section E.21 (run).  
`:stdout`, `:output` (Standard output from a test run, Output files from a test run). See section E.21 (run).  
`:run.lua` (Execution of Lua code). See section E.21 (run).  
`:lua_stdout`, `:lua_output` (Standard output from running Lua code, Output files from running Lua code).  
 See section E.21 (run).  
`:regsub` (Output from regsub substitution). See section E.22 (regsub).

As the type `:OBJECT` (Any object), they may be transformed to any of the following:

`:targets_ptr` (Pointer to the targets file). See section E.2 (odin).  
`:copy` (Copy). See section E.2 (odin).  
`:copy_cmd` (Copy command). See section E.2 (odin).  
`:err` (Errors). See section E.2 (odin).  
`:warn` (Warnings). See section E.2 (odin).  
`:dir` (Directory of a file). See section E.2 (odin).  
`:depend` (Source dependencies). See section E.2 (odin).  
`:drv_help` (Applicable derivation types). See section E.2 (odin).  
`:prm_help` (Applicable parameter types). See section E.2 (odin).  
`:redo` (Redo this object step). See section E.17 (misc).

### E.19.7 TeX formatter input

Files matching the pattern ‘\*.tex’ are of type `:tex` (TeX formatter input). They may be used whenever a rule calls for a supertype of `:tex`:

Type	Description
<code>:FILE</code>	A file
<code>:OBJECT</code>	Any object
<code>:try.co</code>	Co if possible

They may be transformed to any of the following:

`:one_tex_vtd` (Singleton set of TeX formatter inputs) with additional source of type `:label` (Label of a file). Return a file listing the input T<sub>E</sub>X file.

`:texnames` (Text file containing local file names). Given a T<sub>E</sub>X input file, return a list of files needed during a run.

As the type `:FILE` (A file), they may be transformed to any of the following:

`:targets_inc` (Directly nested target definitions). See section E.2 (odin).

`:targets` (Target definitions from an Odinfile). See section E.2 (odin).

`:nested_hooks`, `:hook_values.of` (Nested hooks, Odinfile containing hook values). See section E.2 (odin).

`:expand_hooks` (Expand hooks). See section E.2 (odin).

`:filename` (Filename of a file). See section E.2 (odin).

`:source_names` (Source components of a system model). See section E.2 (odin).

`:operation_names` (Operation components of a system model). See section E.2 (odin).

`:c_inc.view_desc` (View description of C-style included files). See section E.5 (cc).

`:cpp` (Output from cpp pre-processor). See section E.6 (cpp).

`:cxx_inc.view_desc` (View description of C++ style included files). See section E.7 (cxx).

`:idl_inc.view_desc` (View description of IDL-style included files). See section E.14 (idl).

`:boot_name` (Name of the boot-strap file). See section E.17 (misc).

`:diff` (Differences between two files). See section E.17 (misc).

`:rcp` (Archive). See section E.17 (misc).

`:so_inc.spec` (Names of Nroff-style included files). See section E.18 (roff).

`:eqn` (Output from eqn). See section E.18 (roff).

`:run` (Execution of a command line). See section E.21 (run).

`:stdout`, `:output` (Standard output from a test run, Output files from a test run). See section E.21 (run).

`:run.lua` (Execution of Lua code). See section E.21 (run).

`:lua_stdout`, `:lua_output` (Standard output from running Lua code, Output files from running Lua code). See section E.21 (run).

`:regsub` (Output from regsub substitution). See section E.22 (regsub).



As the type `:OBJECT` (Any object), they may be transformed to any of the following:

`:targets_ptr` (Pointer to the targets file). See section E.2 (odin).

`:copy` (Copy). See section E.2 (odin).

`:copy_cmd` (Copy command). See section E.2 (odin).

`:err` (Errors). See section E.2 (odin).

`:warn` (Warnings). See section E.2 (odin).

`:dir` (Directory of a file). See section E.2 (odin).

`:depend` (Source dependencies). See section E.2 (odin).

`:drv_help` (Applicable derivation types). See section E.2 (odin).

`:prm_help` (Applicable parameter types). See section E.2 (odin).

`:redo` (Redo this object step). See section E.17 (misc).

### E.19.8 Postscript file

Files matching the pattern `*.ps` are of type `:ps` (Postscript file). They may be used whenever a rule calls for a supertype of `:ps`:

Type	Description
<code>:FILE</code>	A file
<code>:OBJECT</code>	Any object
<code>:try.co</code>	Co if possible

As the type `:FILE` (A file), they may be transformed to any of the following:

`:targets_inc` (Directly nested target definitions). See section E.2 (odin).

`:targets` (Target definitions from an Odinfile). See section E.2 (odin).

`:nested_hooks`, `:hook_values.of` (Nested hooks, Odinfile containing hook values). See section E.2 (odin).

`:expand_hooks` (Expand hooks). See section E.2 (odin).

`:filename` (Filename of a file). See section E.2 (odin).

`:source_names` (Source components of a system model). See section E.2 (odin).

`:operation_names` (Operation components of a system model). See section E.2 (odin).

`:c_inc.view_desc` (View description of C-style included files). See section E.5 (cc).

`:cpp` (Output from cpp pre-processor). See section E.6 (cpp).

`:cxx_inc.view_desc` (View description of C++ style included files). See section E.7 (cxx).

`:idl_inc.view_desc` (View description of IDL-style included files). See section E.14 (idl).

`:boot_name` (Name of the boot-strap file). See section E.17 (misc).

`:diff` (Differences between two files). See section E.17 (misc).

`:rcp` (Archive). See section E.17 (misc).

**:so\_inc\_spec** (Names of Nroff-style included files). See section E.18 (roff).

**:eqn** (Output from eqn). See section E.18 (roff).

**:run** (Execution of a command line). See section E.21 (run).

**:stdout, :output** (Standard output from a test run, Output files from a test run). See section E.21 (run).

**:run.lua** (Execution of Lua code). See section E.21 (run).

**:lua\_stdout, :lua\_output** (Standard output from running Lua code, Output files from running Lua code). See section E.21 (run).

**:regsub** (Output from regsub substitution). See section E.22 (regsub).

As the type **:OBJECT** (Any object), they may be transformed to any of the following:

**:targets\_ptr** (Pointer to the targets file). See section E.2 (odin).

**:copy** (Copy). See section E.2 (odin).

**:copy\_cmd** (Copy command). See section E.2 (odin).

**:err** (Errors). See section E.2 (odin).

**:warn** (Warnings). See section E.2 (odin).

**:dir** (Directory of a file). See section E.2 (odin).

**:depend** (Source dependencies). See section E.2 (odin).

**:drv\_help** (Applicable derivation types). See section E.2 (odin).

**:prm\_help** (Applicable parameter types). See section E.2 (odin).

**:redo** (Redo this object step). See section E.17 (misc).

### E.19.9 Other transformations

In addition, the following transformations are defined within this package. They are used in the other packages as indicated.

**:texclose** (Names of all files transitively referenced) from source of type **:texfiles** (Accessible files referenced). Given a list of T<sub>E</sub>X files, list all files needed during a run. This is for automatic dependency scanning.

**:texbasis** (Document basis) from source of type **:rootFileName** (Simple name of the TeX file to be used as the root). Given a T<sub>E</sub>X root file name, list all files needed during a run. This is for automatic dependency scanning.

**:tex.view\_desc** (View descriptor for basis files) from source of type **:texnames** (Text file containing local file names) using the following parameters:

**+texsppt**

Parameter type for search path information

Given a list of T<sub>E</sub>X input files, produce a view descriptor. This is for automatic dependency scanning.

**:texfiles** (Accessible files referenced) from source of type **:tex.view\_desc** (View descriptor for basis files). Collapse view descriptor to actual present files.

**:fmtcmd** (Formatting command) from source of type **:rootFileName** (Simple name of the TeX file to be used as the root) using the following parameters:

**+tex**  
Use plain TeX

**+latex**  
Use LaTeX

**+usepdf**  
Produce PDF code

Given a TeX input file, determine which command to run for output. If **+usepdf** is set, PDF is produced directly from input using pdf(la)tex. If **+tex** is set, the input is plain TeX. If **+latex** is set, the input is LaTeX. Otherwise, selection is based on presence of the LaTeX command `\documentstyle` or `\documentclass`.

**:tex.bbl** (TeX bibliographic reference file) from source of type **:citations** (Citation and bibliographic data commands) with additional source of type **:texsp** (Search path for the document basis). Given TeX input, create a bibliography file if needed, using **bibtex**.

**:tex.ind**, **:tex.ilg** (TeX index file, Messages from makeindex) from source of type **:indexntry** (Indexentry commands with page numbers) using the following parameters:

**+index.flags**  
Options for makeindex (may be given more than once)

Given TeX input, create indices using **makeindex**, if needed.

**:texauxin.vtd** (Auxiliary files input to the formatting step) from source of type **:rootName** (Base name (no .tex suffix) of the root file) with additional source of type **:texauxout** (Auxiliary files potentially modified by TeX) and additional source of type **:tex.bbl** (TeX bibliographic reference file) and additional source of type **:tex.ind** (TeX index file). Return file listing all auxiliary inputs associated with a TeX run.

**:texps** (PostScript created from TeX or LaTeX) from source of type **:dvi** (Device independent format file) with additional source of type **:texsp** (Search path for the document basis) using the following parameters:

**+dvips.flags**  
Options for dvips (may be given more than once)

After running TeX, run **dvips** to convert the DVI output to PostScript.

**:refpdf** (Reference to a PDF file) from source of type **:tex.vtd** (Basis of a TeX formatted document). A back path to creating PDF using pdf(la)tex directly.

## E.20 dbx

The **dbx** package is at version 1.14.0. This package supports running a debugger on executables.

### E.20.1 Other transformations

In addition, the following transformations are defined within this package. They are used in the other packages as indicated.

**:dbx** (Symbolic debugger run) from source of type **:exe** (Executable binary) using the following parameters:

**+core**  
Core dump  
**+gnu**  
Use gnu tools

A script is created to run the debugger dbx/gdb (+gnu). A core file may optionally be specified with +core. It is intended to be used interactively with the ! suffix to execute the script.

## E.21 run

The run package is at version 1.14.0. This package supports execution of simple command lines.

### E.21.1 Other transformations

In addition, the following transformations are defined within this package. They are used in the other packages as indicated.

**:run** (Execution of a command line) from source of type **:FILE** (A file) using the following parameters:

**+cmd**  
Host command (may be given more than once)

This creates and runs a script to execute comands given by +cmd, in the order given, in the directory specified as input. The commands are not parsed, so any dependencies must be specified using +needs.

**:stdout**, **:output** (Standard output from a test run, Output files from a test run) from source of type **:FILE** (A file) using the following parameters:

**+cmd**  
Host command (may be given more than once)

This executes the given +cmd on the input file, and returns its standard output as :stdout and any output files in the virtual directory :output, assuming that output is generated in the current working directory. The output file type is retained from the input file. If no +cmd is specified, the result is a copy of the input file.

**:run.lua** (Execution of Lua code) from source of type **:FILE** (A file) using the following parameters:

**+cmd**  
Host command (may be given more than once)

This creates and runs a Lua script to execute comands given by +cmd, in the order given, in the directory specified as input. The commands are not parsed, so any dependencies must be specified using +needs.

**:lua.stdout**, **:lua.output** (Standard output from running Lua code, Output files from running Lua code) from source of type **:FILE** (A file) using the following parameters:

**+cmd**  
Host command (may be given more than once)

**+cmdfile**  
Host command file (may be given more than once)

This executes the given +cmd Lua commands on the input file, and returns any standard output as :stdout and any output files in the virtual directory :output, assuming that output is generated in the current working directory. The output file type is retained from the input file. If no +cmd is specified, the result is a copy of the input file.

Within the Lua code, only io.write() will write to the results. If other commands are used, they should write to a separate file, and the result :output/<other\_file> should be used instead of :lua\_stdout. Otherwise, :lua\_stdout will probably be overwritten on exit.

## E.22 rebsub

The **rebsub** package is at version 1.1.0. This package provides a simple egrep and sed replacement.

### E.22.1 Regsub substitution input

Files matching the pattern ‘\*,re’ are of type **:rebsub\_in** (Regsub substitution input). They may be used whenever a rule calls for a supertype of **:rebsub\_in**:

Type	Description
<b>:FILE</b>	A file
<b>:OBJECT</b>	Any object
<b>:try.co</b>	Co if possible

They may be transformed to any of the following:

**:gnc.rebsub** (Output from rebsub substitution) using the following parameters:

**+match**  
Regular expression to match (may be given more than once)

**+hide\_match**  
Instead of showing matches, hide them

**+subst**  
Regular expression substitution (may be given more than once)

**+subst\_only**  
Only include lines where substitutions were made

**+prefix**  
Prefix to apply to all lines

**+suffix**  
Suffix to apply to all lines

Filters a file, showing (or hiding, if +hide\_match is given) only lines that match the extended regular expression in +match (if given). The type of the file is based on source file name matching after stripping the trailing ,re. On each line, a global substitution is made for each +subst parameter. This parameter consists of the search string, followed by a / that is not escaped, followed by the replacement string. In the replacement string, a backslash escapes any character but the digits 0-9 and open curly braces. An unescaped ampersand (&) represents the entire matching text. A backslash followed by a single digit, or a backslash followed by a curly-brace enclosed decimal number, represents

that parenthesis-enclosed subexpression. Subexpression zero is the entire matching text, and the rest numbered by the order of the opening parenthesis. All +subst parameters are applied in the order they are given, so an earlier one may affect a later one.

If the +subst\_only parameter is given, any line which does not cause a substitution will be filtered out as well. This is a shortcut for adding the +subst patterns as +match arguments. Note that actual filtering using +match takes place before +subst substitution.

The +prefix parameter text is prepended to each line, and the +suffix parameter is appended to each line. They are shortcuts to +subst  $\hat{/}$ <text> and +subst \$/<text>, respectively, except that they do not count towards +subst\_only, and only one of each may be given.

As the type :FILE (A file), they may be transformed to any of the following:

:regsub (Output from regsub substitution) using the following parameters:

- +match  
Regular expression to match (may be given more than once)
- +hide\_match  
Instead of showing matches, hide them
- +subst  
Regular expression substitution (may be given more than once)
- +subst\_only  
Only include lines where substitutions were made
- +prefix  
Prefix to apply to all lines
- +suffix  
Suffix to apply to all lines

Filters a file, showing (or hiding, if +hide\_match is given) only lines that match the extended regular expression in +match (if given). On each line, a global substitution is made for each +subst parameter. This parameter consists of the search string, followed by a / that is not escaped, followed by the replacement string. In the replacement string, a backslash escapes any character but the digits 0-9 and open curly braces. An unescaped ampersand (&) represents the entire matching text. A backslash followed by a single digit, or a backslash followed by a curly-brace enclosed decimal number, represents that parenthesis-enclosed subexpression. Subexpression zero is the entire matching text, and the rest numbered by the order of the opening parenthesis. Named subexpressions may be referenced using a name in curly braces rather than a number; the name must start with something other than a digit, and may contain no close-curly-braces. All +subst parameters are applied in the order they are given, so an earlier one may affect a later one.

If the +subst\_only parameter is given, any line which does not cause a substitution will be filtered out as well. This is a shortcut for adding the +subst patterns as +match arguments. Note that actual filtering using +match takes place before +subst substitution.

The +prefix parameter text is prepended to each line, and the +suffix parameter is appended to each line. They are shortcuts to +subst  $\hat{/}$ <text> and +subst \$/<text>, respectively, except that they do not count towards +subst\_only, and only one of each may be given.

:targets\_inc (Directly nested target definitions). See section E.2 (odin).

:targets (Target definitions from an Odinfile). See section E.2 (odin).

:nested\_hooks, :hook\_values.of (Nested hooks, Odinfile containing hook values). See section E.2 (odin).

:expand\_hooks (Expand hooks). See section E.2 (odin).

`:filename` (Filename of a file). See section E.2 (odin).  
`:source_names` (Source components of a system model). See section E.2 (odin).  
`:operation_names` (Operation components of a system model). See section E.2 (odin).  
`:c_inc.view_desc` (View description of C-style included files). See section E.5 (cc).  
`:cpp` (Output from cpp pre-processor). See section E.6 (cpp).  
`:cxx_inc.view_desc` (View description of C++ style included files). See section E.7 (cxx).  
`:idl_inc.view_desc` (View description of IDL-style included files). See section E.14 (idl).  
`:boot_name` (Name of the boot-strap file). See section E.17 (misc).  
`:diff` (Differences between two files). See section E.17 (misc).  
`:rcp` (Archive). See section E.17 (misc).  
`:so_inc_spec` (Names of Nroff-style included files). See section E.18 (roff).  
`:eqn` (Output from eqn). See section E.18 (roff).  
`:run` (Execution of a command line). See section E.21 (run).  
`:stdout`, `:output` (Standard output from a test run, Output files from a test run). See section E.21 (run).  
`:run.lua` (Execution of Lua code). See section E.21 (run).  
`:lua_stdout`, `:lua_output` (Standard output from running Lua code, Output files from running Lua code).  
 See section E.21 (run).

As the type `:OBJECT` (Any object), they may be transformed to any of the following:

`:targets_ptr` (Pointer to the targets file). See section E.2 (odin).  
`:copy` (Copy). See section E.2 (odin).  
`:copy_cmd` (Copy command). See section E.2 (odin).  
`:err` (Errors). See section E.2 (odin).  
`:warn` (Warnings). See section E.2 (odin).  
`:dir` (Directory of a file). See section E.2 (odin).  
`:depend` (Source dependencies). See section E.2 (odin).  
`:drv_help` (Applicable derivation types). See section E.2 (odin).  
`:prm_help` (Applicable parameter types). See section E.2 (odin).  
`:redo` (Redo this object step). See section E.17 (misc).

# Appendix F

## Odin Lua Library

When Lua code is executed directly by Odin, it first defines some utility functions for convenience and portability.

### F.1 GLib

Portability to other operating systems is supported by including the lua-glib package in the table `glib`. See <https://github.com/darktjm/lua-glib> for details. Some of its commonly used functions are aliased globally to shorten scripts:

**pathcat(dir,name, ...)** `glib.build_filename` — add path elements portably

**getcwd()** `glib.get_current_directory` — return current directory

**chdir(dir)** `glib.chdir` — change working directory

**mkdir(dir[, perm])** `glib.mkdir_with_parents` — create directory and any missing intervening directories

**mv(src, dest)** `glib.rename` — file rename

**exists(name)** `glib.exists` — return true if *name* exists in the file system.

**is\_dir(name)** `glib.is_dir` — return true if *name* is a directory or a soft link to a directory.

**is\_file(name)** `glib.is_file` — return true if *name* is a file or a soft link to a file.

**is\_exec(name)** `glib.is_exec` — return true if *name* is a file and executable permission is set for that file.  
Note that on Windows, a heuristic is applied to *name* determine if a file is executable, since otherwise all files would be executable by that logic.

**is\_link(name)** `glib.is_symlink` — return true if *name* is a soft link. Note that pseudo-soft link files (`*.lnk`) on Windows are not covered by this, so it always returns false on Windows.

In addition, light wrappers are available for a few functions:

**getenv(v)** — return the given environment variable's value, or blank if unset (in comparison to `nil` as returned by `apr.env_get`)

**setenv(name, val)** — overwrite env var *name* with *val*

**basename(name[, split ])** `glib.path_get_basename` — get last path element, and split off last dot-separated extension and return separately



**is\_empty(name)** — return true if *name* either does not exist or is an empty file.

**touch(name[, date])** — if *name* does not exist, it is created as an empty file. Otherwise, it sets the modification time to the given parameter (seconds past epoch).

## F.2 File Commands

Some additional file commands not provided by GLib are provided here:

**cp(src, dest[, mode])** — copy file *src* to *dest*. *src* and *dest* may be a file descriptor or a file name. If *dest* is a file name, protection attributes are set to *mode*, if provided. If *src* and *dest* are both file names, and *mode* is not provided, permissions are copied from *src*.

**rm(name)** — recursive delete

**cmp(f1, f2)** — recursively compare two files or directories, returning true if they match. Only contents are compared; differences in file modification times and permissions are ignored. Soft links are resolved before comparison.

**cp\_RpL(src, dest[, plus\_w])** — copy source to destination recursively, copying permissions if possible. If *plus\_w* is true, user write permission is added to all destination items.

**ln(src, dest)** — make a link from source to destination. If possible, a soft link is made. Otherwise, if possible, a hard link is made. If no link can be made, this is the same as **cp\_RpL**.

**cat(f)** — print *f* on standard output. *f* may be either the name of a file or a Lua file descriptor. The entire file is not read into memory at once, so this can be safely used with large files.

**append\_line(f, msg)** — append *msg* and a newline to the file *f*. *f* may be either the name of a file or a Lua file descriptor.

**split\_filename(f)** — the inverse of **pathcat**. Returns the root part (if present), followed by a table with each path element, in order.

**build\_path(...)** — create a search path using the system separator out arguments. Arguments may be a single table of strings or zero or more strings.

**split\_path(p)** — split a search path into its elements; unlike some such utilities, blank elements are returned as well. Elements are returned as a table of strings.

**glob\_to\_regex(g)** — turn a glob-style pattern to a regular expression. This supports `?`, `*`, character classes in square brackets, and backslash to escape the special meaning of these characters. None of the above patterns can be used to match directory separators. It also supports curly braces, but unlike the shell, only matching alternatives are returned (e.g. `{a,b,c*}` will match `a` or `b` or `c*`, but in the shell it would return `a` and `b` regardless of whether or not `a` or `b` match). Like the shell, mismatched curly brackets and curly brackets without unescaped commas are ignored. Directory separators are supported within curly braces.

**shell\_expand\_braces** — expand curly braces the way a shell would. Returns a table with all entries produced by expanding braces. For example, `{a,b,c*}` would return `{'a', 'b', 'c*'}`.

**dir\_regex([dir,] regex)** — Return an iterator which produces names of directory entries in *dir* matching *regex*. If *dir* is not specified, search the current directory. The *regex* parameter may either be a regular expression string or a compiled regular expression (from **glib.regex\_new**). The special directory elements `'.'` and `'..'` cannot be matched.

**dir\_glob([dir,] glob)** — Return an iterator which produces names of directory entries in *dir* matching glob-style pattern *glob*. If *dir* is not specified, search the current directory. The special directory elements '.' and '..' cannot be matched.

**relative\_glob([dir,] glob)** — Return an iterator which produces relative paths of directory entries which match the glob-style pattern *glob*. Directory separators in *glob* are followed. No part of the pattern may match the special directory entries '.' and '..'.

**glob(pat)** — Return an iterator which produces directory entries which match the glob-style pattern *pat*. Absolute paths are permitted, so long as the absolute prefix contains no pattern elements other than curly braces/commas. Relative patterns are always matched relative to the current working directory. The special path elements '.' and '..' are not matched, unless they appear in the path before the first special character (after curly brace expansion). For example, '../\*.[ch]' works, but '\*/..' does not.

## F.3 Odin Support Utilities

The following functions are for common things done in Odin scripts:

**odin\_error(msg[, ret])** — append *msg* to `ERRORS` and, if *ret* is provided, exit the script with *ret* as the result.

**ODINVERBOSE** — true if `ODINVERBOSE` is set in environment

**ODINRBSHOST** — shortcut for `getenv('ODINRBSHOST')`

**odin\_log(msg)** — print *msg* if `ODINVERBOSE` is set in the environment, and prepend `ODINRBSHOST` if it is set in the environment. This is intended for printing the command to execute before executing it.

**trim(s)** — trim leading and trailing whitespace from *s*. This is intended for use with command strings with embedded odin-expressions, since they are technically separate words, and therefore separated by whitespace (usually newlines):

```
() +cmd='...files = trim[['(%files)']]...' :lua_stdout
```

**odin\_quote(s)** — quote *s* as a single Odin token

**odin\_quote\_file(s[, virt])** — quote *s* as an Odin path; that is, slashes remain unquoted. If *virt* is true, percent signs remain unquoted as well.

**runcmd(cmd[, args])** — execute a system command. *cmd* is parsed like a shell command. Array elements of *args* are appended to arguments split out from *cmd* as if they were fully quoted arguments. *args* may also have named options:

**stdin** — set to the name of a file for standard input to the command; otherwise standard input comes from an empty file.

**stdout** — set to true to retain standard output as the file `MESSAGES`; set to a string file name to retain standard output in a different file. If false or unset, standard output from the command is gathered and then printed to Lua standard output.

**ignret** — set to true to ignore the return value from the command. Otherwise, a non-zero return value indicates an error.

**chdir** — set to a directory path name to execute the command with a different working directory than the current Lua directory.

All output to standard error is captured in a file called **WARNINGS**. If the command fails, standard output and standard error are concatenated to produce **ERRORS**, and neither **WARNINGS** nor the standard output file are produced. In addition, a message indicating failure of the command is appended to **ERRORS**, both to inform the user what command failed and to ensure that the file is non-empty. If no standard error output is produced, **WARNINGS** is removed as well.

## F.4 Configuration Support

The following support tasks traditionally done by a configuration script:

**parse\_ldflags(f)** — Parse *f* as standard UNIX C linker flags, and return as Odin **+lib\_sp**, **+lib**, and **+ld\_flags** options.

**parse\_cflags(f)** — Parse *f* as standard UNIX C compiler flags, and return as Odin **+inc\_sp**, **+define**, and **+cc\_flags** options.

**pkgconfig** — The expected name of the **pkg-config** command (**pkg-config** unless **PKGCONFIG** is set in environment). Note that this can be set to something different before calling the functions below to (temporarily) use a different command that works just like **pkg-config**.

**get\_pkgconfig(arg, pkg[, vermin][, vermax])** — Obtain results from running **pkg-config** with the given *\*arg\**, on the given *\*pkg\** whose version must be at least *\*vermin\** (if supplied) and/or at most *\*vermax\** (if supplied). On error, nil and an error message are returned. The *\*arg\** should normally be something like **--cflags** or **--libs**.

**pkgconfig\_libs** The **pkgconfig** option to obtain linker flags (**--libs**). Note that this may be overridden if a different set of linker flags is needed (for example if **pkgconfig** was overridden).

**pkg\_libs(pkg[, vermin][, vermax])** — Return the libraries from **pkg-config --libs** in Odin option format.

**pkgconfig\_libs** The **pkgconfig** option to obtain C compiler flags (**--cflags**) Note that this may be overridden if a different set of compiler flags is needed (for example if **pkgconfig** was overridden).

**pkg\_cflags(pkg[, vermin][, vermax])** — Return the C compiler flags and libraries from **pkg-config --libs --cflags** in Odin option format. Two separate runs are made to separate generic linker flags (**+ld\_flags**) from generic compiler flags (**+cc\_flags**) properly. format.

## F.5 Miscellaneous

The following defy categorization:

**words(f)** — replacement for shell's **for x in `cat f`**, in that the file *f* (either a file name or a Lua file descriptor) is read, and an iterator is returned which iterates over whitespace-separated words.

**wholefile(f)** — replacement for shell's **`cat f`** as a command-line argument, equivalent to concatenating all of **words(f)** into a single string, each word separated by a single space.

**unpack(table)** — an alias for **table.unpack()** in Lua 5.2.

# Bibliography

- [1] *ClearCase Reference Manual*. Atria Software, 1993.
- [2] B. Erickson and J.F. Pellegrin. Build - a software construction tool. *ATT Bell Labs Technical Journal*, 63:1049–1059, 1984.
- [3] S. I. Feldman. MAKE - a program for maintaining computer programs. *Software - Practice and Experience*, 9:255–265, 1979.
- [4] G. Fowler. A case for make. *Software - Practice and Experience*, 20:35–46, June 1990.
- [5] A. G. Hume. MK: a successor to make. In *USENIX Summer Conference Proceedings*, pages 445–457, 1987.
- [6] Richard M. Stallman and Roland McGrath. *GNU Make: A Program for Directing Recompilation*. Free Software Foundation, 1993.