

# Make: Dependency Manager

Version 6.12

January 26, 2018

The [make](#) library provides a Racket version of the popular make utility. Its syntax is intended to imitate the syntax of make, only in Racket.

# Contents

<b>1</b>	<b>Overview</b>	<b>3</b>
<b>2</b>	<b>Make from Dependencies</b>	<b>5</b>
2.1	Signature . . . . .	6
2.2	Unit . . . . .	6
<b>3</b>	<b>Building Native-Code Extensions</b>	<b>7</b>
<b>4</b>	<b>Making Collections</b>	<b>10</b>
4.1	Signature . . . . .	10
4.2	Unit . . . . .	10

# 1 Overview

If you are already familiar with `make`, skip to the precise details of the `make` library in §2 “Make from Dependencies”. This section contains a brief overview of `make` for everyone else.

When you use `make`, the idea is that you explain how to generate files in a project from a collection of source files that go through several stages of processing.

For example, say that you are writing a project that has three input files (which you create and maintain) called `"a.input"`, `"b.input"`, and `"c.input"`. Further, there are two stages of processing: first you run a particular tool `make-output` that takes an input file and produces an output file, and then you combine the input files into a single file using `combine-files`. Using `make`, you might describe this as:

```
a.output: a.input
          make-output a.input a.output
b.output: b.input
          make-output b.input b.output
c.output: c.input
          make-output c.input c.output
total: a.output b.output c.output
       combine-files a.output b.output c.output
```

Once you’ve put this description in a file called `"Makefile"` you can issue the command:

```
make total
```

to build your entire project. The `"Makefile"` consists of several rules that tell `make` how to create each piece of your project. For example, the rule that is specified in the first two lines say that `"a.output"` depends on `"a.input"` and the command for making `"a.output"` from `"a.input"` is

```
make-output a.input a.output
```

The main feature of `make` is that it uses the time stamps of files to determine when a certain step is necessary. The `make` utility uses existing programs to build your project — each rule has a shell command line.

The `make` library provides similar functionality, except that the description is in Racket, and the steps that are needed to build target files are implemented as Racket functions.

Here’s a Racket program that is equivalent to the above:

```
(require make)

(define (make-output in out)
```

If you want to build Racket modules with automatic dependency tracking, just use `raco make` as described in *raco: Racket Command-Line Tools*.

```

....)

(define (combine-files . args)
  ....)

(make
  (("a.output" ("a.input") (make-output "a.input" "a.output"))
   ("b.output" ("b.input") (make-output "b.input" "b.output"))
   ("c.output" ("c.input") (make-output "c.input" "c.output"))
   ("total" ("a.output" "b.output" "c.output")
            (combine-files "a.output" "b.output" "c.output"))))

```

If you were to fill in the ellipses above with calls to `system`, you'd have the exact same functionality as the original "Makefile". In addition, you can use `make/proc` to abstract over the various lines. For example, the "a.output", "b.output", and "c.output" lines are very similar so you can write the code that generates those lines:

```

(require make)

(define (make-output in out)
  ....)

(define (combine-files . args)
  ....)

(define files '("a" "b" "c"))
(define inputs (map (lambda (f) (string-append f ".input")) files))
(define outputs (map (lambda (f) (string-append f ".output")) files))

(define (line file)
  (let ([i (string-append file ".input")]
        [o (string-append file ".output")])
    `(,o ,(i))
    (list o (list i) (lambda () (make-output o i)))))

(make/proc
  `(,@(map (lambda (i o) `(o ,(i) ,(lambda () (make-output i o))))
          inputs outputs)
    ("total" ,outputs ,(lambda () (apply combine-
files outputs)))))

```

## 2 Make from Dependencies

```
(require make)      package: make

(make ((target-expr (depend-expr ...)
                  command-expr ...)
      ...
      argv-expr))
```

Expands to

```
(make/proc
  (list (list target-expr (list depend-expr ...)
              (lambda () command-expr ...))
        ...
  argv-expr)
```

```
(make/proc spec argv) → void?
spec : (listof
        (cons/c (or/c path-string? (listof path-string?))
                 (cons/c (listof path-string?)
                          (or/c null?
                                (list/c (-> any))))))
argv : (or/c string? (vectorof string?) (listof string?))
```

Performs a make according to `spec` and using `argv` as command-line arguments selecting one or more targets.

Each element of the `spec` list is a target. A target element that starts with a list of strings is the same as multiple elements, one for each string. The second element of each target is a list of dependencies, and the third element (if any) of a target is the optional command thunk.

To make a target, `make/proc` is first called recursively on each of the target's dependencies. If a target is not in `spec` and it exists as a file, then the target is considered made. If a target's modification date is older than any of its dependencies' modification dates, the corresponding command thunk is called. If the dependency has no command thunk then no action is taken; such a target is useful for triggering the make of other targets (i.e., the dependencies).

While running a command thunk, `make/proc` catches exceptions and wraps them in an `exn:fail:make` structure, the raises the resulting structure.}

```
(struct exn:fail:make exn:fail (targets orig-exn)
  #:extra-constructor-name make-exn:fail:make)
targets : (listof path-string?)
orig-exn : any/c
```

The `targets` field is a list of strings naming the target(s), and the `orig-exn` field is the original raised value.

```
(make-print-checking) → boolean?  
(make-print-checking on?) → void?  
  on? : any/c
```

A parameter that controls whether `make/proc` prints a message when making a target. The default is `#t`.

```
(make-print-dep-no-line) → boolean?  
(make-print-dep-no-line on?) → void?  
  on? : any/c
```

A parameter that controls whether `make/proc` prints “checking...” lines for dependencies that have no target in the given `kspec`. The default is `#f`.

```
(make-print-reasons) → boolean?  
(make-print-reasons on?) → void?  
  on? : any/c
```

A parameter that controls whether `make/proc` prints the reason that a command thunk is called. The default is `#t`.

## 2.1 Signature

```
(require make/make-sig)    package: make
```

```
make~ : signature
```

Includes all of the names provided by `make`.

## 2.2 Unit

```
(require make/make-unit)    package: make
```

```
make@ : unit?
```

A unit that imports nothing and exports `make~`.

### 3 Building Native-Code Extensions

```
(require make/setup-extension)    package: make
```

The `make/setup-extension` library helps compile C code via Setup PLT’s “pre-install” phase (triggered by a `pre-install-collection` item in `"info.rkt"`; see also §6.3 “Controlling `raco setup` with `"info.rkt"` Files”).

The `pre-install` function takes a number of arguments that describe how the C code is compiled—mainly the libraries that it depends on. It then drives a C compiler via the `dynext/compile` and `dynext/link` functions.

Many issues can complicate C compilation, and the `pre-install` function helps with a few:

- finding non-standard libraries and header files,
- taming to some degree the differing conventions of Unix and Windows,
- setting up suitable dependencies on Racket headers, and
- using a pre-compiled binary when a `"precompiled"` directory is present.

Many extension installers will have to sort out additional platform issues manually, however. For example, an old `"readline"` installer used to pick whether to link to `"libcurses"` or `"libncurses"` heuristically by inspecting `"/usr/lib"`. More generally, the “last chance” argument to `pre-install` allows an installer to patch compiler and linker options (see `dynext/compile` and `dynext/link`) before the C code is compiled or linked.

```
(pre-install plthome-dir
            collection-dir
            c-file
            default-lib-dir
            include-subdirs
            find-unix-libs
            find-windows-libs
            unix-libs
            windows-libs
            extra-depends
            last-chance-k
            [3m-too?]) → void?
plthome-dir : path-string?
collection-dir : path-string?
c-file : path-string?
default-lib-dir : path-string?
include-subdirs : (listof path-string?)
```

```

find-unix-libs : (listof string?)
find-windows-libs : (listof string?)
unix-libs : (listof string?)
windows-libs : (listof string?)
extra-depends : (listof path-string?)
last-chance-k : ((-> any) . -> . any)
3m-too? : any/c = #f

```

The arguments are as follows:

- *plthome-dir* — the directory provided to a ‘pre-installer’ function.
- *collection-dir* — a directory to use as the current directory while building.
- *c-file* — the name of the source file (relative to *collection-dir*). The output file will be the same, except with a ".c" suffix replaced with (`system-type 'so-suffix`), and the path changed to (`build-path "compiled" "native" (system-library-subpath)`).
- If (`build-path "precompiled" "native" (system-library-subpath) (path-replace-suffix c-file (system-type 'so-suffix))`) exists, then *c-file* is not used at all, and the file in the "precompiled" directory is simply copied.
- *default-lib-dir* — a default directory for finding supporting libraries, often a sub-directory of "collection-dir". The user can supplement this path by setting the `PLT_EXTENSION_LIB_PATHS` environment variable, which applies to all extensions managed by `pre-install`.
- *include-subdirs* — a list of relative paths in which `#include` files will be found; the path will be determined through a search, in case it's not in a standard place like `"/usr/include"`.  
For example, the list used to be `'("openssl")` for the "openssl" collection, because the source uses `#include <openssl/ssl.h>` and `#include <openssl/err.h>`.
- *find-unix-libs* — like *include-subdirs*, but a list of library bases. Leave off the "lib" prefix and any suffix (such as ".a" or ".so"). For "openssl", the list used to be `'("ssl" "crypto")`. Each name will essentially get a -l prefix for the linker command line.
- *find-windows-libs* — like *find-unix-libs*, but for Windows. The library name will be suffixed with ".lib" and supplied directly to the linker.
- *unix-libs* — like *find-unix-libs*, except that the installer makes no attempt to find the libraries in a non-standard place. For example, the "readline" installer used to supply `'("curses")`.



- *windows-libs* — like *unix-libs*, but for Windows. For example, the "openssl" installer used to supply `'("wsock32")`.
- *extra-depends* — a list of relative paths to treat as dependencies for compiling "file.c". Often this list will include "file.c" with the ".c" suffix replaced by ".rkt". For example, the "openssl" installer supplies `'("mzssl.rkt")` to ensure that the stub module "mzssl.rkt" is never used when the true extension can be built.
- *last-chance-k* — a procedure of one argument, which is a thunk. This procedure should invoke the thunk to make the file, but it may add parameterizations before the final build. For example, the "readline" installer used to add an AIX-specific compile flag in this step when compiling on AIX.
- *3m-too?*— a boolean. If true, when the 3m variant is installed, use the equivalent to `raco ctool --xform` to transform the source file and then compile and link for 3m. Otherwise, the extension is built only for CGC when the CGC variant is installed.

## 4 Making Collections

```
(require make/collection)      package: make

(make-collection collection-name
                 collection-files
                 argv)          → void?
collection-name : any/c
collection-files : (listof path-string?)
argv : (or/c string? (vectorof string?))
```

Builds bytecode files for each file in `collection-files`, writing each to a "compiled" subdirectory and automatically managing dependencies. Supply `'#("zo")` as `argv` to compile all files. The `collection-name` argument is used only for printing status information.

Compilation is performed as with `raco make` (see *raco: Racket Command-Line Tools*).

### 4.1 Signature

```
(require make/collection-sig)  package: make

make:collection^ : signature
```

Provides `make-collection`.

### 4.2 Unit

```
(require make/collection-unit)  package: make

make:collection@ : unit?
```

Imports `make^`, `dynext:file^`, and `compiler^`, and exports `make:collection^`.