

roc-time manual

roc-time contributors

0.3.0

Contents

- **Time for real applications**
 - [Start with your task](#)
 - [Why intervals, not just instants?](#)
 - [What it looks like](#)
 - [Explicit choices, useful guarantees](#)
 - [Prior art](#)
- **Run your first application**
 - [Get the examples archive](#)
 - [Run the booking example](#)
 - [Change an input](#)
 - [Use the library in your own app](#)
 - [Choose your next step](#)
- **Find free booking windows**
 - [The pipeline](#)
 - [Keep the gaps](#)
 - [Know what can fail](#)
 - [Store standard values](#)
 - [API reference](#)
- **Work with dates and clock changes**
 - [Invoice terms: calendar arithmetic](#)
 - [Overnight staffing: interpret a local selection](#)
 - [Appointments need a choice](#)
 - [Parse and present civil labels](#)
 - [API reference](#)
- **Generate and exchange schedules**
 - [Start with a bounded reservation query](#)
 - [Treat incomplete output honestly](#)
 - [Importing iCalendar values](#)
 - [Editing and exchanging definitions](#)
 - [API reference](#)
- **Know the supported scope**
 - [Match the profile to your input](#)
 - [Failure is part of the interface](#)
 - [Archive interval descriptions](#)
 - [What is outside these guides?](#)
- **Find the right API**
 - [Exact time and availability](#)
 - [Calendar meaning and interpretation](#)

- [Recurrence, import and storage](#)
- [Developing against main?](#)
- **[Keep examples and versions together](#)**
 - [Published examples](#)
 - [Core and optional zone data](#)
 - [Development examples](#)
 - [When upgrading](#)
- **[Release notes](#)**
 - [roc-time 0.3.0](#)
 - [roc-time 0.2.0](#)

roc-time manual

Find free time between bookings, calculate calendar dates and query recurring appointments in Roc. This manual accompanies version 0.3.0; links to repository source can show newer APIs, identified in the relevant guide.

Start with [the first application](#), choose a task from [the overview](#), or use [the API map](#). Download the [PDF manual](#) for offline reading.

If you're contributing a change, start with the [contributor guide](#). The module API pages linked from each release provide exact function signatures; the chapters here explain how to put them together.

Time for real applications

roc-time helps you answer questions such as “When is this room free?”, “When is this invoice due?” and “Which appointments belong to next week’s schedule?” It keeps dates, elapsed time and bookings distinct, so you don’t have to recover their meaning from bare timestamps.

If you know some Roc, you can [run your first application](#) with a released examples archive. You don’t need to build the library first.

Start with your task

You need to...	Start here
Read booking timestamps and find free space	Booking and coverage
Calculate an invoice date or an overnight shift	Calendars and zones
Generate recurring reservations and handle exceptions	Schedules
Check whether your input format is supported	Profiles and limits
Find a module or function	API map
Upgrade an existing application	Versions

The [releases page](#) has runnable examples, matching API documentation and the compiler used to test each release. Contributors can start with [the contributor guide](#).

Why intervals, not just instants?

A booking occupies time between two boundaries. To find availability, subtract that time from opening hours and keep every gap that remains. `Coverage` holds those pieces for you.

Each span includes its start and excludes its end. Two bookings can meet at 10:00 without overlapping: `[09:00, 10:00)` followed by `[10:00, 11:00)`. You don’t need a “last microsecond of the morning.”

Sometimes you also need to know whose booking it is. Two customers can book the same window; their events keep separate identities even when their occupied coverage is equal. Likewise, an archive date of “June 2026” keeps month precision rather than becoming a recorded instant at midnight on June 1.

What it looks like

An opening window is 09:00–17:00 UTC. A booking arrives as 12:00–14:00 at UTC+02:00, which occupies 10:00–12:00 UTC. Subtracting it leaves **09:00–10:00 and 12:00–17:00 UTC**. A single earliest-start/latest-end pair would accidentally fill the gap.

The [booking application source](#) parses inputs, subtracts bookings and prints UTC timestamps. The [first application guide](#) shows how to run the release’s copy.

Explicit choices, useful guarantees

Exact calculations use signed 64-bit integer microseconds and checked arithmetic. Your application supplies a clock reading or timezone rules when it needs them. For repeated local times, you choose which occurrence an appointment means. For a long schedule, you choose how much work one query can do and resume it if necessary.

[Profiles and limits](#) describes the formats and operations supported today. Parsing, display, explanation and storage serve different purposes; a readable label doesn’t replace the original declaration or its interpretation context.

Prior art

roc-time is inspired by [Kip Cole’s Tempo](#), especially its treatment of calendar precision, intervals and set algebra. Credit belongs to Kip and Tempo’s contributors for that foundation. roc-time is an independent Roc implementation with its own API.

Run your first application

You’ll run a booking application that reads opening hours and two bookings, then prints the free windows in UTC. You’ll need a Roc compiler and the examples archive from a package release.

Get the examples archive

Choose a [release](#), download `roc-time-examples-<version>.zip` and extract it. Install the Roc nightly named in that release’s notes. A later nightly may work too; check the compatibility results if it fails.

The archive already contains the package URLs you need. Keep each application’s whole folder: `main.roc` imports the neighboring module that does the calculation.

Run the booking example

From the extracted folder:

```
roc version
roc examples/booking_exchange/main.roc
```

The version command lets you compare your compiler with the release notes. The application prints:

```
Restored available booking windows (UTC)
2026-06-15T09:00:00.000000Z/2026-06-15T10:00:00.000000Z
2026-06-15T12:00:00.000000Z/2026-06-15T15:00:00.000000Z
2026-06-15T16:00:00.000000Z/2026-06-15T17:00:00.000000Z
```

Each line is a free interval with its start and end separated by /. The shop opens at 09:00 UTC and closes at 17:00. One booking arrives as 12:00–14:00 at UTC+02:00, which is 10:00–12:00 UTC; the other occupies 15:00–16:00 UTC.

Roc downloads the content-addressed package dependencies on first use. You don't need Python or a separate example runner.

Change an input

Open `examples/booking_exchange/main.roc`. Remove the second booking, `2026-06-15T15:00:00Z/2026-06-15T16:00:00Z`, from the list and run the application again. The two afternoon windows join into a single 12:00–17:00 UTC window.

You can also move a booking by editing its timestamps. Keep the `z` or numeric offset: without one, a local clock time doesn't tell the application where the booking belongs on the timeline. `BookingExchange.roc` parses the text and subtracts the occupied time through the public library API.

Use the library in your own app

Copy the `time` dependency from an archived example's header into your application's header, keeping your own platform declaration. This gives you the exact package URL tested for that release. The library doesn't pin your compiler.

For named zones, also copy the `zones` dependency from the archived `examples/staffing/main.roc`. It supplies timezone data separately, so an application working only with explicit offsets doesn't need the database.

If you copied an example from the repository instead, its relative dependencies point at the local checkout. [Versions](#) explains how to keep source, packages and API documentation together.

Choose your next step

- [Find availability](#): timestamps, spans and coverage.

- [Calculate dates and handle clock changes](#): due dates and local appointments.
 - [Generate schedules](#): recurring bookings, exceptions and bounded queries.
- If an imported API is missing, compare your package URL with the example's version before changing the compiler or rewriting imports.

Find free booking windows

Suppose your application receives bookings as timestamp strings and needs to offer the time left in a working day. Start with an opening window, turn each booking into occupied time, then subtract it. Keep an explicit `z` or numeric offset in each timestamp so bookings from different offsets can be compared.

The pipeline

The [booking exchange source](#) separates input/output in `main.roc` from the calculation in `BookingExchange.roc`. After [getting the examples](#), run:

```
roc examples/booking_exchange/main.roc
```

The application follows five steps:

1. `ExactInterval.parse` validates the opening and booking strings.
2. `ExactInterval.span` gives each interval's timeline span.
3. `Coverage.from_spans` combines the occupied spans.
4. `Coverage.complement_within` finds the free pieces inside the opening span.
5. The application formats their endpoints as canonical UTC timestamps.

For a 09:00–17:00 opening and a 10:00–12:00 booking, the free windows are **09:00–10:00 and 12:00–17:00**. Each span includes its start and excludes its end; a booking ending at noon doesn't occupy noon onward.

Keep the gaps

`Coverage` keeps every separate piece of time. It merges overlapping or touching bookings, so you don't have to remove duplicate occupied time before finding availability.

That merge loses booking identity. If two customers both book 10:00–12:00, their combined coverage is still one span. Keep an `EventCollection` alongside coverage when you need to report conflicts or identify whose booking contributed the time.

Know what can fail

An interval needs a start earlier than its end. Reversed endpoints, an empty interval and malformed timestamps return errors; the parser won't swap endpoints or guess a timezone.

The timestamp profile accepts microseconds. Input with finer precision is rejected rather than truncated, and arithmetic near the signed coordinate limits is checked. [Profiles and](#)

[limits](#) explains how to distinguish invalid input from valid forms the selected parser doesn't support.

Store standard values

When saving availability, keep each free span and serialize its endpoints. Your application chooses the surrounding collection format. A single earliest-start/latest-end pair would fill the gaps you've just calculated.

The [catalogue storage example](#) saves EDTF date descriptions and RFC 3339 timestamps in application-owned fields, then parses them again. The formats' serializers are the storage boundary; debug inspection text is for diagnostics.

For an expiry check, the [clock deadline example](#) reads the clock in its platform root and passes that reading to a pure comparison. Its saved JSON describes the check at that recorded time. Loading the JSON later doesn't perform a new check.

API reference

Start with `ExactInterval`, `OffsetTimestamp` and `Coverage` in the [API map](#). Repository links above show current source; the release archive and API pages show the version you're using.

Work with dates and clock changes

An invoice due next month and a shift that crosses a clock change need different calculations. Here you'll calculate a due date, then work out how many hours an overnight shift covers.

The commands below work from a checkout with a current Roc nightly. For released packages, get the same applications from the [examples archive](#). Repository links show current source, which may be ahead of your release.

Invoice terms: calendar arithmetic

A merchant issues an invoice on January 31 with payment due one calendar month later. February has no 31st, so the merchant's terms clamp the date to the month's last day. The [invoice example](#) applies that policy:

```
roc examples/invoice/main.roc
```

```
Payment terms: one calendar month, clamped to month end
Issued: 2025-01-31
Payment due: 2025-02-28
```

`Calendar.Arithmetic` shifts the calendar date with an explicit destination policy. If your business rule requires a missing day to be rejected, choose `rejection` instead. Adding a fixed number of seconds can't express either rule reliably. Adding a month and subtracting one also needn't recover the original date after clamping: February 28 doesn't remember January 31.

Gregorian and Julian dates use their rules in every supported year, rather than modeling each country's historical adoption. Year zero means 1 BCE. Converting between the calendars keeps the civil day; it doesn't choose midnight or a timezone.

Overnight staffing: interpret a local selection

A Melbourne team works from 22:00 to 06:00 across the spring clock change. The clock skips an hour, so the shift covers seven hours rather than eight. Run the [staffing example](#):

```
roc examples/staffing/main.roc
```

```
Overnight staffing in Australia/Melbourne
Local shift: 22:00 to 06:00 across the spring clock change
Budget 7 hours per team member.
```

The application selects named-zone data from the optional package, validates it with `ZoneRules.from_database`, and asks which timeline positions have labels in the local shift. The library doesn't consult your machine's timezone. The data package's manifest tells you which release and date range it covers.

A **local selection** means all positions whose clock labels fall in the chosen range. During a repeated hour, it can contain separate pieces; a skipped date can contain none. Keeping only the earliest and latest positions would fill gaps that weren't selected.

Appointments need a choice

An appointment at a repeated 01:30 is a different question: which occurrence did the person book? A resolver reports a repeated time as `Fold`, a skipped time as `Gap`, and a single match as `Unique`. Your application then applies the appointment's policy. The [repeated local time example](#) compares the choices with selecting the whole repeated range.

The rules must cover enough time to find every possible match. If the query reaches beyond the data's validity, it returns an error even when one candidate is visible. Select rules covering the request rather than treating that candidate as complete or assuming UTC.

Parse and present civil labels

`GregorianCalendar`, `ClockTime` and `LocalDateTime` parse date and clock labels without assigning a zone. `EnglishGregorianCalendar` can display them as 7 Sep 2026, 09:30. Its minute/second choices reject nonzero omitted fields, so presentation won't silently discard precision. The [appointment display example](#) shows these choices.

A displayed local label hasn't been proved to exist in any particular zone. For a named-zone appointment, interpret it first and present the selected result, as in the [named-zone appointment example](#).

Gregorian dates also provide weekday, ordinal-day and ISO week-date queries. The [invoice report](#) combines them with calendar arithmetic. Consult your release's API pages for its available operations.

API reference

`Calendar.Arithmetic`, `Calendar.Date`, `LocalDateTime` and `ZoneRules` are mapped in [the API guide](#).

Generate and exchange schedules

A weekly reservation needs a starting date, a repetition rule and any exceptions. When you query next month, you still want the same series: its count and cancellations shouldn't restart at the beginning of your screenful.

Start with a bounded reservation query

The [equipment reservations source](#) demonstrates recurring appointments with identities and ending times. Run the copy in your [release archive](#), or run the current source from a checkout:

```
roc examples/reservations/main.roc
```

There are three choices to make:

1. **Definition:** when does the series start, how does it repeat, and which dates are added or excluded?
2. **Interpretation:** which timezone rules and repeated/skipped-time policies apply to those local starts?
3. **Query:** which start-date window do you want, and how much work and output can this call produce?

`ScheduleDefinition` retains the declaration and context for fresh queries.

`TimedSchedule.collect` collects a bounded batch from a cursor. For dates without clock times or appointment endings, start with `DateRecurrence`.

A query selects appointment **starts**. A booking starting at 16:00 can end after a 17:00 query boundary. To find every booking overlapping a reporting window, account for bookings that started earlier as well; a start query alone doesn't answer that question.

Treat incomplete output honestly

A batch marked `Complete` finished the requested query. A `Limited` batch contains partial output and the cursor you need to continue. Keep that cursor: starting again from the next date can lose counting and exception state.

Work and output limits serve different purposes. A rule might examine many dates before finding a match, so “one appointment per batch” doesn't bound the search. If a budget can't make progress, increase it or report the incomplete result rather than retrying forever. The [API reference](#) describes the limits for each query.

Importing iCalendar values

`ICalDateRule` and `ICalTimedRule` accept supported RFC 5545 property values after your application has extracted them. They don't read a whole `.ics` file, unfold content lines or accept arbitrary parameters. Check the chosen adapter's profile before importing an external calendar.

For example, a rule's `COUNT` counts its anchored recurrence before exclusions. Canceling one of four rule occurrences doesn't create a fifth. A `PERIOD` addition can instead contribute a separate appointment with its own end. An occurrence's original local label stays meaningful even if a gap policy moves its resolved start.

Unsupported forms return errors. That lets your importer report which part needs another adapter without quietly changing the schedule.

Editing and exchanging definitions

The [meeting exchange example](#) edits a weekly meeting, exports its extracted properties, then reimports them across a fixed zone transition. These adapters expose checked definitions and canonical property text, so application code doesn't have to reconstruct a rule from query output.

Save supported declarations using their standard property serializers and supply interpretation context when loading them. A property string doesn't embed a zone table, and some native calendar policies can't be expressed by the selected standard profile. Export rejects those cases rather than dropping the policy.

Repository examples show the development API. Check your release archive and API pages before copying an operation into an application using an older package.

API reference

`DateRecurrence`, `TimedRecurrence`, `TimedSchedule`, `ScheduleDefinition` and `ICalTimedRule` appear in [the API map](#).

Know the supported scope

Before connecting an importer, check that the selected parser supports the forms your source produces. A calendar feed can contain a valid rule that a particular adapter doesn't handle. A date description can parse successfully without identifying a single instant.

Each module documents a **profile**: the syntax and meaning it accepts. The API pages for your release list exact grammars, limits and errors.

Match the profile to your input

Area	Support and limits
Exact values	Compare explicit-offset timestamps and combine spans. Calculations use signed I64 microseconds and check overflow and precision.
Calendars	Convert Gregorian/Julian dates and calculate Gregorian deadlines. Their rules apply in all supported years; historical national reforms aren't inferred.
Named zones	Interpret local labels with the optional IANA data package. Supply rules that cover the requested time range.
Timestamp/interval text	Parse supported explicit-offset timestamps and exact intervals. These profiles reject leap seconds and sub-microsecond input.
EDTF	Retain supported date precision and qualifications. The date adapter supports a subset of EDTF; check its grammar before importing a wider form.
IXDTF	Retain annotations and check supplied offset/zone assertions. An annotation can be retained without supporting its presentation; unknown critical extensions fail.
Recurrence	Import supported extracted RFC 5545 DATE/DATE-TIME values. There is no general ICS reader, and recurrence rules differ from repeated clamped arithmetic.
Explanation	Ask what a supported value means and which context it needs. Work/output limits apply; diagnostic prose isn't a storage format.
Standard text storage	Save values through their standard parser/serializer pair. The text doesn't embed every interpretation choice or an immutable zone table.

Follow the [API map](#) to find the module for your task. A format's name alone doesn't establish support for every part of its standard.

Failure is part of the interface

Different failures call for different responses:

- **Malformed:** ask the user to fix the input's spelling or structure.
- **Unsupported:** the input needs a capability outside this parser's profile; changing its spelling won't add that capability.
- **Out of range:** choose a supported date range or rules covering the requested time.
- **Incomplete evaluation:** resume with the returned cursor or choose a suitable budget. Partial output isn't a complete empty result.

Your application supplies clock readings, timezone data and any choice of a repeated local occurrence. The library won't use your machine's timezone or fetch replacement data. Standard adapters can have their own documented interpretation policies, so check the adapter as well as the underlying type.

Archive interval descriptions

The current source also provides `EdtfInterval` and `CalendarInterval` for date intervals with independently described endpoints. For example, `2004-02-01/2005-02` retains a day on one side and a month on the other. `1985/` has an unknown end; `1985/..` has an explicitly open end. Those are different descriptions.

These modules and the [archive intervals example](#) are available from 0.3.0. Parsing and saving the interval doesn't resolve it into timeline coverage. The module reports interpretation as unsupported rather than inventing exact endpoint dates. See the 0.3.0 API for its declared EDTF subset.

What is outside these guides?

General ICS processing, wider calendar providers, leap-aware time and complete ISO/EDTF/RRULE coverage need separately declared support. The [design](#) describes the intended architecture, including work beyond the selected release.

Find the right API

This map helps you choose a starting module. It describes the current source; follow your [release's API link](#) for the exact functions available in your package.

Exact time and availability

For booking availability, start with `ExactInterval.parse`, project each booking with `ExactInterval.span`, and build `Coverage`. Use `Coverage.complement_within` with an explicit opening span to find free windows. Keep an `EventCollection` when you also need booking identities: coverage merges occupied time and cannot recover which booking contributed it.

For a single timestamp, `OffsetTimestamp.parse` checks its text and `to_text` writes the canonical form. A `PosixBoundary` identifies its position on the timeline; `PosixDelta` represents a displacement between positions.

Task	Modules
Parse explicit-offset timestamps	OffsetTimestamp
Carry a timeline position or displacement	PosixBoundary, PosixDelta
Work with nonempty spans and interval text	PosixSpan, ExactInterval
Combine occupied time and find gaps	Coverage
Keep event identities	EventCollection

Calendar meaning and interpretation

`Calendar.Date` holds a validated date; `Calendar.Delta` says how far to shift it in calendar units; `Calendar.Value` can describe a whole year or month. These are separate types: “June 2026” and “one month later” do different jobs. Import `Calendar` to access them and `Calendar.Arithmetic` for date calculations.

For invoice dates, start with `GregorianCalendar` and use `Calendar.Arithmetic` when advancing a month. Choose the invalid-date policy explicitly: January 31 with clamping and January 31 with rejection answer different business questions. Use `Calendar.Value` when supplied precision, such as a whole month, is part of the meaning rather than an exact date.

A `LocalDateTime` describes what a clock shows. `FixedOffset` places it on a timeline when you know the offset. `ZoneRules` handles named-zone appointments, including skipped and repeated times. Selecting all positions in a local day is a separate question: a skipped day can be empty, and repeated labels can produce separate pieces of coverage.

Task	Modules
Construct and convert civil dates	GregorianCalendar, JulianDate, Calendar.Date
Preserve supplied precision	Calendar.Value, QualifiedCalendarValue
Advance with an explicit calendar policy	Calendar.Arithmetic, Calendar.Delta
Express clock labels and resolve them	ClockTime, LocalDateTime, FixedOffset, ZoneRules
Display a Gregorian date or local label in English	EnglishGregorian

Recurrence, import and storage

For reusable timed appointments, `ScheduleDefinition.from_ical` imports a supported iCalendar definition and `from_native` accepts a native one. Create a cursor for the start-date window you want, then call `TimedSchedule.collect` with work and output limits. A `Limited` batch includes the cursor needed to continue; keep it with the partial output. [Schedules](#) explains what the query selects.

Start with `DateRecurrence` for date-only rules. The lower-level patterns and `TimedRecurrence` are useful when composing selectors or working with starts without appointment endings; ordinary scheduling need not begin there.

If you're composing execution inputs directly, `TimedSchedule.Endings` validates ending definitions for reuse. The adapters' `prepare` operations and `TimedSchedule.from_prepared` expose that advanced path. To retain a declaration for later queries, keep its `ScheduleDefinition`; prepared execution inputs aren't a storage format.

Task	Modules
Generate calendar dates	<code>DateRecurrence</code> , <code>CalendarPattern</code>
Query reusable timed appointments	<code>ScheduleDefinition</code> , <code>TimedSchedule</code>
Compose timed recurrence starts	<code>TimedRecurrence</code>
Import extracted iCalendar values	<code>ICalDateRule</code> , <code>ICalTimedRule</code>
Preserve imported descriptions	<code>EdtfDate</code> , <code>Ixdtf</code>
Explain a supported value	<code>Explanation</code>

Version 0.3.0 adds `CalendarInterval` and `EdtfInterval` for independently described interval endpoints, including unknown and open ends. They retain those meanings for storage without claiming resolved coverage. [The archive interval guide](#) shows the distinction and links the new example.

[Choose a release and its API documentation](#)

Developing against main?

The [module source and documentation on main](#) describe the development API. Generate matching local API documentation with a current Roc nightly:

```
roc docs package/main.roc --output=.roc-time-tmp/api-main
```

Use documentation matching the package in your application header. Source documentation may describe APIs not yet included in your chosen release.

Keep examples and versions together

When copying an example, keep its whole application folder and its package URLs together. The folder supplies neighboring modules; the URLs select the library version. The release notes tell you which Roc compiler tested that combination.

Published examples

A release's `roc-time-examples-<version>.zip` contains complete applications whose `time` and optional `zones` dependencies point at immutable packages. Download it from the [releases page](#) when you want to target that version.

The daily compatibility workflow tests the latest published release against the latest Roc nightly. If a new compiler breaks an older package, check the workflow result and available patch releases. Existing release URLs don't change; adopting a fix means updating your package URL.

Roc files don't pin the compiler. Starting with the compiler named in the release notes gives you the tested combination; later nightlies need their own compatibility check.

Core and optional zone data

The release notes identify the core and zone-data packages. The core is enough for explicit-offset timestamps, interval algebra and calendar calculations. Add the data package for named zones.

A data update can change how a future local meeting resolves. Existing in-memory interpretations keep their original rules; explicitly resolve again when you want the new rules. The [interpretation context source](#) demonstrates both choices.

Saving an IXDTF declaration records its supported text and annotations, not the rules used to interpret it. Supply the appropriate context when loading it. A zone name and version identify a snapshot; they don't authenticate an arbitrary transition table or fetch one for you.

The [repeated local time example](#) also shows why a local timestamp alone can't record every choice: selecting the first occurrence, the last occurrence or the whole repeated range gives different results.

Development examples

Repository examples import the working-tree packages by relative path. Run them from a complete checkout with a current Roc nightly. They can use APIs added after the latest release; copying a new module name into an old release's header won't make that API available.

The manual links to current source for browsing. For matching executable examples and function signatures, follow your release's archive and API links.

When upgrading

1. Read the target release's notes and API changes.
2. Copy the relevant package URLs from its archive. Start with its tested compiler or verify your later nightly.
3. Update your application's dependencies and keep any example companion modules you copied.
4. Run the application with the date, precision and clock-change cases that matter to it.

Each release also has a tested HTML/PDF manual and compiler-generated API archive. Historical API URLs stay stable. The pinned native renderer builds the manual without a package compiler; that doesn't change which compiler your application needs.

Release notes

The [release assets](#) carry the exact package URLs and tested compiler versions. Historical API archives remain available under their original versioned URLs.

roc-time 0.3.0

This release adds date interval descriptions, updates the bundled timezone rules and makes the manual easier to follow.

`CalendarInterval` and `EdtfInterval` preserve independently described endpoints. For example, `2004-02-01/2005-02` keeps a day at the start and a month at the end. An unknown end (`1985/`) stays distinct from an open end (`1985/. .`). Checked construction, parsing, canonical text and codecs retain those meanings for storage and exchange. The new archive intervals application shows how to use them. This is a declared EDTF date-interval subset; interpreting these descriptions as timeline coverage remains unsupported.

The optional zone package now contains IANA 2026e rules from tzdata 2026.5. This corrects changed offsets, including a Winnipeg appointment at 09:00 on 2026-11-02: the updated rules resolve it to 14:00 UTC. All previously accepted zone names remain available.

Recurrence behavior is unchanged. New independent checks cover 108 composed local-zone schedules, including gaps, folds, exclusions, ending intent and resumption. They complement the existing UTC fixtures and bounded resource checks.

The guides now start with concrete tasks and real example output. The HTML and PDF manual remove repeated chapter-home links and give the supported-format summary more room. The examples archive includes the new archive intervals application, bound to this release's packages.

Thanks to Kip Cole and the Tempo contributors for the ideas behind roc-time's interval-oriented model.

roc-time 0.2.0

This release expands calendar, recurrence and interchange support, and runs on the current Roc nightly. It includes complete example applications bound to the released packages, plus an archived API reference and an offline HTML/PDF manual.

Calendar APIs now cover Gregorian and Julian conversion, calendar queries and explicit calendar arithmetic. Recurrence APIs support bounded, resumable date and timed schedules, with checked iCalendar import and export, exceptions and cancellations. Interpretation keeps local selections, appointments, gaps and folds distinct; named-zone data remains an optional companion package.

Additional examples demonstrate meeting exchange, historical archive search, availability, invoice reporting and named-zone appointments. Download the examples archive below to run them against this release.

Package headers now float on the nightly compiler. Use the tested compiler listed below when reproducing a result. Repository automation runs separately on a Blueprint-pinned compiler. Releases verify every uploaded asset before publication; versioned API documentation is restored from those same assets. The guide site now uses the shared native roc-docs renderer, with a linked PDF containing all eight guides. Existing guide routes and versioned API URLs remain available.

The supported standard profiles and limits remain documented in each module; these APIs do not claim complete ISO 8601, EDTF or RFC 5545 support.

Thanks to Kip Cole and the Tempo contributors for the ideas behind roc-time's interval-oriented model.