The Tcl Interface to Ical

Sanjay Ghemawat

March 16, 1995

Contents

1 Introduction **1**

2 Computation Model **1**

3 Calendars **1**
- 3.1 Files and Includes 1
- 3.2 Adding and Removing Items 2
- 3.3 Input and Output 2
- 3.4 User Preferences 2
- 3.5 Queries 2

4 Items **3**
- 4.1 Creation and Deletion 3
- 4.2 Item Occurrences 4
- 4.3 Item Properties 5

5 Dates **6**

6 Times **7**

7 Customization via Hooks **8**
- 7.1 General Hooks 8
- 7.2 Day Window Hooks 8

8 Day Windows **9**

9 Common Editing Actions **9**

10 Miscellaneous Commands **9**

1 Introduction

This document contains a brief description of the Tcl interface to ical. Part of this interface is implemented in C++ and the rest is implemented by Tcl support libraries.

The C++ code exports calendar and item objects to tcl code. A number of operations are provided to create such objects and to manipulate dates and times. In addition, the calendar and item objects have various methods that can be called from tcl code.

2 Computation Model

Calendars and items are stored persistently on file systems. Copies of these calendars and items are read into ical's address space. The Tcl code operates on these copies through well-defined interfaces. These interfaces contain special operations for saving pending modifications to persistent storage on the file system.

Each user has a main calendar. This main calendar is represented as a calendar object in Tcl code, and persistently as a file stored on the file system. The main calendar contains a list of names of included calendar files. The main calendar file and the included calendar files all contain items. Each item in the main calendar file and the included calendar files is visible as an item object to the Tcl code.

3 Calendars

The following calendar operations raise errors if the user does not have sufficient permission to perform the required operation. Errors are also raised if specified files do not contain a calendar. Some of these operations also take a calendar file name as an optional argument. If the file name is omitted, then the main calendar file is used.

Arguments named *file* in the following list are names of Unix files. Arguments named *cal* are calendar objects created by the "calendar" command.

3.1 Files and Includes

This section lists the operations for creating and deleting calendar objects from calendar files, and also operations for handling included calendars.

- **calendar** *cal file*

Create a calendar object named *cal* with *file* as the main calendar file. The actual file named *file* contains a list of included calendar files. The files mentioned in this list are also incorporated into *cal*. Typically only one calendar object is created in an ical application. Included calendars are considered part of this one calendar object.

- *cal* **delete**

  Discard *cal* and all contained items from process memory. The underlying files are not destroyed and can be used to initialize a calendar object in another invocation.

- *cal* **main**

  Return the name of the main calendar file for *cal*.

- *cal* **include** *file*

  Include *file* into *cal*. All items contained in *file* are made available as item objects to Tcl code.

- *cal* **exclude** *file*

  Exclude *file* from *cal*. All objects for items contained in *file* are removed from the Tcl interpreter and cannot be used by Tcl code any more.

- *cal* **forincludes** *var body*

  For each include *file* in *cal*, set *var* to *file* and execute *body*.

3.2 Adding and Removing Items

This section describes the operations for adding and removing items from calendars.

- *cal* **add** *item* **[***file***]**

  Add *item* to the specified *file* in *cal*.

- *cal* **remove** *item*

  Remove *item* from *cal*.

- *cal* **hide** *item*

  Hide *item* from this user. Other users will still see the item.

3.3 Input and Output

This section describes the operations for reading and writing calendars to persistent storage.

- *cal* **readonly [***file***]**

  Returns true iff the user does not have permission to modify the specified *file*.

- *cal* **dirty [***file***]**

  Has the specified *file* been modified locally without being saved to persistent storage?

- *cal* **stale [***file***]**

  Has the specified *file* been modified by another user or process?

- *cal* **save [***file***]**

  Save any local changes made to *file* to persistent storage. For example, the following code saves all modifications —

```
if [cal dirty [cal main]] {
    cal save [cal main]
}

cal forincludes file {
    if [cal dirty $file] {
        cal save $file
    }
}
```

- *cal* **reread [***file***]**

  Incorporate any changes made to *file* by other users or processes. Any existing items objects from *file* may be deleted and replaced by new item objects even if the file has not been modified recently.

3.4 User Preferences

Each calendar file contains a general mapping from option names to string values. This mapping is typically used to store user preferences for ical. Sometimes, however this mapping contains miscellaneous properties of the calendar file.

- *cal* **option [-calendar** *file***]** *name*

  Look up the value associated with *name* in the named calendar *file*. If no calendar file is specified, the value is looked up in the main calendar file. An error is raised if the named option does not exist in the calendar. Example —

```
set p "lpr"
catch {set p [cal option PrintCmd]}
```

- *cal* **option [-calendar** *file***]** *name value*

  Set the value of the option named *name* to *value* in the named calendar *file*. If no calendar file is specified, the value is looked up in the main calendar file. Example —

  ```
  cal option PrintCmd "lpr −m"
  ```

3.5 Queries

The following operations can be used to get listings of various items in a calendar. The range of most of these queries can be controlled by specifying one of the following options:

- **-all**

  Query ranges over all calendars and items.

- **-calendar** *file*

  Query ranges over the contents of the specified calendar file.

- **-list** *item-list*

  Query ranges over only the specified list of items.

The actual query methods are as follows:

- *cal* **query [***range options***]** *start finish itemvar datevar body*

  For any item occurrence in the range *start..finish*, set *itemvar* to the item, set *datevar* to the date of the item occurrence, and then evaluate *body*. The executions of *body* are sorted by occurrence.

- *cal* **listing [***range options***]** *start finish itemvar datevar body*

  For all items *i*, For each occurrence of *i* in *start*..(*finish*+[*i* earlywarning]), Set *itemvar* to *i*, *datevar* to the occurrence date and execute *body*. The executions of *body* are sorted by occurrence. This operation differs from "*cal* query" only in its handling of early warnings. Example —

  ```
  cal listing $date [expr $date+9] i d {
      print_date $d
      print_item $i
  }
  ```

- *cal* **incalendar** *file itemvar body*

  For all items *i* in *file*, Set *itemvar* to *i* and execute *body*. The executions of *body* are sorted by date of first occurrence. This operation is equivalent to the "query" operation executed with a "-calendar" range option. Example —

  ```
  cal incalendar [cal main] item {
      print_item $item
  }
  ```

- *cal* **loop_forward [***range options***]** *item date itemvar datevar body*

  For each item occurence that occurs after the specified *item* on the specified *date*, set *itemvar* and *datevar* to the item occurrence and execute *body*. If *item* is the empty string, then start yielding with the first item that occurs on *date*. For example, the following code searches for the string "birthday" —

  ```
  cal loop_forward $item $date i d {
      if [string match "*birthday*" [$i text]] {
          ...
          break
      }
  }
  ```

- *cal* **loop_backward [***range options***]** *item date itemvar datevar body*

  For each item occurence that occurs before the specified *item* on the specified *date*, set *itemvar* and *datevar* to the item occurrence and execute *body*. If *item* is the empty string, then start yielding with the last item that occurs on *date*. For example, the following code searches for the string "birthday" —

  ```
  cal loop_backward $item $date i d {
      if [string match "*birthday*" [$i text]] {
          ...
          break
      }
  }
  ```

4 Items

Item objects come in two flavors – appointments and notices. An appointment occurs at a specific time of the day. A notice does not have any associated time of day.

For example, a meeting at 3pm on March 27th will be recorded as an appointment while somebody's birthday on March 28th will be recorded as a notice.

Each item object also has associated text and a set of dates on which the item occurs.

4.1 Creation and Deletion

The following operations can be used to create and delete items.

- **notice**

  Create a notice object with empty text and an empty set of dates.

- **appointment**

  Create an appointment object with empty text and an empty set of dates.

- *item* **delete**

  Remove *item* from any calendar that contains it and delete all storage required for the item.

- *item* **clone**

  Create and return a copy of *item*.

4.2 Item Occurrences

The following operations can be used to manipulate and query the set of dates on which an item occurs.

- *item* **contains** *date*

  Returns true iff *item* occurs on *date*.

- *item* **empty**

  Returns true iff *item* has no occurrences.

- *item* **repeats**

  Returns true iff *item* occurs more than once.

- *item* **first**

  Returns the date of first occurrence of *item*. Raises an error if *item* has no occurrences.

- *item* **next** *date*

  Returns the date of the first occurrence of *item* after *date*. Raises an error if *item* has no occurrences after *date*.

- *item* **range** *startVar finishVar*

  Sets *startVar* and *finishVar* to the repetition range for *item* and returns 1. Returns 0 if item does not repeat.

- *item* **type**

  Returns a brief textual description of how the item repeats.

- *item* **describe_repeat**

  Like "*item* **type**", except that it returns a more complete description of how the item repeats..

- *item* **date** *date*

  Modifies *item* to occur exactly on *date*.

- *item* **dayrepeat** *interval anchor*

  Modifies *item* to occur on all dates that are an integral multiple of *interval* days away from *anchor*. For example, the following code makes "$item" repeat every Saturday assuming that the anchor date occurs on a Saturday.

  ```
  $item dayrepeat 7 $anchor
  ```

- *item* **monthrepeat** *interval anchor*

  Modifies *item* to occur on all dates that are an integral multiple of *interval* months away from *anchor*. For example, the following code makes "$item" repeat on the 23rd of February every year.

  ```
  $item monthrepeat 12 [date make 23 12 1994]
  ```

- *item* **weekdays [***weekday***...]**

  Modifies *item* to repeat on all specified *weekdays*. For example, the following code makes "$item" repeat on every Monday, Wednesday and Friday (Sunday is represented by 1, Monday is represented by 2, . . .).

  ```
  $item weekdays 2 4 6
  ```

- *item* **month_day** *n* **[***anchor interval***]**

  Modifies *item* to repeat on the *n*th day of every month that is an integral number of *interval* months away from the date specified in *anchor*. If *anchor* and *interval* are not specified, then the *item* repeats every month. For example, the following code makes

"$item" repeat on the 17th of every January because the anchor date is in January, and the interval is 12.

```
$item month_day 17 [date make 1 1 1994] 12
```

- *item* **month_last_day** *n* **[***anchor interval***]**

  Behaves like "*item* **month_day**" except that *item* is made to repeat on the *n*th-*last* day of the month. For example, the following code makes "$item" repeat on the second last day of every month.

  ```
  $item month_last_day 2
  ```

- *item* **month_work_day** *n* **[***anchor interval***]**

  Behaves like "*item* **month_day**" except that *item* is made to repeat on the *n*th *working day* of the month. For example, the following code makes "$item" repeat on the first working day of each month.

  ```
  $item month_work_day 1
  ```

- *item* **month_last_work_day** *n*

  Behaves like "*item* **month_last_day**" except that *item* is made to repeat on the *n*th-last *working day* of the month. For example, the following code makes "$item" repeat on the fourth-last working day of each month.

  ```
  $item month_last_work_day 4
  ```

- *item* **month_week_day** *day n* **[***anchor interval***]**

  Behaves like "*item* **month_day**" except that *item* is made to repeat on the *n*th occurrence of a particular day of the week of the month.

  The *day* argument should be an integer in the range 1 . . . 7. An argument of 1 means that the item should repeat on the *n*th Sunday of the month. An argument of 7 means that the item should repeat on the *n*th Saturday of the month. For example, the following code makes "$item" repeat on the third Thursday of every month. (Thursday is specified by the integer 5).

  ```
  $item month_week_day 5 3
  ```

- *item* **month_last_week_day** *day n* **[***anchor interval***]**

  Behaves like "*item* **month_week_day**" except that occurrences of *day* are counted from the end of the month. For example, the following code makes "$item" repeat on the last Friday of every month. (Friday is specified by the integer 6.)

  ```
  $item month_last_week_day 6 1
  ```

- *item* **start** *date*

  Modifies *item* by removing all occurrences that occur before *date*.

- *item* **finish** *date*

  Modifies *item* by removing all occurrences that occur after *date*. For example, the following code restricts "$item" to occur only in 1994.

  ```
  $item start   [date make 1  1   1994]
  $item finish [date make 31 12 1994]
  ```

- *item* **deleteon** *date*

  If *item* occurs on *date*, removes that occurrence of *item*. Otherwise leaves *item* unmodified.

4.3 Item Properties

The following operations can be used to examine and manipulate various properties of an item. The first few operations apply only to appointments. The remaining operations apply to both appointments and notices.

- *appt* **length**

  Return *appt*'s length in minutes.

- *appt* **length** *length*

  Set *appt*'s length to *length* minutes.

- *appt* **starttime**

  Return the starting time for *appt* in minutes since midnight.

- *appt* **starttime** *minutes*

  Set the starting time for *appt* to *minutes* after midnight.

- *appt* **alarms**

  Return a list of alarm times. For each entry *t* in this list, ical will generate an alarm *t* minutes before the appointment starts. All of the elements in this list are non-negative.

- *appt* **alarms** *list*

  Set the list of alarm times for *appt*. All of the elements of this list should be non-negative. For example, the following code will remove all alarms from "$appt".

  ```
  $appt alarms }
  ```

  The following code will set alarms to occur at the appointment start time, and also 5, 10, and 15 minutes before the appointment starts.

  ```
  $appts alarms {0 5 10 15}
  ```

- *item* **is appt**

  Returns true iff *item* is an appointment.

- *item* **is note**

  Returns true iff *item* is a notice.

- *item* **calendar**

  If *item* is contained in a calendar, return the name of that calendar's file. Otherwise return the empty string.

- *item* **text**

  Return text for *item*.

- *item* **text** *text*

  Replace *item*'s text with *text*.

- *item* **earlywarning**

  Items start appearing in item listings some number of days before their actual occurrence. For example, a birthday notice may start appearing five days early to give you enough time to go buy a birthday card. This operation returns the number of days of early warning you get for *item*.

- *item* **earlywarning** *days*

  Set the early warning for *item* to *days*. *days* must be non-negative.

- *item* **owner**

  Return the name of *item*'s owner.

- *item* **owner** *o*

  Make *o* the owner of *item*.

- *item* **owned**

  Returns true iff *item* is owned by the current user.

- *item* **own**

  Make the current user the owner of *item*.

- *item* **hilite**

  Return the highlight mode for *item*. "always" means all item occurrences should be highlighted. "never" means no item occurrences should be highlighted. "expire means only occurrences on or after today should be highlighted. "holiday" means that all item occurrences should be highlighted, but as holidays.

- *item* **hilite** *mode*

  Set the highlight mode for *item* to *mode*. For example, the following code will create an item for Christmas day and add it to the calendar.

  ```
  set i [notice]
  $i monthrepeat 12 [date make 25 12 1994]
  $i text {Christmas}
  $i hilite holiday
  cal add $i
  ```

- *item* **todo**

  Ical supports *todo* items. If a todo item occurs in the past, that occurrence is automatically moved forward to today every day until the item is deleted, or marked as a non-todo item. This operation returns true iff *item* is a todo item.

- *item* **todo 1**

  This operation makes *item* a todo item.

- *item* **todo 0**

  This operation makes *item* a non-todo item.

- *item* **is_done**

  This operation returns true iff *item* is a todo item that has been marked as done.

- *item* **done 1**

  This operation makes *item* a todo item (if it isn't one already), and also marks it as done. A done todo item stops moving forward every day.

- *item* **done 0**

  This operation marks the item as not done. If the item is also a todo item, it will now move forward every day.

- *item* **option** *name*

  Look up the value associated with *name* in the named item *item*. An error is raised if the named option does not exist in the item. Example —

  ```
  set priority normal
  catch {set priority [$item option Priority]}
  ```

- *item* **option** *name value*

  Set the value of the option named *name* to *value* in the named item *item*. Example —

  ```
  $item option Priority high
  ```

5 Dates

Dates are represented in Tcl code as integers from some unspecified date. Therefore the "expr" command can be used to manipulate dates by addition and subtraction. Here is a list of other supported operations on dates.

- **date make** *day month year*

  Returns the date for the specified *day*, *month*, and *year*. *day*, *month*, and *year* should all be integers. This operation raises an error if the date specification is invalid. Here is an example of a date creation —

  ```
  # March 5, 1994.
  set d [date make 5 3 1994]
  ```

- **date today**

  Returns today's date.

- **date first**

  Returns the first date that can be legally represented by the ical implementation.

- **date last**

  Returns the last date that can be legally represented by the ical implementation.

- **date monthsize** *date*

  Return the number of days in the month containing *date*.

- **date monthday** *date*

  Returns the day of the month component of *date* as an integer in the range 1...31.

- **date weekday** *date*

  Returns the day of the week component of *date* as an integer in the range 1...7. One corresponds to Sunday, and Seven corresponds to Saturday.

- **date month** *date*

  Returns the month component of *date* as an integer in the range 1...12.

- **date year** *date*

  Returns the year component of *date* as an integer.

- **date split** *date*

  Returns a four element list of the components of *date*. The first element is the day of the month, the second element is the day of the week, the third element is the month, and the fourth element is the year.

- **date extract** *s d pre post*

  Returns true iff a date specification is successfully parsed from *s*. The parsed date is stored in the variable named by *d*. The portion of *s* before the parsed date specification is stored in the variable named by *pre*, and the portion of *s* after the parsed date specification is stored in the variable named by *post*. For example, the following code will store today's date in the variable "date", the string "before" in the variable "pre", and the string "after" in the variable "post".

  ```
  date extract {before today after} date pre post
  ```

6 Times

Times are represented in Tcl code as real numbers that give the number of seconds since some unspecified time. Therefore the "expr" command can be used to manipulate time values by addition and subtraction. Here is a list of other supported operations on times.

- **time now**

  Returns the current time.

- **time date** *time*

  Returns the date on which *time* occurs.

- **time hour** *time*

  Returns the hour of the day represented by *time* as an integer in the range 0. . . 23.

- **time minute** *time*

  Returns the minute of the hour represented by *time* as an integer in the range 0. . . 59.

- **time second** *time*

  Returns the second of the minute represented by *time* as an integer in the range 0. . . 59.

- **time millisecond** *time*

  Returns the millisecond of the second represented by *time* as an integer in the range 0. . . 999.

- **time split** *time*

  Returns a four element list of the components of *time*. The first element is the hour, the second element is the minute, the third element is the second, and the fourth element is the millisecond.

- **time extract_time** *s t pre post*

  Returns true iff a time of day specification is successfully parsed from *s*. The parsed time is stored in the variable named by *d* as the number of seconds since midnight. The portion of *s* before the parsed time is stored in the variable named by *pre*, and the portion of *s* after the parsed time is stored in the variable named by *post*. For example, the following code will store 180 (for "3am") in the variable "time, the string "X" in the variable "pre", and the string "Y" in the variable "post".

  ```
  time extract_time {X 3am Y} time pre post
  ```

- **time extract_range** *s start finish pre post*

  Returns true iff a time range specification is successfully parsed from *s*. The parsed time is stored in the variables named by *start* and *finish* as the number of seconds since midnight. The portion of *s* before the parsed range is stored in the variable named by *pre*, and the portion of *s* after the parsed time is stored in the variable named by *post*. For example, the following code will store 180 in the variable "s", 240 in the variable "f", the string "X" in the variable "pre", and the string "Y" in the variable "post".

  ```
  time extract_range {X 3-4am Y} s f pre post
  ```

7 Customization via Hooks

Ical provides a number of hooks. Users can attach code to these hooks to customize ical behavior. Code is attached to hooks by one of the following commands.

- **append-hook** *hook* {*varnames...*} {*code*}

  Code will be executed when the named *hook* is invoked by ical. Ical will provide a number of arguments when it invokes *hook*. (The number of arguments will depend on the particular hook being invoked.) The variables named by *varnames...* will be assigned the arguments supplied in the hook invocation before *code* is executed.

  For example, ical calls the "dayview-startup" hook when it creates a new calendar window. The hook is supplied one argument, the window being created. The following code changes the window to have a vertical layout by moving the appointment subwindow from the right side of the window to the bottom.

  ```
  append-hook dayview-startup {w} {
      pack [$w window].al -side bottom
  }
  ```

- **prepend-hook** *hook* {*varnames...*} {*code*}

  This command is very similar to "append-hook". The only difference is that the specified *code* will be executed before any code that is currently attached to *hook*. (*Code* specified by an "append-hook" command is executed after code that is already attached to the hook.)

7.1 General Hooks

Here is a list of some general hooks for ical.

- **ical-startup**

  This hook is invoked without any arguments when ical starts up.

- **ical-exit**

  This hook is invoked without any arguments just before ical exits.

- **item-create** *item*

  This hook is invoked when a new item is created. The created item is passed as the only argument to attached code.

- **todo-item-done** *item*

  This hook is invoked when a todo item is marked as done. The done item is passed as the only argument to attached code.

- **alarm-fire** *appt*

  This hook is invoked whenever an alarm is generated for an appointment. The appointment object for which the alarm is generated is passed as the only argument to the attached code.

7.2 Day Window Hooks

The following hooks are invoked when interesting things happen to a window that displays the items for a particular day. These windows are referred to as *dayviews* in the rest of the document. The object representing the dayview is passed as the first argument to these hooks. See Section 8 for a description of the dayview interface.

- **dayview-startup** *view*

  This hook is invoked when a dayview is created.

- **dayview-close** *view*

  This hook is invoked just before a dayview is deleted.

- **dayview-set-date** *view date*

  This hook is invoked when the date displayed in a dayview is changed. The new date is passed as the second argument to the hook.

- **dayview-focus** *view item*

  This hook is invoked when an item displayed in a dayview is selected. The selected item is passed as the second argument to the hook.

- **dayview-unfocus** *view*

  This hook is invoked when an item displayed in a dayview is unselected.

8 Day Windows

This section is incomplete!

This section describes the operations available on objects that represent the main ical windows. These windows display the items for a particular day.

9 Common Editing Actions

This section is incomplete!

This section describes the common editing operations that can be invoked from user customizations. These operations build on the calendar and item interfaces described in earlier sections. Use these operations when you customize ical's user interface because these operations provide a consistent editing interface to the user. For example, the operations described in this section produce friendly messages if the user tries to edit an item in a read-only calendar. By contrast, the low-level interfaces described in Sections 3 and 4 generate a stack trace if they are passed invalid arguments.

10 Miscellaneous Commands

The C++ code exports a few miscellaneous commands to tcl code.

- **wbeep** *volume*

  Produces a beep at the specified *volume*. The volume should be an integer in the range -100…100. A volume of -100 is the quietest. A volume of 100 is the loudest. This command is only available when ical is running on an X display.

A few more commands are implemented in C++ to speed-up some frequent computations. The specifications for these commands are given in the C++ code itself.