Using Libical

Eric Busboom (eric@softwarestudio.org) January 2001

Contents

1 Introduction

Libical is an Open Source implementation of the iCalendar protocols and protocol data units. The iCalendar specification describes how calendar clients can communicate with calendar servers so users can store their calendar data and arrange meetings with other users.

Libical implements RFC2445, RFC2446 and some of RFC2447 and the CAP draft.

This documentation assumes that you are familiar with the iCalendar standards RFC2445 and RFC2446. these specifications are online on the CALSCH webpage at:

```
http://www.imc.org/ietf-calendar/
```

1.1 The libical project

This code is under active development. If you would like to contribute to the project, you can contact me, Eric Busboom, at eric@softwarestudio.org. The project has a webpage at

```
http://softwarestudio.org/libical/index.html
```

and a mailing list that you can join by sending the following mail:

```
To: minimalist@softwarestudio.org
Subject: subscribe libical
```

1.2 License

The code and datafiles in this distribution are licensed under the Mozilla Public License. See http://www.mozilla.org/NPL/MPL-1.0.html for a copy of the license. Alternately, you may use libical under the terms of the GNU Library General Public License. See http://www.fsf.org/copyleft/lesser.html for a copy of the LGPL.

This dual license ensures that the library can be incorporated into both proprietary code and GPL'd programs, and will benefit from improvements made by programmers in both realms. I will only accept changes into my version of the library if they are similarly dual-licensed.

1.3 Example Code

A lot of the documentation for this library is in the form of example code. These examples are in the "examples" directory of the distribution. Also look in "src/test" for additional annotated examples.

2 Building the Library

Libical uses autoconf to generate makefiles. It should built with no adjustments on Linux, FreeBSD and Solaris under gcc. Some version have been successfully been build on MacOS, Solaris, UnixWare, And Tru64 UNIX without gcc, but you may run into problems with a particular later version.

For a more complete guide to building the library, see the README file in the distribution.

3 Structure

The iCal calendar model is based on four types of objects: components, properties, values and parameters.

Properties are the fundamental unit of information in iCal, and they work a bit like a hash entry, with a constant key and a variable value. Properties may also have modifiers, called parameters. In the iCal content line

```
ORGANIZER;ROLE=CHAIR:MAILTO:mrbig@host.com
```

The property name is "ORGANIZER," the value of the property is "mrbig@host.com" and the "ROLE" parameter specifies that Mr Big is the chair of the meetings associated with this property.

Components are groups of properties that represent the core objects of a calendar system, such as events or timezones. Components are delimited by "BEGIN" and "END" tags.

When a component is sent across a network, if it is un-encrypted, it will look something like:

```
BEGIN:VCALENDAR
METHOD:REQUEST
PRODID: -//hacksw/handcal//NONSGML v1.0//EN
BEGIN:VEVENT
DTSTAMP:19980309T231000Z
UID:guid-1.host1.com
ORGANIZER;ROLE=CHAIR:MAILTO:mrbig@host.com
ATTENDEE;RSVP=TRUE;ROLE=REQ-PARTICIPANT;CUTYPE=GROUP:
  MAILTO:employee-A@host.com
DESCRIPTION:Project XYZ Review Meeting
CATEGORIES:MEETING
CLASS:PUBLIC
CREATED:19980309T130000Z
SUMMARY:XYZ Project Review
DTSTART;TZID=US-Eastern:19980312T083000
DTEND;TZID=US-Eastern:19980312T093000
LOCATION:1CP Conference Room 4350
END:VEVENT
END:VCALENDAR
```

Note that components can be nested; this example has both a VCALENDAR and a VEVENT component, one nested inside the other.

3.1 Core iCal classes

Libical is an object-based, data-oriented library. Nearly all of the routines in the library are associated with an opaque data types and perform some operation on that data type. Although the library does not actually have classes, we will use those terms since the behavior of these associations of data and routines is very similar to a class.

3.1.1 Properties

Properties are represented with the icalproperty class and its many "derived" classes with on "derived" class per property type in RFC2445. Again, there is no actual inheritance relations, but there are clusters of routines that make this term useful. A property is a container for a single value and a set of parameters.

3.1.2 Components

In libical, components are represented with the icalcomponent class. Icalcomponent is a container for a set of other components and properties.

3.1.3 Values

Values are represented in a similar way to properties; a base class and many "derived " classes. A value is essentially a abstract handle on a single fundamental type, a structure or a union.

3.1.4 Parameters

Parameters are represetned in a similar way to properties, except that they contain only one value

3.2 Other elements of libical

In addition to the core iCal classes, libical has many other types, structures, classes that aid in creating and using iCal components.

3.2.1 Enumerations and types

Libical is strongly typed, soo every component, property, parameter, and value type has an enumeration, and some have an associated structure or union.

3.2.2 The parser

The libical parser offers a variety of ways to convert RFC2445 text into a libical iinsteral component structure. the parser can parse blocks of text as a string, or it can parse lin-by-line.

3.2.3 Error objects

Libical has a substantial error reporting system for both programming errors and component usage errors.

3.2.4 Memory Management

Since many of libicals interfaces return strings, the library has its own memory management system to elimiate the need to free every string returned from the libraru.

3.2.5 Storage classes

The library also offers several classes to store components to flies, memory or databases.

4 Differences From RFCs

Libical has been designed to follow the standards as closely as possible, so that the key objects in the standards are also key objects in the library. However, there are a few areas where the specifications are (arguably) irregular, and following them exactly would result in an unfriendly interface. These deviations make libical easier to use by maintaining a self-similar interface.

4.1 Pseudo Components

Libical defines components for groups of properties that look and act like components, but are not defined as components in the specification. XDAYLIGHT and XSTANDARD are notable examples. These pseudo components group properties within the VTIMEZONE components. For instanace, the timezone properties associated with daylight savings time starts with "BEGIN:DAYLIGHT" and ends with "END:DAYLIGHT, just like other components, but is not defined as a component in RFC2445. (See RFC2445, page 61) In Libical,this grouping is represented by the XDAYLIGHT component. Standard iCAL components all start with the letter "V," while pseudo components start with"X."

There are also pseudo components that are conceptually derived classes of VALARM. RFC2446 defines what properties may be included in each component, and for VALARM, the set of properties it may have depends on the value of the ACTION property.

For instance, if a VALARM component has an ACTION property with the value of "AUDIO," the component must also have an "ATTACH" property. However, if the ACTION value is "DISPLAY," the component must have a DESCRIPTION property.

To handle these various, complex restrictions, libical has pseudo components for each type of alarm: XAUDIOALARM, XDISPLAYALARM, XEMAILALARM and XPROCEDUREALARM.

4.2 Combined Values

Many values can take more than one type. TRIGGER, for instance, can have a value type of with DURATION or of DATE-TIME. These multiple types make it difficult to create routines to return the value associated with a property.

It is natural to have interfaces that would return the value of a property, but it is cumbersome for a single routine to return multiple types. So, in libical, properties that can have multiple types are given a single type that is the union of their RFC2445 types. For instance, in libical, the value of the TRIGGER property resolves to struct icaltriggertype. This type is a union of a DURATION and a DATE-TIME.

4.3 Multi-Valued Properties

Some properties, such as CATEGORIES have only one value type, but each CATEGORIES property can have multiple value instances. This also results in a cumbersome interface – CATEGORIES accessors would have to return a list while all other accessors returned a single value. In libical, all properties have a single value, and multi-valued properties are broken down into multiple single valued properties during parsing. That is, an input line like,

```
CATEGORIES: work, home
```

becomes in libical's internal representation

```
CATEGORIES: work
CATEGORIES: home
```

Oddly, RFC2445 allows some multi-valued properties (like FREEBUSY) to exist as both a multi-values property and as multiple single value properties, while others (like CATEGORIES) can only exist as single multi-valued properties. This makes the internal representation for CATEGORIES illegal. However when you convert a component to a string, the library will collect all of the CATEGORIES properties into one.

5 Using libical

5.1 Creating Components

There are three ways to create components in Libical: creating individual objects and assembling them, building entire objects in massive vaargs calls, and parsing a text file containing iCalendar data.

5.1.1 Constructor Interfaces

Using constructor interfaces, you create each of the objects separately and then assemble them in to components:

```
icalcomponent *event;
icalproperty *prop;
icalparameter *param;
struct icaltimetype atime;
event = icalcomponent_new(ICAL_VEVENT_COMPONENT);
prop = icalproperty_new_dtstamp(atime) ;
icalcomponent_add_property(event, prop);
prop = icalproperty_new_uid(''guid-1.host1.com'') );
icalcomponent_add_property(event,prop);
prop=icalproperty_new_organizer(''mrbig@host.com'');
param = icalparameter_new_role(ICAL_ROLE_CHAIR)
icalproperty_add_parameter(prop, param);
icalcomponent_add_property(event,prop);
```

Notice that libical uses a semi-object-oriented style of interface. Most things you work with are objects, that are instantiated with a constructor that has "new" in the name. Also note that, other than the object reference, most structure data is passed in to libical routines by value. Libical has some complex but very regular memory handling rules. These are detailed in section 5.5 ().

If any of the constructors fail, they will return 0. If you try to insert 0 into a property or component, or use a zero-valued object reference, libical will either silently ignore the error or will abort with an error message. This behavior is controlled by a compile time flag (ICAL_ERRORS_ARE_FATAL), and will abort by default.

5.1.2 vaargs Constructors

There is another way to create complex components, which is arguably more elegant, if you are not horrified by varargs. The varargs constructor interface allows you to create intricate components in a single block of code. Here is the previous examples in the vaargs style.

```
calendar =
    icalcomponent_vanew(
        ICAL_VCALENDAR_COMPONENT,
        icalproperty_new_version(''2.0''),
        icalproperty_new_prodid(
             ''-//RDU Software//NONSGML HandCal//EN''),
        icalcomponent_vanew(
            ICAL_VEVENT_COMPONENT,
            icalproperty_new_dtstamp(atime),
            icalproperty_new_uid(''guid-1.host1.com''),
            icalproperty_vanew_organizer(
                ''mrbig@host.com''),
                icalparameter_new_role(ICAL_ROLE_CHAIR),
                0
                ),
            icalproperty_vanew_attendee(
                ''employee-A@host.com'',
                icalparameter_new_role(
                    ICAL_ROLE_REQPARTICIPANT),
                icalparameter_new_rsvp(1),
                icalparameter_new_cutype(ICAL_CUTYPE_GROUP),
                0
                ),
            icalproperty_new_location(
               1CP Conference Room 4350),
            0
            ),
        0
        );
```

This form is similar to the constructor form , except that the constructors have "vanew" instead of "new" in the name. The arguments are similar too, except that the component constructor can have a list of properties, and the property constructor can have a list of parameters. Be sure to terminate every list with a '0', or your code will crash, if you are lucky.

5.1.3 Parsing Text Files

The final way to create components will probably be the most common; you can create components from RFC2445 compliant text. If you have the string in memory, use

```
icalcomponent* icalparser_parse_string(char* str);
```

If the string contains only one component, the parser will return the component in libical form. If the string contains multiple components, the multiple components will be returned as the children of an ICAL_XROOT_COMPONENT component.

Parsing a whole string may seem wasteful if you want to pull a large component off of the network or from a file; you may prefer to parse the component line by line. This is possible too by using:

```
icalparser* icalparser_new();
void icalparser_free(icalparser* parser);
icalparser_get_line(parser,read_stream);
icalparser_add_line(parser,line);
icalparser_set_gen_data(parser,stream)
```

These routines will construct a parser object to which you can add lines of input and retrieve any components that the parser creates from the input. These routines work by specifing an adaptor routine to get string data from a source. For an example:

```
char* read_stream(char *s, size_t size, void *d)

  char *c = fgets(s,size, (FILE*)d);
  return c;

main()
  char* line;
  icalcomponent *c;
  icalparser *parser = icalparser_new();
  FILE* stream = fopen(argv1,r);
  icalparser_set_gen_data(parser,stream);
  do
    line = icalparser_get_line(parser,read_stream);
    c = icalparser_add_line(parser,line);
    if (c != 0)
     printf(s,icalcomponent_as_ical_string(c));
     icalparser_claim(parser);
     printf(n---------------n);
     icalcomponent_free(c);

   while ( line != 0);
```

The parser object parameterizes the routine used to get input lines with icalparser_set_gen_data() andicalparser_get_line(). In this example, the routine read_stream() will fetch the next line from a stream, with the stream passed in as the void* parameter d. The parser calls read_stream() from icalparser_get_line(), but it also needs to know what stream to use. This is set by the call to icalparser_set_gen_data(). By using a different routine for read_stream or passing in different data with icalparser_set_gen_data, you can connect to any data source.

Using the same mechanism, other implementations could read from memory buffers, sockets or other interfaces.

Since the example code is a very common way to use the parser, there is a convenience routine;

```
icalcomponent* icalparser_parse(icalparser *parser,
               char* (*line_gen_func)(char *s, size_t size,  void* d))
```

To use this routine, you still must construct the parser object and pass in a reference to a line reading routine. If the parser can create a single component from the input, it will return a pointer to the newly constructed component. If the parser can construct multiple components from the input, it will return a reference to an XROOT component (of type ICAL_XROOT_COMPONENT.) This XROOT component will hold all of the components constructed from the input as children.

5.2 Accessing Components

Given a reference to a component, you probably will want to access the properties, parameters and values inside. Libical interfaces let you find sub-component, add and remove sub-components, and do the same three operations on properties.

5.2.1 Finding Components

To find a sub-component of a component, use:

```
icalcomponent* icalcomponent_get_first_component(
                                   icalcomponent* component,
                                   icalcomponent_kind kind);
```

This routine will return a reference to the first component of the type 'kind.' The key kind values, listed in icalenums.h are:

```
ICAL_ANY_COMPONENT
ICAL_VEVENT_COMPONENT
ICAL_VTODO_COMPONENT
ICAL_VJOURNAL_COMPONENT
ICAL_VCALENDAR_COMPONENT
ICAL_VFREEBUSY_COMPONENT
ICAL_VALARM_COMPONENT
```

These are only the most common components; there are many more listed in icalenums.h.

As you might guess, if there is more than one subcomponent of the type you have chosen, this routine will return only the first. to get at the others, you need to iterate through the component.

5.2.2 Iterating Through Components

Iteration requires a second routine to get the next subcomponent after the first:

```
icalcomponent* icalcomponent_get_next_component(
               icalcomponent* component,
               icalcomponent_kind kind);
```

With the 'first' and 'next' routines, you can create a for loop to iterate through all of a components subcomponents

```
  for(c = icalcomponent_get_first_component(comp,ICAL_ANY_COMPONENT);
         c != 0;
 c = icalcomponent_get_next_component(comp,ICAL_ANY_COMPONENT))

      do_something(c);
```

This code bit wil iterate through all of the subcomponents in 'comp' but you can select a specific type of component by changing ICAL_ANY_COMPONENT to another component type.

5.2.3 Using Component Iterators

The iteration model in the previous section requires the component to keep the state of the iteration. So, you could not use this model to perform a sorting operations, since you'd need two iterators and there is only space for one. If you ever call icalcomponent_get_first_component() when an iteration is in progress, the pointer will be reset to the beginning.

To solve this problem, there are also external iterators for components. The routines associated with these external iterators are:

```
icalcompiter icalcomponent_begin_component(icalcomponent* component, icalcomponent_kind kind);
icalcompiter icalcomponent_end_component(icalcomponent* component, icalcomponent_kind kind);
icalcomponent* icalcompiter_next(icalcompiter* i);
icalcomponent* icalcompiter_prior(icalcompiter* i);
icalcomponent* icalcompiter_deref(icalcompiter* i);
```

The _begin_() and _end_() routines return a new iterator that points to the beginning and ending of the list of subcomponent for the given component, and the kind argument works like the kind argument for internal iterators.

After creating an iterators, use _next_() and _prior_() to step forward and backward through the list and get the component that the iterator points to, and use _deref() to return the component that the iterator points to without moving the iterator. All routines will return 0 when they move to point off the end of the list.

Here is an example of a loop using these routines:

```
for(
   i = icalcomponent_begin_component(impl-cluster,ICAL_ANY_COMPONENT);
   icalcompiter_deref(i)!= 0;
   icalcompiter_next(i)
)
      icalcomponent *this = icalcompiter_deref(i);
```

5.2.4 Removing Components

Removing an element from a list while iterating through the list with the internal iterators can cause problems, since you will probably be removing the element that the internal iterator points to. The _remove() routine will keep the iterator valid by moving it to the next component, but in a normal loop, this will result in two advances per iteration, and you will remove only every other component. To avoid the problem, you will need to step the iterator ahead of the element you are going to remove, like this:

```
for(c = icalcomponent_get_first_component(parent_comp,ICAL_ANY_COMPONENT);
       c != 0;
       c = next

    next = icalcomponent_get_next_component(parent_comp,ICAL_ANY_COMPONENT);
    icalcomponent_remove_component(parent_comp,c);
```

Another way to remove components is to rely on the side effect of icalcomponent_remove_component: if component iterator in the parent component is pointing to the child that will be removed, it will move the iterator to the component after the child. The following code will exploit this behavior:

```
icalcomponent_get_first_component(parent_comp,ICAL_VEVENT_COMPONENT);
while((c=icalcomponent_get_current_component(c)) != 0 )
   if(icalcomponent_isa(c) == ICAL_VEVENT_COMPONENT)
      icalcomponent_remove_component(parent_comp,inner);
    else
      icalcomponent_get_next_component(parent_comp,ICAL_VEVENT_COMPONENT);
```

5.2.5 Working with properties and parameters

Finding, iterating and removing properties works the same as it does for components, using the property-specific or parameter-specific interfaces:

```
icalproperty* icalcomponent_get_first_property(
     icalcomponent* component,
     icalproperty_kind kind);
icalproperty* icalcomponent_get_next_property(
     icalcomponent* component,
     icalproperty_kind kind);
void icalcomponent_add_property(
     icalcomponent* component,
     icalproperty* property);
void icalcomponent_remove_property(
     icalcomponent* component,
     icalproperty* property);
```

For parameters:

```
icalparameter* icalproperty_get_first_parameter(
     icalproperty* prop,
     icalparameter_kind kind);
icalparameter* icalproperty_get_next_parameter(
     icalproperty* prop,
     icalparameter_kind kind);
void icalproperty_add_parameter(
     icalproperty* prop,
     icalparameter* parameter);
void icalproperty_remove_parameter(
     icalproperty* prop,
     icalparameter_kind kind);
```

Note that since there should be only one parameter of each type in a property, you will rarely need to use icalparameter_get_nect_paameter.

5.2.6 Working with values

Values are typically part of a property, although they can exist on their own. You can manipulate them either as part of the property or independently.

The most common way to work with values to is to manipulate them from they properties that contain them. This involves fewer routine calls and intermediate variables than working with them independently, and it is type-safe.

For each property, there are a _get_ and a _set_ routine that access the internal value. For instanace, for the UID property, the routines are:

```
void icalproperty_set_uid(icalproperty* prop, const char* v)
const char* icalproperty_get_uid(icalproperty* prop)
```

For multi-valued properties, like ATTACH, the value type is usually a struct or union that holds both possible types.

If you want to work with the underlying value object, you can get and set it with:

```
icalvalue* icalproperty_get_value (icalproperty* prop)
void icalproperty_set_value(icalproperty* prop, icalvalue* value);
```

Icalproperty_get_value() will return a reference that you can manipulate with other icalvalue routines. Most of the time, you will have to know what the type of the value is. For instance, if you know that the value is a DATETIME type, you can manipulate it with:

```
struct icaltimetype icalvalue_get_datetime(icalvalue* value);
void icalvalue_set_datetime(icalvalue* value, struct icaltimetype v);
```

When working with an extension property or value (and X-PROPERTY or a property that has the parameter VALUE=x-name) the value type is always a string. To get and set the value, use:

```
void icalproperty_set_x(icalproperty* prop, char* v);
char* icalproperty_get_x(icalproperty* prop);
```

All X properties have the type of ICAL_X_PROPERTY, so you will need these routines to get and set the name of the property:

```
char* icalproperty_get_x_name(icalproperty* prop)
void icalproperty_set_x_name(icalproperty* prop, char* name);
```

5.2.7 Checking Component Validity

RFC 2446 defines rules for what properties must exist in a component to be used for transferring scheduling data. Most of these rules relate to the existence of properties relative to the METHOD property, which declares what operation a remote receiver should use to process a component. For instance, if the METHOD is REQUEST and the component is a VEVENT, the sender is probably asking the receiver to join in a meeting. In this case, RFC2446 says that the component must specify a start time (DTSTART) and list the receiver as an attendee (ATTENDEE).

Libical can check these restrictions with the routine:

```
int icalrestriction_check(icalcomponent* comp);
```

This routine returns 0 if the component does not pass RFC2446 restrictions, or if the component is malformed. The component you pass in *must* be a VCALENDAR, with one or more children, like the examples in RFC2446.

When this routine runs, it will insert new properties into the component to indicate any errors it finds. See section 6.5.3, X-LIC-ERROR for more information about these error properties.

5.2.8 Converting Components to Text

To create an RFC2445 compliant text representation of an object, use one of the *_as_ical_string() routines:

```
char* icalcomponent_as_ical_string (icalcomponent* component)
char* icalproperty_as_ical_string (icalproperty* property)
char* icalparameter_as_ical_string (icalparameter* parameter)
char* icalvalue_as_ical_string (icalvalue* value)
```

In most cases, you will only use icalcomponent_as_ical_string (), since it will cascade and convert all of the parameters, properties and values that are attached to the root component.

Icalproperty_as_ical_string() will terminate each line with the RFC2445 specified line terminator "n" However, if you compile with the symbol ICAL_UNIX_NEWLINE undefined, (it is defined by default) it will terminate lines with "nr"

Remember that the string returned by these routines is owned by the library, and will eventually be re-written. You should copy it if you want to preserve it.

5.3 Time

5.3.1 Time structure

LIbical defines it's own time structure for storing all dates and times. It would have been nice to re-use the C library's *struct tm,* but that structure does not differentiate between dates and times, and between local time and UTC. The libical structure is:

```
struct icaltimetype
  int year;
  int month;
  int day;
  int hour;
  int minute;
  int second;
  int is_utc; /* 1- time is in UTC timezone */
  int is_date; /* 1 - interpret this as date. */ ;
```

The year, month, day, hour, minute and second fields how the broken-out time values. The is_utc field distinguishes between times UTC and a local time zone. The is_date field indicates if the intra-day fields hold valid data.

5.3.2 Creating time structures

There are several ways to create a new icaltimetype structure:

```
struct icaltimetype icaltime_from_string(const char* str);
struct icaltimetype icaltime_from_timet(time_t v, int is_date);
struct icaltimetype icaltime_from_int(int v, int is_date, int is_utc);
```

Icaltime_from_string takes any RFC2445 compliant time string:

```
struct icaltimetype tt = icaltime_from_string(19970101T103000);
```

Icaltime_from_timet takes a timet value, representing seconds past the POSIX epoch, and a flag to indicate if the time is a date. Dates have an identical structure to a time, but they time portion (hours, minuts and seconds) is always 00:00:00. Dates act differently in sorting an comparision, and they have a different string representation in RFC2445.

The icaltime_from_int is like icaltime_from_timet, but with an arbitrary epoch. This routine was a mistake and is deprecated.

5.3.3 Time manipulating routines

The null time value is used to indicate that the data in the structure is not a valid time.

```
struct icaltimetype icaltime_null_time(void);
int icaltime_is_null_time(struct icaltimetype t);
```

It is sensible for the broken-out time fields to contain values that are not permitted in an ISO compliant time string. For instance, the seconds field can hold values greater than 59, and the hours field can hold values larger than 24. The excessive values will be rolled over into the next larger field when the structure is normalized.

```
struct icaltimetype icaltime_normalize(struct icaltimetype t);
```

Normalizing allows you to do arithmetic operations on time values.

```
struct icaltimetype tt = icaltime_from_string("19970101T103000");
tt.days +=3
tt.second += 70;
tt = icaltime_normalize(tt);
```

There are several routines to get the day of the week or month, etc, from a time structure.

```
short icaltime_day_of_year(struct icaltimetype t);
struct icaltimetype icaltime_from_day_of_year(short doy, short year);
short icaltime_day_of_week(struct icaltimetype t);
short icaltime_start_doy_of_week(struct icaltimetype t);
short icaltime_week_number(short day_of_month, short month, short year);
struct icaltimetype icaltime_from_week_number(short week_number, short year);
short icaltime_days_in_month(short month,short year);
```

Two routines convert time structures to and from the number of seconds since the POSIX epoch. The is_date field indicates whether or not the hour, minute and second fields should be used in the conversion.

```
struct icaltimetype icaltime_from_timet(time_t v, int is_date);
time_t icaltime_as_timet(struct icaltimetype);
```

The compare routine works exactly like strcmp, but on time structures.

```
int icaltime_compare(struct icaltimetype a,struct icaltimetype b);
```

The following routines convert between UTC and a named timezone. The tzid field must be a timezone name from the Olsen database, such as "America/Los_Angeles."

The utc_offset routine returns the offset of the named time zone from UTC, in seconds.

The tt parameter in the following routines indicates the date on which the conversion should be made. The tt parameter is necessary because timezones have many different rules for when daylight savings time is used, and these rules can change over time. So, for a single timezone one year may have daylight savings time on March 15, but for other years March 15 may be standard time, and some years may have standard time all year.

```
int icaltime_utc_offset(struct icaltimetype tt, char* tzid);
int icaltime_local_utc_offset();
struct icaltimetype icaltime_as_utc(struct icaltimetype tt,char* tzid);
struct icaltimetype icaltime_as_zone(struct icaltimetype tt,char* tzid);
struct icaltimetype icaltime_as_local(struct icaltimetype tt);
```

5.4 Storing Objects

The libical distribution includes a separate library, libicalss, that allows you to store iCal component data to disk in a variety of ways. This library also includes code to implement the CSTP protocol of CAP and has some routines for deciphering incomming messages.

The file storage routines are organized in an inheritance heirarchy that is rooted in icalset, with the derived class icalfileset and icaldirset. Icalfileset stores components to a file, while icaldirset stores components to multiple files, one per month based on DTSTAMP. Other storages classess, for storage to a heap or a mysql database are planned for the future.

All of the icalset derived classes have the same interface:

```
icaldirset* icaldirset_new(const char* path);
void icaldirset_free(icaldirset* store);
const char* icaldirset_path(icaldirset* store);
void icaldirset_mark(icaldirset* store);
icalerrorenum icaldirset_commit(icaldirset* store);
icalerrorenum icaldirset_add_component(icaldirset* store, icalcomponent* comp);
icalerrorenum icaldirset_remove_component(icaldirset* store, icalcomponent* comp);
int icaldirset_count_components(icaldirset* store, icalcomponent_kind kind);
icalerrorenum icaldirset_select(icaldirset* store, icalcomponent* gauge);
void icaldirset_clear(icaldirset* store);
icalcomponent* icaldirset_fetch(icaldirset* store, const char* uid);
int icaldirset_has_uid(icaldirset* store, const char* uid);
icalcomponent* icaldirset_fetch_match(icaldirset* set, icalcomponent *c);
icalerrorenum icaldirset_modify(icaldirset* store, icalcomponent *oldc, icalcomponent *newc);
icalcomponent* icaldirset_get_current_component(icaldirset* store);
icalcomponent* icaldirset_get_first_component(icaldirset* store);
icalcomponent* icaldirset_get_next_component(icaldirset* store);
```

5.4.1 Creating a new set

You can create a new set from either the base class or the direved class. From the base class use one of:

```
icalset* icalset_new_file(const char* path);
icalset* icalset_new_dir(const char* path);
icalset* icalset_new_heap(void);
icalset* icalset_new_mysql(const char* path);
```

You can also create a new set based on the derived class, For instance, with icalfileset:

```
icalfileset* icalfileset_new(const char* path);
icalfileset* icalfileset_new_open(const char* path, int flags, mode_t mode);
```

Icaset_new_file is identical to icalfileset_new. BOth routines will open an existing file for readinga and writing, or create a new file if it does not exist. Icalfilset_new_open takes the same arguments as the open() system routine and behaves in the same way.

The icalset and icalfilset objects are somewhat interchangable – you can use an icalfileset* as an argument to any of the icalset routines.

The following examples will all use icalfileset routines; using the other icalset derived classess will be similar.

5.4.2 Adding, Finding and Removing Components

To add components to a set, use:

```
icalerrorenum icalfileset_add_component(icalfileset* cluster, icalcomponent* child);
```

The fileset keeps an inmemory copy of the components, and this set must be written back to the file ocassionally. There are two routines to manage this:

```
void icalfileset_mark(icalfileset* cluster);
icalerrorenum icalfileset_commit(icalfileset* cluster);
```

Icalfileset_mark indicates that the in-memory components have changed. Calling the _add_component routine will call _mark automatically, but you may need to call it yourself if you have made a change to an existing component. The _commit routine writes the data base to disk, but only if it is marked. The _commit routine is called automatically when the icalfileset is freed.

To iterate through the components in a set, use:

```
icalcomponent* icalfileset_get_first_component(icalfileset* cluster);
icalcomponent* icalfileset_get_next_component(icalfileset* cluster);
icalcomponent* icalfileset_get_current_component (icalfileset* cluster);
```

These routines work like the corresponding routines from icalcomponent, except that their output is filtered through a gauge. A gauge is a test for the properties within a components; only components that pass the test are returned. A gauge can be constructed from a MINSQL string with:

```
icalgauge* icalgauge_new_from_sql(char* sql);
```

Then, you can add the gauge to the set with :

```
icalerrorenum icalfileset_select(icalfileset* store, icalgauge* gauge);
```

Here is an example that puts all of these routines together:

```
void test_fileset()

    icalfileset *fs;
    icalcomponent *c;
    int i;
    char *path = test_fileset.ics;
    icalgauge  *g = icalgauge_new_from_sql(
```

```
    SELECT * FROM VEVENT WHERE DTSTART  '20000103T120000Z' AND DTSTART = '20000106T120000Z');

fs = icalfileset_new(path);

for (i = 0; i!= 10; i++)
    c = make_component(i); /* Make a new component where DTSTART has month of i */
    icalfileset_add_component(fs,c);


icalfileset_commit(fs); /* Write to disk */

icalfileset_select(fs,g); /* Set the gauge to filter components */

for (c = icalfileset_get_first_component(fs);
     c != 0;
     c = icalfileset_get_next_component(fs))
    struct icaltimetype t = icalcomponent_get_dtstart(c);

    printf(sn,icaltime_as_ctime(t));

icalfileset_free(fs);
```

5.4.3 Other routines

There are several other routines in the icalset interface, but they not fully implemented yet.

5.5 Memory Management

Libical relies heavily on dynamic allocation for both the core objects and for the strings used to hold values. Some of this memory the library caller owns and must free, and some of the memory is managed by the library. Here is a summary of the memory rules.

1)

If the function name has new in it, the caller gets control of the memory. (such as icalcomponent_new(), or icalproperty_new_clone())

2)

If you got the memory from a routine with new in it, you must call the corresponding *_free routine to free the memory. (Use icalcomponent_free() to free objects created with icalcomponent_new())

3)

If the function name has add in it, the caller is transferring control of the memory to the routine. (icalproperty_add_parameter())

4)

If the function name has remove in it, the caller passes in a pointer to an object and after the call returns, the caller owns the object. So, before you call icalcomponent_remove_property(comp,foo), you do not own foo and after the call returns, you do.

5)

If the routine returns a string, libical owns the memory and will put it on a ring buffer to reclaim later. For example, icalcomponent_as_ical_string(). You'd better strdup() it if you want to keep it, and you don't have to delete it.

5.6 Error Handling

Libical has several error handling mechanisms for the various types of programming, semantic and syntactic errors you may encounter.

5.6.1 Return values

Many library routines signal errors through their return values. All routines that return a pointer, such as icalcomponent_new(), will return 0 (zero) on a fatal error. Some routines will return a value of enum icalerrorenum.

5.6.2 icalerrno

Most routines will set the global error value icalerrno on errors. This variable is an enumeration; permissible values can be found in libical/icalerror.h. If the routine returns an enum icalerrorenum, then the return value will be the same as icalerrno. You can use icalerror_strerror() to get a string that describes the error

5.6.3 X-LIC-ERROR and X-LIC-INVALID-COMPONENT

The library handles semantic and syntactic errors in components by inserting errors properties into the components. If the parser cannot parse incoming text (a syntactic error) or if the icalrestriction_check() routine indicates that the component does not meet the requirements of RFC2446 (a semantic error) the library will insert properties of the type X-LIC-ERROR to describe the error. Here is an example of the error property:

```
X-LIC-ERROR;X-LIC-ERRORTYPE=INVALID_ITIP :Failed iTIP restrictions for property DTSTART.
Expected 1 instances of the property and got 0
```

This error resulted from a call to icalrestriction_check(), which discovered that the component does not have a DTSTART property, as required by RFC2445.

There are a few routines to manipulate error properties:

The following data is supposed to be in a table. It looks OK in LyX, but does not format propertly in output.

Routine Purpose void icalrestriction_check() Check a component against RFC2446 and insert

error properties to indicate non compliance int icalcomponent_count_errors() Return the number of error properties in a component void icalcomponent_strip_errors() Remove all error properties in as

component void icalcomponent_convert_errors() Convert some error properties into

REQUESTS-STATUS proprties to indicate the inability to

process the component as an iTIP request.

The types of errors are listed in icalerror.h. They are:

```
ICAL_XLICERRORTYPE_COMPONENTPARSEERROR
ICAL_XLICERRORTYPE_PARAMETERVALUEPARSEERROR
ICAL_XLICERRORTYPE_PARAMETERNAMEPARSEERROR
ICAL_XLICERRORTYPE_PROPERTYPARSEERROR
ICAL_XLICERRORTYPE_VALUEPARSEERROR
ICAL_XLICERRORTYPE_UNKVCALPROP
ICAL_XLICERRORTYPE_INVALIDITIP
```

The libical parser will generate the error that end in PARSEERROR when it encounters garbage in the input steam. ICAL_XLICERRORTYPE_INVALIDITIP is inserted by icalrestriction_check(), and ICAL_XLICERRORTYPE_UNKVCALPROP is generated by icalvcal_convert() when it encounters a vCal property that it cannot convert or does not know about.

Icalcomponent_convert_errors() converts some of the error properties in a component into REQUEST-STATUS properties that indicate a failure. As of libical version0.18, this routine only convert *PARSEERROR errors and it always generates a 3.x (failure) code. This makes it more of a good idea than a really useful bit of code.

5.7 Naming Standard

Structures that you access with the "struct" keyword, such as "struct icaltimetype" are things that you are allowed to see inside and poke at.

Structures that you access though a typedef, such as "icalcomponent" are things where all of the data is hidden.

Component names that start with "V" are part of RFC 2445 or another iCal standard. Component names that start with "X" are also part of the spec, but they are not actually components in the spec. However, they look and act like components, so they are components in libical. Names that start with "XLIC" or "X-LIC" are not part of any iCal spec. They are used internally by libical.

Enums that identify a component, property, value or parameter end with "_COMPONENT," "_PROPERTY," "_VALUE," or "_PARAMETER"s

Enums that identify a parameter value have the name of the parameter as the second word. For instance: ICAL_ROLE_REQPARTICIPANT or ICAL_PARTSTAT_ACCEPTED.

The enums for the parts of a recurarance rule and request statuses are irregular.

6 Hacks and Bugs

There are a lot of hacks in the library – bits of code that I am not proud of and should probably be changed. These are marked with the comment string "HACK."