11 11 TEX LATEX

Using Libical

Eric Busboom (eric@softwarestudio.org) May 2000

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 for users can store their calendar data and arrange meetings with other users.

Libical implements RFC2445 and RFC2446. Eventually, it will also implement iRIP and CAP.

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 more annotated examples.

2 Building the Library

Libical uses autoconf to generate makefiles, although it uses none of the autoconf flags to influence the compilation. It should built with no adjustments on Linux, FreeBSD and Solaris.

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.

The central goal of libical is to parse iTIP data into an internal representation of Components, Properties, Parameters an Values, and to allow the user to manipulate the data in various ways

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

```
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
```

3.1 Core iCal classes

3.1.1 Components

3.1.2 Properties

3.1.3 Values

3.1.4 Parameters

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.

[1]

3.2.1 Enumerations

3.2.2 Types

3.2.3 The Parser

3.2.4 Restrictions

3.2.5 Error objects

3.2.6 Memory Management

3.2.7 Storage classes

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 keey 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 classess 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 cumbersone 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 Implementation Limitations

6 Using libical

6.1 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.

6.1.1 Constructor Interfaces

Using constructor interfaces, you create each of the objects seperately and them 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(strdup("guid-1.host1.com")) );
icalcomponent_add_property(event,prop);
prop=icalproperty_new_organizer(strdup("mrbig@host.com"));
param = icalparameter_new_role(ICAL_ROLE_CHAIR)
icalproperty_add_parameter(prop, param);
icalcomponent_add_property(event,prop);
```

While we are on this example, you should 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. Strings, of course, are passed in by reference, but libical will take ownership of the memory, so you had beter strdup() the data unless you want a core dump when the memory is freed for the second time. Libical has some complex but very regular memory handling rules. These are detailed in section 6.4 ().

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.

6.1.2 vaargs Constructors

There is another way to create complex components, which is arguable more elegant, if you are not horrified by varargs. The varargs constructor interface all you to create intricate components in a single block of text.

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

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

6.1.3 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);
```

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

```
icalparser_get_line(parser,read_stream);
icalparser_add_line(parser,line);
icalparser_set_gen_data(parser,stream)
```

For an example of how to use this type of parsing, see examples/parse_text.c for an example.

6.2 Accessing Components

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

6.2.1 Finding Components

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

```
icalproperty* 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.

6.2.2 Interating 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.

6.2.3 Removing Components

Libical component have internal iterators, so you can only have one iteration over a component at a time. Removing an element from a list while iterating through the list can cause problems, since you will probably be removing the element that the internal iterator points to. This will result in the iteration loop terminating immediately after removing the element. 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);
}
```

6.2.4 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);
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);
```

6.2.5 Getting Values

6.2.6 Setting Values

6.2.7 Getting Parameters

6.2.8 Setting Parameters

6.2.9 Removing Parameters

6.2.10 Checking Component Validity

6.3 Storing Objects

The libical distribution inclues a seperate library, libicalss, that allows you to store iCal component data to disk in a variety of ways. This library is documented seperately. (& currently, not at all.)

6.4 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 transfering 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. You'd better strdup() it if you want to keep it, and you don't have to delete it.

6.5 Error Handling

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

6.5.1 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.

6.5.2 icalerrno

Most routines will set the global error value icalerrno on errors. This variable is an enumeration; permissable 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

6.5.3 X-LIC-ERROR

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 requirments 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:

Routine Purpose void icalrestriction_check() Check a component against RFC2446 and insert error properties to indicate any non-complianes int icalcomponent_count_errors() Return the number of error properties in a component void icalcomponent_strip_errors() Remove all error properties in a component void icalcomponent_convert_errors() Convert some error properties into REQUEST-STATUS properties to indicate the inability to process the component as an iTIP request.

6.6 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.

7 Useful Recipies

Iteration

Copying components. Remember that you must clone or remove an object before putting in on another list.

Finding compliance errors

8 Performance

Checking restrictions is computationally expensive.

9 Hacks and Bugs

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