W. M. KECK OBSERVATORY KSD/KTL_MANUAL_28
Keck Software Document 28
Version 4
W. Lupton
14-Aug-96

KTL Programming Manual

File: 3.5.6 (KSDs), Library KSD Binder, Summit KSD Binder

Circulation: <circulation>

Table of Contents

1.0 Introduction

This document describes how to write KTL (Keck Task Library) applications. Refer to [3] for the original KTL design document; refer to Section 2.0, "What is KTL?," on page 3 for a brief description of KTL.

Most aspects of using KTL are illustrated with real examples. In addition, the appendices contain detailed descriptions of all user-callable routines, including some low-level ones which will occasionally be of use in applications.

In addition, there are brief sections on how to implement new KTL services (this will be a quite common requirement) and styles (this will be less common).

This document refers to KTL release 4.

2.0 What is KTL?

KTL is a set of routines which eases the job of writing applications which must interact with a variety of underlying sub-systems (known as *services*). A typical such application is an X Window user interface coordinating telescope and instruments. In order to connect to a service, application code specifies a service name---typically an instrument name---and a *style*, which defines the way in which the application will interact with the service. Two styles are currently supported: *keyword*, where the application reads and writes named keywords and the resulting inter-task message traffic is hidden; and *message*, where the application deals directly with messages. The keyword style is intended mainly for user interfaces, and the message style is intended mainly for lower-level applications.

KTL applications are event driven: a typical application first connects to all its desired services, then expresses interest in specified events. The application then enters an event dispatch loop in which it waits for events and calls the appropriate service's event-handling routine. Each event is associated with a callback routine which is invoked when the event occurs. Callback routines may (and typically do) interact with other sub-systems and KTL provides the means of doing so without blocking the application (vital for X Window user interfaces). This approach is a marriage of ideas culled from the X window, ADAM, Keck instrument and Keck telescope control systems.

A novel feature of KTL is that it knows nothing about any services or styles. Instead it defines a generic set of routines which must be implemented by all services and styles (`open()`, `ioctl()`, `read()`, `write()`, `event()`, `respond()` and `close()`) and activates shareable libraries at run-time. Services have been implemented (in both keyword and message styles) for HIRES (the Keck high resolution echelle spectrograph built by Lick Observatory), LWS (the Keck long wavelength spectrometer built by UC San Diego) and the Keck telescope. Each of these implementations uses different underlying message systems: the Lick MUSIC system, RPCs, and direct sockets (respectively). Services for the remaining three front-line Keck instruments will be implemented over the next few months.

3.0 Structure of KTL applications

In this section, the overall structure of a KTL application is discussed, with particular attention being given to callbacks. This is followed by brief descriptions of the various routines which may be used in KTL applications (detailed descriptions are to be found in the Appendices, together with descriptions of some additional low-level routines which may occasionally be useful to application programmers).

3.1 Main program

All KTL main programs have essentially the same structure:

```
#include "ktl.h"

ktl_open( service, style, flags, &handle );
more ktl_open() calls if other services are to be used;

context.callback = callback;
ktl_read( handle, KTL_CONTINUOUS, object_name, client_data, NULL, &context );
repeat for other objects and/or handles;

while ( !done ) {
      build fd_set and call select();
      dispatch non-KTL events;
      ktl_dispatch( handle );
      repeat for other handles, if any;
}

ktl_close( handle );
repeat for other handles, if any;
```

1 All KTL applications should include `ktl.h`, which contains all public KTL definitions. Note that `ktl.h` includes `<sys/types.h>`, `<stdio.h>`, `<stdlib.h>` and `<errno.h>`.

2 All KTL applications will call `ktl_open()` once per server to which they wish to connect. Simple applications like `show` and `modify` (Appendix A.1 on page 15) connect to only a single server. More complex applications like `xshow` (Appendix A.2 on page 16) connect to multiple servers.

3 Note that the term `server` is probably the best one to use here but is nevertheless potentially confusing: the model to keep in mind is that `ktl_open()` makes a *bidirectional* connection to an external agent. That agent may be both the source of events and the destination of commands.

4 Refer to Appendix A, "Examples of KTL applications," on page 15 for examples of the use of `ktl_open()` and to its detailed description in Appendix C.4, "ktl_open() --- Open a connection to a server.," on page 34 for further details.

5 Each successful `ktl_open()` call returns a KTL handle, which is a pointer to a (largely) opaque structure containing all information necessary to access that service. This handle is supplied to most other KTL routines.

6 Having connected to all services, applications which are going to use an event loop (not all applications do; for example `show` and `modify` do not) call `ktl_read()` with the `KTL_CONTINUOUS` flag. In this way an application can express interest in nominated events and

specify callbacks to be invoked when those events occur. The interpretation of `object_name` is dependent on `style`. For the keyword style it is a keyword name; for the message style it is a message type; for other styles it could be anything.

7 Note how `ktl_read()` specifies the callback address by filling in a `context` structure. The maintainance of context is an essential part of `KTL` and is discussed in Section 3.3, “Command context,” on page 6.

8 Refer to Appendix A, “Examples of `KTL` applications,” on page 15 for examples of the use of `ktl_read()` and to its detailed description in Appendix C.8, “ktl_read() --- Read a `KTL` keyword or message, continuously or once only.,” on page 38 for further details.

9 The application has now expressed interest in all its events and is ready to enter its event loop (applications without event loops would now simply make a sequence of `ktl_read()` and/or `ktl_write()` calls, and exit). At this point the application loops on a blocking call to `select()`. Each `select()` call blocks on pending input from a combination of `KTL` and non- `KTL` file descriptors. `KTL` events are handled by calling `ktl_dispatch()`.

10 Refer to Appendix A, “Examples of `KTL` applications,” on page 15 for examples of the use of `ktl_dispatch()` and to its detailed description in Appendix C.2, “ktl_dispatch() --- Dispatch a `KTL` event for a specified service.,” on page 30 for further details.

11 Once the event loop (if any) exits (few do), `ktl_close()` (Appendix C.1 on page 30) should be called to close the connections to the servers and to free the storage associated with the `KTL` handles.

3.2 Callbacks

All but the most trivial `KTL` applications do all of their work in callback routines. For those not used to writing event-driven applications, this may seem an unnatural style and it may seem that excessive constraints are being placed on the programmer. Such feelings should diminish with experience.

We have already seen how `ktl_read()` is called with the `KTL_CONTINUOUS` flag in order to associate specified events with callbacks. What happens when such an event occurs?

1 In the event loop, the `select()` call returns and `ktl_dispatch()` gets called with the handle corresponding to the server which originated the event.

2 `ktl_dispatch()` calls a service/style-specific `event()` routine to return details of the event in a generic form which is understood by `KTL`.

3 It then checks that the event is one of those in which the application is interested.

4 It creates a *command context* structure. Command context is discussed in Section 3.3, “Command context,” on page 6.

5 It invokes the callback.

6 It tidies up after the callback (Section 3.4 on page 7).

`KTL` callbacks have a standard calling sequence:

Function value : int: 1 if all `OK`, -1 otherwise. The function value is used for two separate purposes. Firstly, if a negative (error) status is returned, `KTL` always regards the command as being complete and will delete the command context. Secondly, the callback function value is copied to the status field of the response to the initiator of the command (if any such response is sent).

Invocation :

int callback (object_name, client_data, call_data, context)

Parameters :

> `KTL_OBJECT_NAME object_name;`

The name of the object which has caused this event. The exact meaning of this is style-dependent. For the keyword style, it is the name of the keyword whose value has changed; for the message style it is the type of the message which resulted in the event.

> `void * client_data;`

The same arbitrary piece of data which was supplied as the `client_data` parameter to the `ktl_read()` or `ktl_write()` call which has resulted (directly or indirectly) in this callback being invoked.

> `void * call_data;`

The value of the object which has caused this event. The exact meaning of this is style-dependent. For the keyword style, it is a pointer to a union of type `KTL_POLYMORPH` which contains the keyword value. For the message style, it is a pointer to the message which resulted in the event.

! `KTL_CONTEXT context;`

A pointer to the command context structure. This is a public structure and the callback author must obey various conventions in its use. The contents of the structure and the conventions which must be obeyed are described in Section 3.3, "Command context," on page 6.

Refer to Appendix A.1, "`show` and `modify` --- show and modify keyword values," on page 15, Appendix A.2, "`xshow` --- continuously display keyword values," on page 16 and Appendix A.3, "`go` --- modify multiple keywords in parallel," on page 19 for examples of keyword-style callbacks. Refer to Appendix A.4, "`watch_dcs` --- instrument telescope interface," on page 21 for examples of message-style callbacks.

3.3 Command context

Event-driven applications frequently need to initiate interactions with other applications. One way to do this is for a callback to call `ktl_read()` or `ktl_write()` to initiate a keyword read or write (or send a message to somebody) and then return. At some later time, a response will arrive and will result in invocation of another callback. This is fine provided that the first callback is not interested in monitoring the progress of the keyword read or write. But what if it wanted to write, say, ten keywords in parallel and, when the tenth write completes, do something completely different? What if it wants to time out if no response is received within ten seconds?

`KTL` supports the writing of callbacks which manage complex interactions such as the ones mentioned above. See Section 6.0, "More on command context," on page 10 for further details.

3.4 Tidying up after callback invocation

Application programmers do not normally have explicitly to allocate or free `KTL` command contexts (or sub-contexts, which have not yet been mentioned). This is all done by `KTL` behind the scenes. Once control returns from a callback, the command context will be deleted if the callback returned a negative function value, or if `context->state` is `KTL_INITIAL_STATE` or `KTL_FINAL_STATE` (the rule about `KTL_INITIAL_STATE` means that simple-minded callbacks which don't care about command context can just ignore it and it will always be deleted when they return).

Another thing that `KTL` does automatically is the sending of responses to the originators of commands. Obviously this only makes sense for external commands (*ie*, commands initiated by messages from this or another task). This is handled by the service/style-dependent `respond()` routine. Refer to Appendix D.2, "`KTL_CONTEXT` --- structure created when command event occurs.," on page 44 for details of how to control these responses, to Appendix A.4, "`watch_dcs` --- instrument telescope interface," on page 21 for an example, and to Appendix 8.0, "Writing a new `KTL` service," on page 12 for a definition of the interface between `KTL` and service/style-dependent routines.

3.5 `KTL` kernel routines

The `KTL` kernel routines provide low-level facilities which are used not only within `KTL` but also within service/style-specific code. They are linked into a separate shareable library from the rest of the `KTL` routines.

`ktl_log`() should be used for logging all messages and errors. Whenever a routine detects an error, it should call `ktl_log`() to log the error, call `ktl_set_errtxt`() to set the corresponding error text, and return a negative (error) status. Whenever a routine receives an error status from a `KTL` call, it can call `ktl_get_errtxt`() to get the corresponding error text and `ktl_log`() to report it to the user.

ktl_log(): Log a status message.
`ktl_log`() should be used for logging all messages and errors. Eventually it will probably send all such traffic to a central logging task.

ktl_log_function(): Select current logging function.
`ktl_log_function`() specifies the function which is to be called with a single character string argument every time a message needs to be logged. Specify `ktl_log_default`() or `NULL` to revert to the default.

ktl_log_stream(): Select current logging stream.
`ktl_log_stream`() sets the stream (`FILE *`) to use for logging. This is used only if the default logging function is used. It will write to the specified stream (`NULL` uses the default of logging to standard error).

ktl_log_default(): Default log message output function.
`ktl_log_default()` is the default logging function. If `ktl_log_stream`() has been called, it writes the string to the stream that it specified. Otherwise it writes to standard error.

ktl_set_errtxt(): Save the current `KTL` error text in a static variable.

`ktl_set_errtxt`() should be called to set the error text whenever a routine returns a failure (negative) status. A temporary buffer is used for formatting the error text and a call to `ktl_get_errtxt`() can appear among `ktl_set_errtxt`()'s parameters.

ktl_get_errtxt(): Return a pointer to the current `KTL` error text.
`ktl_get_errtxt`() returns the error text previously set using `ktl_set_errtxt`(). It can be used in the argument list to `ktl_log`().

ktl_seq_num(): Obtain a new positive sequence number.
`ktl_seq_num`() should always be used for generating sequence numbers in messages sent by `ktl_read`() and `ktl_write`() since such sequence numbers are used for matching responses to commands.

3.6 `KTL` high-level routines

The `KTL` high-level routines define the application programmer's interface to `KTL`. In general, application programmers should call only these and the kernel routines.

ktl_open(): Open a connection to a server.
`ktl_open`() is normally the first `KTL` routine which is called by an application. It returns a handle which is used in all subsequent calls to high-level `KTL` routines.

ktl_open1(): Create and initialize a `KTL` handle.
`ktl_open1`() does the first part of `ktl_open`()'s job.

ktl_open2(): Open shareable library and find functions.
`ktl_open2`() does the second part of `ktl_open`()'s job.

ktl_open3(): Invoke service/style-dependent open functions.
`ktl_open3`() does the third and last part of `ktl_open`()'s job.

ktl_read(): Read a `KTL` keyword or message, continuously or once only.
`ktl_read`() occupies two slightly different roles. Firstly it handles the enabling and disabling of continuous reads of `KTL` objects (with a callback being invoked whenever a value becomes available). Secondly it handles one-shot reads. In the latter case, the caller can block awaiting the response, can perform other processing and then later block awaiting the response, or can ask that a callback be invoked when the response is received.

ktl_write(): Write a `KTL` keyword or message.
`ktl_write`() is in many respects complementary to `ktl_read`(). However, there is no concept of "continuous writes" and `ktl_write`() does not support the `KTL_CONTINUOUS` flag. For the keyword style (and any future styles where there is an obvious distinction between reading and writing an object), `ktl_read`() is used to read the value of an object and `ktl_write`() is used to write the value of an object. For the message style, there is no distinction between a message that reads values and a message that writes values (the difference is in the semantics of the messages and `KTL` does not concern itself with this); hence for the message style `ktl_write`() is used to initiate all transactions and `ktl_read`() is used only with `KTL_CONTINUOUS` and `KTL_WAITFOR`.

ktl_close(): Close a connection to a server.
`ktl_close`() closes a connection to a server. Ideally, every successful `ktl_open`() call

should have a corresponding `ktl_close`() call. In practice, `ktl_close`() calls are often omitted!

ktl_dispatch(): Dispatch a `KTL` event for a specified service.
`ktl_dispatch`() is called whenever an event may have occurred on a handle. It is `NOT` an error for it to be called when there are in fact no events to be dispatched on the handle. It can often be convenient to call `ktl_dispatch`() for all handles whenever an event occurs on any handle.

ktl_ioctl(): Control or return information about a service and its objects.
`ktl_ioctl`() is used in very much the same way as the Unix `ioctl`() system call. It is possible to add functionality to the `KTL` interface in an upwards-compatible way without adding new routines to the interface. It can also be used to send information to and receive information from service/style-specific code.

ktl_register(): Register service/style-dependent functions.
`ktl_register`() can be used to avoid the use of shareable libraries for service/style-specific code. Rather than using `ktl_open`() (which itself calls `ktl_open1`(), `ktl_open2`() and `ktl_\-open3`()), `ktl_open1`() can be called to create the `KTL` handle, `ktl_registe`() can be called to register the service/style-specific functions, and `ktl_open3`() can be used to open a connection to the server.

4.0 Running `KTL` applications

`KTL` applications need to access shareable libraries and it is necessary to define some environment variables before running them. There are two sorts of shareable libraries involved here:

1 Shareable libraries containing `KTL` itself. There are two of these:

libktl.so.0.0 contains all the high-level `KTL` routines.

libktlker.so.0.0 contains the `KTL` kernel routines.

Normally they will be found at run-time in the same place that they were found at link time. If this is not the case, the directory where they live will have to be added to the `LD_LIBRARY_PATH` environment variable.

2 Shareable libraries containing service and style implementations. These are not accessed at all at link time (new ones can be written without re-linking the `KTL` applications which will invoke them) and are expected to exist in the directory pointed to by `KTL_LIBRARY_PATH` (currently not actually a path but a single directory). If this environment variable is not defined, such libraries must exist in the current directory. Refer to Appendix C.6, "ktl_open2() --- Open shareable library and find functions.," on page 36 for a more detailed description of how these shareable libraries are located.

The other case where it is necessary to define environment variables is where the library name is not directly derivable from the service and style names. For example, the existing `CARA123` message style implementation uses the service name to decide which host and port to connect to. The `dcstst` service connects to the port corresponding to the `dcstst` service in the `/etc/services` file but activates the `libcara123_message.so.0.0` shareable library; hence the need for an environment variable DCSTST_MESSAGE with value cara123_message. Again, refer to Appendix C.6, “ktl_open2() --- Open shareable library and find functions.,” on page 36 for more details.

By convention, all these shareable libraries will be found in a directory whose name is of the form $KROOT/$RELNUM/lib. $KROOT is an environment variable whose value is the root of the tree in which all the Keck telescope and instrument software is to be found (here in Waimea, it translates to kroot). $RELNUM is an environment variable whose value is the KTL release number. If it is undefined or has a blank value, the latest (and possibly less well-tested) release will be used. Note that these are just conventions, and recent ones at that, and may not be adhered to at all sites.

5.0 Building KTL applications

To build a KTL application it is necessary only to know where to find the KTL and KTL kernel shareable libraries. Assuming that the conventions described above are being obeyed, an application called fred could be built thus:

```
% cc -o fred fred.o -L$KROOT/$RELNUM/lib -lktl -lktlker
```

Note how the application is *not* linked against any service/style-specific code. The only reason to do so is for ease of debugging (dbx and dbxtool can access source code in shareable libraries if they were mentioned to the linker). To link against the hires keyword shareable library, for example, you could do:

```
% cc -o fred fred.o -L$KROOT/$RELNUM/lib -lktl -lktlker -lhires_keyword
```

6.0 More on command context

A command context structure is created when a *command event* occurs (a command event is an event in which an application has expressed interest using ktl_read() with the KTL_CONTINUOUS flag). This command context is retained possibly over many subsequent callback invocations, each of which corresponds to the receipt of an associated *response event* (don’t worry for now how the responses get associated with the right command contexts). The callback uses the command context both to tell it what to do and to tell KTL what to do. Consider a simple example of a callback which wants to read the value of an integer keyword called fred but wants to do it without blocking:

```
int
callback( keyword, handle, value, context )
char              *keyword;    /* “object name” is keyword name    */
KTL_HANDLE        *handle;     /* “client data” is server handle   */
KTL_POLYMORPH     *value;      /* “call data” is keyword value     */
KTL_CONTEXT       *context;    /* “context” is command context     */
{
switch( context->local_state) {
      case 0:
       if ( ktl_read( handle, KTL_NOTIFY, “fred”, NULL, NULL, context ) < 0 )
            return -1;
       context->local_state   = 1;
       context->state         = KTL_MESSAGE_STATE;
       break;
      case 1:
       if ( context->status < 0 )
```

```
            return context->status;
        printf( "Value of %s is %d\n", keyword, value.i );
        context->state          = KTL_FINAL_STATE;
        break;
       }
       return 1;
}
```

1 `context->local_state` is always `0` the first time the callback is invoked (*ie*, when the command event has just occurred). It is used by the callback for its own private purposes (note that it is better to use a component of the context rather than a static variable since there is nothing to stop there being multiple contexts which are simultaneously using the same callback routine).

2 `context->state` is always `KTL_INITIAL_STATE` on entry and is altered by the callback in order to tell `KTL` what to do. Setting it to `KTL_MESSAGE_STATE` tells `KTL` that this command is expecting a message (which will, if we are lucky, contain the value of the `fred` keyword).

3 When the response arrives, *the same callback is again invoked*. This time `context->local_-state` is `1` and `fred`'s value is in `value`. Setting `context->state` to `KTL_FINAL_STATE` tells `KTL` that the command is complete and that it can delete the command context.

4 Just to stay honest: this example was chosen because it is simple and illustrates the usage of all four of the callback parameters, but no current `KTL` keyword style implementations support `ktl_read()` with `KTL_NOTIFY`, so it doesn't actually work!

Refer to Appendix D.2, "`KTL_CONTEXT` --- structure created when command event occurs.," on page 44 for a full description of all command context components and their usage.

7.0 Existing `KTL` services and styles

At the time of writing, four `KTL` services have been implemented in the keyword style and three in the message style. All the implementations of a given style share *glue* code which connects the generic `KTL` interface (Section 8.0 on page 12) to service-specific code. The following paragraphs give a little more information on the existing implementations.

7.1 Keyword style implementations

Keyword style implementations exist for `HIRES`, `LRIS`, `LWS` and `DCS`. All of these use an earlier set of interfaces known as the `KICS` interfaces [1] (`KICS` stands for `K`eck `I`nstrument `C`ontrol `S`ystem). The `LWS` implementation uses Sun `RPC` as its transport mechanism; the other three use the Lick MUSIC system and all use the `FIORD` (`FITS` `I`nput/`O`utput `R`outine `D`atabase) mechanisms [2].

All the keyword implementations use what is referred to as " `KTL` glue" to connect the generic `KTL` interface (Section 8.0 on page 12) to the `KICS` interfaces. This glue contains keyword-style implementations of the service/style-specific `open()`, `ioctl()`, `read()`, `write()`, `event()`, `respond()` and `close()` routines.

7.2 Message style implementations

Message style implementations exist for `CARA123` [4], `MUSIC`, and `DCSSIM`. The latter is a simple simulation of the `DCS`.

All the message implementations define glue similar to that used for the keyword implementations. This glue assumes the existence of service-specific `open()`, `ioctl()`, `read()`, `write()` and `close()` routines, all of which have standard calling sequences. All services share a generic message header structure.

The `watch_dcs` application (Appendix A.4 on page 21) uses the `MUSIC` message style implementation to talk to `MUSIC` and either the `CARA123` or `DCSSIM` message style implementation to talk to the `DCS`.

8.0 Writing a new `KTL` service

Generic `KTL` code does not know anything about services or styles. It defines a standard interface to service/style-dependent code, which is summarized below (obviously this section will have to be expanded in a later version of this document):

int open (char *service, char *style, int flags, void **handle)
: opens a connection to a server. `service`, `style` and `flags` are the corresponding parameters from `ktl_open()` (Appendix C.4 on page 34). `handle` is the address of the `KTL` handle's specific pointer and can be used by the service in whatever way it wants. Typically it will allocate its own handle structure and store its address in the `KTL` handle's `specific` pointer.

int ioctl (void *handle, int flags, ...)
: controls and returns information about a service. All parameters are exactly the same as for `ktl_ioctl()` (Appendix C.3 on page 31) except that the handle is the `KTL` handle's `specific` pointer. Note that three of the flag values return information about this service implementation. These are `KTL_IMPLCAPS`, `KTL_NAMEDESCR` and `KTL_DATADESCR`. The information returned by these three `ioctl`s is all that `KTL` knows about this service implementation.

int read (void *handle, int flags, caddr_t object_name, void *call_data)
: reads the value of an object. The parameters are similar to those of `ktl_read()` (Appendix C.8 on page 38), but there are no `client_data` or `context` parameters, these being meaningful only at higher levels.

int write (void *handle, int flags, caddr_t object_name, void *call_data)
: writes the value of an object. The parameters are similar to those of `ktl_write()` (Appendix C.10 on page 42), but there are no `client_data` or `context` parameters, these being meaningful only at higher levels.

int close (void *handle)
: closes a connection to a server. Again, the parameters are similar to those of `ktl_close()` (Appendix C.1 on page 30).

int event (void *handle, int *nevent, KTL_EVENT_DESCR **pdescr)
: checks for pending events and returns the number of events and an array of event descrip-

tors. If no events are pending, the routine must set `nevent` to zero and return; it must not block! `KTL_EVENT_DESCR` is defined in `ktl.h` and is a service/style-independent way of describing an event.

int respond (void *handle, KTL_CONTEXT *context)
decides whether to respond to the initiator of a command and, if required, sends the response. For all the current keyword style implementations this routine does nothing, but for the message style it is fully implemented.

As already mentioned, all existing implementations of a given style share glue code that implements the above routines and calls lower-level service-specific routines (which have a service-independent interface). This maximizes code sharing. It is beyond the scope of this document to go any further.

9.0 Writing a new `KTL` style

Writing a new style will not be common. Indeed, we will probably never write a new style for use at Keck. Perhaps outside organizations will do so though. The following are some ideas for new styles:

- like the keyword style but keyword values include more information about the keyword, such as its type, its units, how to output it....
- like the message style but using a self-describing data system for message bodies.

10.0 Portability to other systems

`KTL` has been implemented undser SunOS and has not been tested under any other operating systems. It assumes the existence of the `select()` system call and it assumes that a mechanism exists for run-time activation of shareable libraries, looking up of symbol names to give routine addresses, and invocation of those routines.

Many operating systems have such a shareable library capability, including SunOS, `HP/UX`, `VMS` and `VxWorks`. Access to the capability is non-portable, as is the naming of such shareable libraries. `KTL` restricts such knowledge to three routines (not documented here) `ktl_load_open()`, `ktl_load_find()` and `ktl_load_close()`, which will have to be re-implemented for each system. This is far and away the biggest portability issue. Note that `ktl_register` (Appendix C.9 on page 41) can be used to avoid the need for shareable libraries, but many of the advantages of `KTL` are lost in so-doing.

Any other portability issues are minor.

11.0 Changes to this document

Version 4 was the original written by William Lupton (`wlupton@keck.hawaii.edu`). It refers to `KTL` release 4.

12.0 References

1 Al Conrad. CSET/CREAD. HIRES/LRIS CDR Document number 2.2.A. Lick Observatory, March 1991.

2 Al Conrad. FIORD. HIRES/LRIS CDR Document number 2.2, Lick Observatory, January 1991.

3 Al Conrad and William Lupton. KTL: the Keck Task Library. Keck Software Document KSD 8, W.M. Keck Observatory, December 1991.

4 Hilton Lewis. DCS Software Interface Specification. California Association for Research in Astronomy Document CARA 123, W.M. Keck Observatory, January 1991.

Appendix A Examples of KTL applications

In this section, several real KTL applications are introduced and explained. Many of these applications were written by Al Conrad, but I take full responsibility for any errors.

A.1 show and modify --- show and modify keyword values

The show and modify commands are amongst the simplest possible KTL applications: they use only one service, use the keyword style, and do not handle any asynchronous events. We will take modify as an example (not all code is shown):

```
#include "ktl.h"
#include "kval.h"
```

1 KTL applications should include ktl.h, which contains all external KTL declarations. So also should service/style-specific code which wants to call ktl_log() or ktl_set_errtxt().

2 kval.h is not part of KTL---it is shared between show, modify and several related applications.

```
char service[256];       /* string buffer to receive serv name     */
KTL_HANDLE* khand;       /* for connection to KTL                  */
KTL_POLYMORPH data;      /* current keyword's value                */
u_int sync = KTL_WAIT;   /* KTL_WAIT or KTL_NOWAIT                 */
```

3 service will receive the service name, which is specified as a command-line argument. This might be hires for modifying HIRES keywords, or dcs for modifying DCS (telescope drive and control system) keywords.

4 khand is the KTL handle (Appendix C.5 on page 36). This is returned by ktl_open() and is used in most subsequent KTL calls. Compare it with the FILE * returned by fopen() and used by fread(), fwrite() and fclose().

5 data is in fact normally just a scalar value (KTL_POLYMORPH is in fact a union with int, float, double, int * etc members). Keyword values are always passed to and from KTL routines (and callbacks) as pointers to KTL_POLYMORPH.

6 sync is a synchronization flag which will be passed to ktl_write(), indicating whether ktl_write() is to block waiting for completion of the keyword write operation. modify uses either KTL_WAIT or KTL_NOWAIT, but KTL supports other values (Appendix C.8 on page 38, Appendix C.10 on page 42); for example KTL_NOTIFY causes completion to be notified via callback invocation.

```
if ( ( ktl_open( service, "keyword", 0, &khand ) ) < 0 ) {
      fprintf( stderr, "Couldn't connect to %s: %s\n", service,
                                                        ktl_get_errtxt() );
      exit( -1 );
}
```

7 ktl_open() (Appendix C.4 on page 34) connects to the specified service for use of the specified style. What really happens is that the service and style names are concatenated to give a string such as hires_keyword, this is used to generate the name of a shareable library which must exist in the path defined by the KTL_LIBRARY_PATH environment variable, the shareable library is opened, and a service/style-specific open() routine (in the shareable library) is invoked in order to connect to the service.

8 If a `KTL` routine returns a negative (failure) status, `ktl_get_errtxt()` can always be used to return error text in a form suitable for presenting to the user. More detailed information is written to a log file.

```
for ( i = 0; kvals[i] != NULL; i++ ) {
      if    ( ktl_ioctl( khand, KTL_FORMAT | KTL_ASCIN | KTL_BINOUT,
                  kvals[i]->keyword, kvals[i]->value, &data ) < 0 ) {
            fprintf( stderr, "%s\n", ktl_get_errtxt() );
            exit( -1 );
      }
      if    ( ktl_ioctl( khand, KTL_FORMAT | KTL_ASCIN | KTL_ASCOUT,
                  kvals[i]->keyword, kvals[i]->value, &form ) < 0 ) {
            fprintf( stderr, "%s\n", ktl_get_errtxt() );
            exit( -1 );
      }
      if    ( ! silent ) printf( "setting %s = %s (%s)\n",
                  kvals[i]->keyword, form, sync ==KTL_WAIT? "wait":"no wait");
      if    ( ktl_write( khand, sync, kvals[i]->keyword,
                  NULL, &data, NULL ) < 0 ) {
            fprintf( stderr, "error setting %s\n", kvals[i]->keyword );
            fprintf( stderr, "%s\n", ktl_get_errtxt() );
            exit( -1 );
      }
}
```

9 The call to the routine which parsed the command line and built the `kvals` structure is not shown.

10 The first `ktl_ioctl()` (Appendix C.3 on page 31) call takes the value typed by the user and converts it to binary (if the keyword value is a string then the “binary” value is still a string!). This is not necessarily just a question of calling `sprintf()` since there may be unit conversions or conversions from hexagesimal format.

11 The second `ktl_ioctl()` call takes the value typed by the user and formats it as an `ASCII` string with any available extra information (such as units) appended. This is done just so that the value can be presented to the user.

12 `ktl_write()` (Appendix C.10 on page 42) sets the value of the keyword. `sync` is either KTL_WAIT, in which case the routine blocks until the keyword has been set (which may require round-trip messages), or `KTL_NOWAIT`, in which case the routine returns immediately and any responses will be ignored.

```
ktl_close( khand );
```

13 `ktl_close()` (Appendix C.1 on page 30) closes the connection that was opened by `ktl_open()`, and frees the `KTL` handle and any other memory allocated by `ktl_open()`.

A.2 `xshow` --- continuously display keyword values

`show` supports only a single service and simply reads and reports the values of the specified keywords. `xshow` is more complicated in that it supports multiple services and, rather than just reading keyword values once and then exiting, it expresses interest in the specified keywords, and is notified when any of the keywords change value. It maintains an `X` display with a list of keywords and their current values.

The following code fragments illustrate the ways in which `xshow` differs from `show` and `modify`:

```
for ( i = 0; i < nservices; i++ ) {
```

```
      if    ( ktl_open( services[i].name, "keyword", 0,
                                  &services[i].handle ) < 0 ) {
            fprintf( stderr, "Couldn't open KTL service %s: %s\n",
                                  services[i].name, ktl_get_errtxt() );
            exit( -1 );
      }
}
```

1 This is pretty similar to `modify`. The `services` structure was created when the command line was parsed (not shown) and `ktl_open()` fills in its `handle` components.

```
KTL_CONTEXT context;    /* command context (used only for callback)     */
KTL_HANDLE  *handle;    /* current KTL handle                           */

context.callback = keyword_value_callback;
for ( i = 0; i < nkeywords; i++ ) {
      handle = keywords[i].service->handle;
      if    ( ktl_ioctl( handle, KTL_TYPE|KTL_BINOUT, keywords[i].keyword,
                                  &keywords[i].datatype, &size ) < 0 ) {
            fprintf( stderr, "%s\n", ktl_get_errtxt() );
            exit( -1 );
      }
      if ( ktl_read( handle, KTL_CONTINUOUS, keywords[i].keyword,
                                  &keywords[i], NULL, &context ) < 0 ) {
            fprintf( stderr, "%s\n", ktl_get_errtxt() );
            exit( -1 );
      }
}
```

2 `context` is a command context (Appendix D.2 on page 44) structure. It is needed because it is a parameter to `ktl_read()` (Appendix C.8 on page 38), which is being called to express interest in the keywords which were specified on the command line. When used for this purpose, the callback address is the only context component which needs to be defined and it is acceptable to declare an instance of a command context rather than use `ktl_context_create()` (Appendix D.4 on page 47) to create one.

3 `ktl_ioctl()` is being used here to to look up all the keyword types. This information will be used later in the callback routine. `KTL_BINOUT` returns the type as a binary (enumerated) value---`KTL_ASCOUT` would have returned it as a character string.

4 `ktl_read()` is called with the `KTL_CONTINUOUS` flag. This arranges for `keyword_value_callback()` to be called whenever the value of the keyword changes (it will be called once immediately so that no special code is required in order to obtain the initial value of the keyword). `&keywords[i]` is passed as one of the arguments to the callback and allows it directly to determine what to do (as we will see). The `NULL` parameter is for the keyword value (`NULL` because the value is available only when the callback is invoked, not when `ktl_read()` returns).

```
maxfds = getdtablesize();
for(;;) {

      [X event stuff (not shown)]...

      FD_ZERO( &readfds );
      FD_SET( ConnectionNumber( XtDisplay( toplevel ) ), &readfds );
      for   ( i = 0; i < nservices; i++ ) {
            (void) ktl_ioctl( services[i].handle, KTL_FDSET, &tempfds );
            for   ( j = 0; j < maxfds; j++ )
                  if    ( FD_ISSET( j, &tempfds ) )
                        FD_SET( j, &readfds );
      }
```

```
    if ( select( maxfds, &readfds, NULL, NULL, NULL ) == -1 ) {
          fprintf (stderr, "%s, %d: Unexpected return from select: %d\n",
                            __FILE__, __LINE__, errno );
          exit( -1 );
    }

    for   (i = 0; i < nservices; i++ )
          if    ( ktl_dispatch( services[i].handle ) < 0 )
                fprintf( stderr, "%s event dispatch failure: %s\n",
                                  services[i].name, ktl_get_errtxt() ); }
```

5 `xshow` now has to enter an infinite loop in which it services `X` and `KTL` events. This is done by first servicing any outstanding `X` events and then using the `select()` system call to block until an event arrives on the `X` or any of the `KTL` file descriptors.

6 A new `fd_set` is built on every iteration through the loop. This is probably unnecessary but is done because, in the general case, the set of active fds can change (for example an `accept()` call may complete and the application may start reading a new socket). `ktl_ioctl()` is used with the `KTL_FDSET` flag to return each service's `fd_set`, all of which are combined to make a single `fd_set` which can be used by `select()`.

7 Once an event occurs on any of the file descriptors, `select()` returns and `ktl_dispatch()` (Appendix C.2 on page 30) is called for each service. One could use the modified `fd_set` to determine which services require service and then only call `ktl_dispatch` for those services. However, some service implementations can queue events which they are not yet ready to service and the existence of queued events does not cause `select()` to exit; hence the pragmatic calling of `ktl_dispatch()` for all services on each event.

8 `ktl_dispatch()` calls the service/style-specific `event()` routine to return an array of event descriptors which it processes in order to call the callback associated with each event.

```
int
keyword_value_callback( keyword, keydesc, data, context )
char *keyword;                      /* changed keyword name (unused)    */
KEYWORD_DESC *keydesc;              /* changed keyword descriptor       */
KTL_POLYMORPH *data;                /* changed keyword value            */
KTL_CONTEXT *context;               /* command context (unused)         */
{
      switch( keydesc->datatype ) {
            case _int:
                  sprintf( buff, "%12d", client_data->i );
                  break;
            case _float:
                  sprintf( buff, "%12.3f", client_data->f );
                  break;
            [other types (not shown)]...
      };

      [update keydesc->value_w widget label string (not shown)]...

      return 1;
}
```

9 The body of the callback (Section 3.2 on page 5) is very simple: the new value is formatted as a string (actually `ktl_ioctl()` / `KTL_FORMAT` might be a better way to do this) and then the keyword's value widget label string is set to the value string.

A.3 `go` --- modify multiple keywords in parallel

It may seem as though `go` does the same as `modify`, but it differs in that it first initiates all keyword changes and then uses callbacks to monitor progress. It also has an option to update progress on an `X` display (`go -x`).

The following code fragments illustrate the ways in which `go` differs from `modify` and `xshow`:

```
if ( ktl_context_create( khand, go_callback, NULL, NULL, &context ) < 0 ) {
      fprintf( stderr, "Couldn't create context: %s\n", ktl_get_errtxt() );
      exit( -1 );
}
```

1 After `go` has parsed its command line and called `ktl_open()` (which returns `khand`), it calls `ktl_context_create()` (Appendix D.4 on page 47) to create a command context. The arguments are the `KTL` handle, the callback associated with the context (`go_callback`), client data to pass as a callback parameter (here `NULL` but can be anything), a pointer to the message that resulted in context creation (here `NULL` because the context is not being created as a result of an incoming message), and the address of a variable to receive a pointer to the newly-allocated context structure.

2 When using `KTL` to write server tasks, command contexts are created automatically when command messages are received, and deleted automatically when commands complete. However, `go` is a user interface and does not receive command messages (it does receive response messages), and the explicit creation of the context is a way of simulating the arrival of an external message. A server task would save the incoming command message for the duration of the command, partly to allow it to respond to the client and partly for the use of the callback routine. Here there is no incoming command message so the corresponding component of the command context contains `NULL`.

```
if ( go_callback( NULL, &desc, NULL, context ) < 0 ) {
      fprintf( stderr, "Couldn't start go command: %s\n",
                                        ktl_get_errtxt() );
      exit( -1 );
}
```

3 Having created the context, the callback routine is called explicitly to get the ball rolling. This is done from the main program if not using an `X` display, and when the `go` button is pressed otherwise (in the go button's activate callback).

4 The above works fine in this case, but it would be better not to call the callback routine explicitly, since the application programmer does not, and need not, know the rules about what to do before and after calling it---normally only `ktl_dispatch()` (Appendix C.2 on page 30) calls callbacks. A better plan is to use the `KTL` timer routines, whose use is illustrated in Appendix A.4, "`watch_dcs` --- instrument telescope interface," on page 21.

```
switch ( context->local_state ) {
      case GO_STATE_IDLE:
            khand = context->event_handle;
            godesc = (GO_DESCRIPTOR *) client_data;
            kvals = godesc->kvals;
```

5 This callback is more complicated than `xshow`'s because it needs to be invoked multiple times in order to manage the various asynchronous messages which will arrive as the various keywords get set. These messages can arrive in any order.

6 `context->local_state` is always guaranteed to be `0` the first time that a callback is invoked for a given command. The programmer can thus use it to drive the callback as a state machine. In `go_callback` there are only two states: `GO_STATE_IDLE`, which is the initial state, and `GO_STATE_ACTIVE`, which is the state when at least one keyword write is active.

7 `khand` is the `KTL` handle, as in the other examples. The handle on which the current event has occurred can always obtained from `context`'s `event_handle` component. This is not necessarily the same as the handle on which the event arrived which resulted in the context being created in the first place (this is in `create_handle`).

8 `godesc` and `kvals` are pointers derived from the client data parameter. `godesc` is a descriptor for the entire `go` program, and `kvals` is an array of keyword names and desired values.

```
for ( i = 0; kvals[i].keyword != NULL; i++ ) {
      if    ( ktl_ioctl( khand, KTL_FORMAT | KTL_ASCIN | KTL_BINOUT,
                  kvals[i].keyword, kvals[i].value, &data ) < 0 ) {
            fprintf( stderr, "%s\n", ktl_get_errtxt() );
            exit( -1 );
      }
      if    ( ktl_ioctl( khand, KTL_FORMAT | KTL_ASCIN | KTL_ASCOUT,
                  kvals[i].keyword, kvals[i].value, &form ) < 0 ) {
            fprintf( stderr, "%s\n", ktl_get_errtxt() );
            exit( -1 );
      }
      printf( "setting %s = %s\n", kvals[i].keyword, form );
      if    ( ktl_write( khand, KTL_NOTIFY, kvals[i].keyword,
                        &(godesc->keystat[i]), &data, context ) < 0 ) {
            fprintf( stderr, "%s\n", ktl_get_errtxt() );
            exit( -1 );
      }
}
```

9 `ktl_ioctl()` is used to convert the user-supplied keyword values to binary and nicely-formatted `ASCII` in exactly the same way as was done by `modify`.

10 `ktl_write()` initiates the setting of each keyword to its desired value. This time the `KTL_NOTIFY` flag is used, which means that `ktl_write()` does not block waiting for completion but instead creates a sub-command context (Appendix D.3 on page 46) , to which it allocates a unique sequence number. This sequence number is present in any messages sent to other tasks in order to set the keyword, and is present in any replies returned by such tasks.

11 The other parameters to `ktl_write()` are the keyword name, the client data to use for callbacks resulting from this call to `ktl_write()` (here this is a pointer to a keyword-specific status structure), the value to which to set the keyword, and the command context.

```
context->local_state = GO_STATE_ACTIVE;
context->state = KTL_MESSAGE_STATE;
break;
```

12 Recall that `context->local_state` is for use by the programmer in structuring the callback. `context->state`, on the other hand, communicates the programmer's wishes to `KTL`. Setting it to `KTL_MESSAGE_STATE` indicates that the callback is next to be invoked when a message arrives which is associated with this command. `KTL` also provides timeout support and can re-invoke the callback if no message arrives within a specified interval, but this is not illustrated here.

```
case GO_STATE_ACTIVE:
      keydesc = (KEYWORD_STATUS_DESC *) client_data;
      switch( context->status ) {
            case 1:
```

```
            [report "commenced" (not shown)]...
            if ( xflag ) {
            break;
      case 2:
            [report "completed" (not shown)]...
            break;
      default:
            [report error (not shown)]...
            exit( -1 );
      }
```

13 This time the callback has been invoked because a response has been received from one of the server tasks to which messages were sent on the first callback invocation. When using `KTL_NOTIFY`, the callback has to be prepared to handle possibly several responses and can use `context`'s `done`, status and `subcontext` components to follow the progress of a given keyword (`done` indicates whether this is a final response, `status` contains an application-dependent status value from the message, and `subcontext` points to the associated sub-command context, if any).

14 Here the callback knows that a `status` value of `1` indicates an initial acknowledgment and that a value of `2` indicates completion. It uses the `status` value to update the display accordingly.

```
ktl_subcontext_count( context, &count );
if ( count == 0 )
      [report "all done" (not shown)]...
break;
```

15 `ktl_subcontext_count()` returns the number of active sub-command contexts: the number of keyword writes which have not yet completed. Alternatively the callback could have maintained its own records.

```
default:
      fprintf( stderr, "Unexpected local state: %d\n",
                    context->local_state );
      exit( -1 );
}
```

16 This is the bottom of the `switch` on `context->local_state`.

17 If this all seems rather complicated, recall that at `go` was at all times responsive to both external (message) and `X` events.

A.4 `watch_dcs` --- instrument ⟷ telescope interface

`watch_dcs` is our first example of a `KTL` application which uses the message style. When using the message style, the application sees actual messages rather than the abstraction to keywords (you might think that there is a 1--1 correspondence between setting a keyword and sending a message but this is not necessarily so; many keywords can be set using a single message, and conversely a single message can provide new values for many keywords).

`watch_dcs` is essentially a translation task. It runs on an instrument Sun, receives `MUSIC` messages from instrument tasks, and forwards them to the `DCS` (telescope control system) as messages conforming to the document known as `CARA123` [4], which defines the external message interface to the `DCS`, `ACS` (mirror system) and `PCS` (phasing camera). It then receives responses and forwards them to the originating `MUSIC` clients, plus provides timeout support. In addition to all that, it sends a 1Hz telescope status request to the `DCS` and forwards that as a `MUSIC` broadcast to all MUSIC clients which have expressed interest in receiving DCS status. This broadcast allows applications such as xshow to display telescope status.

```
if ( ktl_open( music_service, "message", KTL_USETIMER, &music_handle ) < 0)
      exit( -1 );
if ( ktl_open( cara123_service, "message", KTL_USETIMER, &cara123_handle) < 0)
      exit( -1 );
```

1 music_service defaults to dcs (not shown) and is the name by which watch_dcs wants to be known to MUSIC (other clients need to know this name so that they can talk to watch_dcs). The DCS_MESSAGE environment variable must be defined and have value music_message, which is used to find the shareable library libmusic_message.so.0.0. See Appendix C.6, "ktl_open2() --- Open shareable library and find functions.," on page 36 for more details.

2 cara123_service defaults to dcstst (yes, confusing names) and indicates the port to which to connect on the DCS host (dcstst exists in the /etc/services file; alternatively a valid number can be used as the service name). The DCSTST_MESSAGE environment variable must be defined and have value cara123_message, which is used to find the shareable library libcara123_message.so.0.0.

3 KTL_USETIMER is used as the flags parameter and declares that this service plans to the KTL timer routines to support timeouts and timed reschedules.

```
tcontext.callback = music_command_callback;
if ( ktl_read( music_handle, KTL_CONTINUOUS | KTL_NOPRIME,
                  DCS_STATUS_REQ_IP_MSG_ID, NULL, NULL, &tcontext ) < 0 )
      return -1;
[other similar calls (not shown)]...
```

4 These calls to ktl_read() are the message-style equivalent of the calls made in xshow. When using the message style, the object_name parameter is a message type rather than a keyword name (and for future styles it could be anything else). DCS_STATUS_REQ_IP_MSG_ID is the DCS command message type which watch_dcs is to handle (we plan to support a "match-all" message type but this has not yet been implemented).

5 The KTL_NOPRIME flag suppresses an attempt to read an initial value for a given message. This concept makes sense for keyword style (for example, xshow wants an initial value for each keyword which it is monitoring), but is meaningless for the message style.

```
if ( ktl_context_create( cara123_handle, periodic_callback, NULL, NULL,
                                                &context ) < 0 ) {
      fprintf( stderr, "Failed to create context: %s\n", ktl_get_errtxt() );
      exit( -1 );
}
if ( ktl_timer_create( cara123_handle->timer_handle,
                                                &context->timer_entry ) < 0 ){
      fprintf( stderr, "Failed to create timer: %s\n", ktl_get_errtxt() );
      exit( -1 );
}
if ( ktl_timer_arm( context->timer_entry, 0, context ) < 0 ) {
      fprintf( stderr, "Failed to arm timer: %s\n", ktl_get_errtxt() );
      exit( -1 );
}
```

6 Here, as in go, we create a command context explicitly, but this time to implement the 1Hz poll of DCS status. ktl_context_create() states that the callback is periodic_callback().

7 Having created the context, `ktl_timer_create()` creates a timer associated with the context and `ktl_timer_arm()` arms it with a delay of `0` milliseconds. The third parameter to `ktl_timer_arm()` is user data and has to be the address of the context (you couldn't be expected to know this!). The result is that the timer fires immediately, resulting in a `KTL` event and an invocation of `periodic_callback`. This is a better way of doing it than direct callback invocation, as practiced by `go`.

8 There really should be a utility routine to combine the effects of these three calls---application programmers should not have to be aware of such nitty-gritty details.

9 The main loop is similar to those already seen for `xshow` and `go` and is not shown.

```
int
music_command_callback( msg_type, client_data, message, context )
int msg_type;
void *client_data;
DCS_EXT_MSGS *message;
KTL_CONTEXT *context;
{
```

10 This callback is called whenever a `MUSIC` message arrives in which interest has been expressed. For the message style, the first callback parameter is always the message type rather than the keyword name, and the third parameter is always the message which has resulted in the callback invocation.

```
switch( context->local_state ) {
      case 0:
            if ( ktl_write( cara123_handle, KTL_NOTIFY, msg_type, client_data,
                                                  message, context ) < 0 )
                  return -1;
            context->flags          = KTL_INHIBIT_REPLY;
            context->timeout        = 10000;
            context->state          = KTL_MESSAGE_STATE;
            context->local_state    = 1; break;
```

11 `context->local_state` is `0` on the first invocation. `ktl_write()` forwards the message to the `DCS`. Note that both the `MUSIC` amd `CARA123` `KTL` message-style implementations use the same message structures, so no translation is necessary (the `CARA123` implementation worries about host-to-network conversions and in fact uses `XDR` to achieve this).

12 \item Having forwarded the message, the callback does not need invoking again until either a response is received from the `DCS` or else a timeout occurs. This is achieved by setting `context->state` to `KTL_MESSAGE_STATE`, indicating that a message is expected, and `context->timeout` to `10000`, indicating that the message must arrive within 10 seconds.

13 `context->flags` are set to `KTL_INHIBIT_REPLY` to prevent a response from immediately being sent back to the `MUSIC` client (the default is to send replies automatically on the first and last callback invocations---see Appendix C.2, “ktl_dispatch() --- Dispatch a `KTL` event for a specified service.,” on page 30). This is done because the `MUSIC` clients of `watch_dcs` are written to expect only the replies that actually return from the `DCS`.

14 `context->local_state` is set to `1` so that the reply-handling code will be executed on the next callback invocation (compare `go`).

```
case 1:
      status = message->DCS_EXT_MSGS_u.msg_hdr.status;
      flags = message->DCS_EXT_MSGS_u.msg_hdr.flags;
      if    ( status & KTL_MESSAGE_STATUS_ERROR || !( status & 1 ) ) {
```

```
            status |     = KTL_MESSAGE_STATUS_ERROR;
            retval       = status;
      }
      context->response_msg   = message;
      context->flags          = KTL_INHIBIT_FREE;
      if    ( status & KTL_MESSAGE_STATUS_FINAL || flags & COM_M_FINAL )
            context->state    = KTL_FINAL_STATE;
      break;
}
return retval;
```

15 When a reply arrives from the DCS, or a timeout occurs, the callback will be invoked. The processing here is somewhat application-specific---suffice to say that setting `context->response_msg` to `message` tells KTL to forward `message` back to the MUSIC client, that `KTL_INHIBIT_FREE` prevents it from being freed (since the callback didn't allocate it), and that the final test on `status` and `flags` is to determine whether this was a final response from the DCS, in which case setting `context->state` to `KTL_FINAL_STATE` instructs KTL to delete the context that it allocated when the original message was received from the MUSIC client.

16 While all this is happening, any number of other transactions can be occurring, each of which will have a different context and will be entirely independent of this transaction.

```
int
periodic_callback( msg_type, client_data, message, context )
int msg_type;
void *client_data;
DCS_EXT_MSGS *message;
KTL_CONTEXT *context;
{
```

17 This is the periodic (1Hz) callback, which sends a DCS status request once a second, receives the response, and forwards it as a MUSIC broadcast (meaning that MUSIC automatically sends it to all clients which have expressed interest in it).

```
switch (context->local_state) {
      case 0:
            request.type = DCS_STATUS_REQ_IP_MSG_ID;
            request.len = sizeof( DCS_STATUS_REQ_IP_MSG );
            request.status = DCS__RET_OK; request.flags = 0;
            request.seq_num = ktl_seq_num();
            request.srcid = 0;
            request.destid = 0;
            if ( ktl_write( cara123_handle, KTL_NOTIFY, request.type,
                                    client_data, &request, context ) < 0 )
                  return -1;
            context->timeout        = 10000;
            context->state          = KTL_MESSAGE_STATE;
            context->local_state    = 1; break;
```

18 Here the DCS status request is built and sent, again with a 10 second timeout. Note the use of `ktl_seq_num()` to allocate a unique sequence number: responses to this request will all contain this sequence number and KTL uses it to look up the relevant sub-command context and hence command context.

```
case 1:
      if ( message->DCS_EXT_MSGS_u.msg_hdr.status &
                                                KTL_MESSAGE_STATUS_ERROR ) {
            printf( "In periodic_callback() DCS status request failed, \
             status = 0x%08.08x\n", message->DCS_EXT_MSGS_u.msg_hdr.status );
      }
      else if ( msg_type != DCS_TEL_STATUS_IP_MSG_ID ) {
            printf( "In periodic_callback() response is not DCS status!\n" );
```

```
      }
      else {
            msg_type = DCS_TEL_STATUS_MUSIC_BROAD;
            message->DCS_EXT_MSGS_u.msg_hdr.msg_type = msg_type;
            message->DCS_EXT_MSGS_u.msg_hdr.src_prc_id = 0;
            message->DCS_EXT_MSGS_u.msg_hdr.dst_prc_id = -1;
            message->DCS_EXT_MSGS_u.msg_hdr.usr_msg_id = 0;
            if ( ktl_write( music_handle, KTL_NOREPLY, msg_type, client_data,
                                                    message, context ) < 0 )
                  return -1;
      }

      context->timeout        = 1000;
      context->state          = KTL_DELAY_STATE;
      context->local_state    = 0;
      break;
}
return 1;
```

19 This also is fairly similar to the other callback. The biggest difference is the explicit call to `ktl_write()` to send the broadcast. Don't worry about any bits that you don't understand---they are probably to do with specific aspects of the `CARA123` interface definition.

Appendix B KTL kernel routines

The KTL kernel routines provide low-level facilities which are used not only within KTL but also within service/style-specific code. They are linked into a separate shareable library from the rest of the KTL routines.

ktl_log() should be used for logging all messages and errors. Whenever a routine detects an error, it should call ktl_log() to log the error, call ktl_set_errtxt() to set the corresponding error text, and return a negative (error) status. Whenever a routine receives an error status from a KTL call, it can call ktl_get_errtxt() to get the corresponding error text and ktl_log() to report it to the user.

B.1 ktl_get_errtxt() --- Return a pointer to the current KTL error text.

Abstract : Return a pointer to the current KTL error text.

Description : ktl_get_errtxt() returns the error text previously set using ktl_set_errtxt(). It can be used in the argument list to ktl_log().

Function value : char *: address of current KTL error text

Invocation :

char * ktl_get_errtxt()

Parameters : None

B.2 ktl_log()---Log a status message.

Abstract : Log a status message.

Description : ktl_log() should be used for logging all messages and errors. Eventually it will probably send all such traffic to a central logging task.

Function value : int: always 1

Invocation :

int ktl_log (dest, status, fmt, va_alist)

Parameters : None

> KTL_LOG_DEST dest;
message destination, a bit-mask with zero or more of the following set (if none are set, KTL_STDLOG is assumed):

KTL_STDLOG	log to standard logging destination as set using ktl_log_function() or ktl_log_stream()

KTL_STDOUT	log to standard output
KTL_STDERR	log to standard error

> KTL_LOG_STATUS status;
  indicates the severity of the message; should be one of:

KTL_INFO	informational message
KTL_WARN	warning message
KTL_ERROR	error message
KTL_DEBUG	debugging message

Eventually there will be a routine to set the current logging level and this will allow control over which classes of message are reported. Currently, messages with KTL_INFO are ignored and all others are logged.

> char* fmt;
  a printf() format string, followed by extra arguments as implied by the string.

B.3 ktl_log_default() --- Default log message output function.

Abstract : Default log message output function.

Description : ktl_log_default() is the default logging function. If ktl_log_stream() has been called, it writes the string to the stream that it specified. Otherwise it writes to standard error.

Function value : int: always 1

Invocation :

int ktl_log_default (buff)

Parameters : None

> char *buff;
  string to be logged.

B.4 ktl_log_function() --- Select current logging function.

Abstract : Select current logging function.

Description : ktl_log_function() specifies the function which is to be called with a single character string argument every time a message needs to be logged. Specify ktl_log_default() or NULL to revert to the default.

Function value : int: always 1

Invocation :

int ktl_log_function (function)

Parameters :

> `int (*function)();`
> address of logging function (`NULL` uses default)

B.5 ktl_log_stream() --- Select current logging stream.

Abstract : Select current logging stream.

Description : `ktl_log_stream`() sets the stream (`FILE *`) to use for logging. This is used only if the default logging function is used. It will write to the specified stream (`NULL` uses the default of logging to standard error).

Function value : int: always 1

Invocation :

int ktl_log_stream (stream)

Parameters :

> `FILE *stream;`
> stream to which to log (`NULL` uses default)

B.6 ktl_seq_num() --- Obtain a new positive sequence number.

Abstract : Obtain a new positive sequence number.

Description : `ktl_seq_num`() should always be used for generating sequence numbers in messages sent by `ktl_read`() and `ktl_write`() since such sequence numbers are used for matching responses to commands.

Function value : int: the new sequence number

Invocation :

int ktl_seq_num()

Parameters : None

B.7 ktl_set_errtxt() --- Save the current `KTL` error text in a static variable.

Abstract : Save the current `KTL` error text in a static variable.

Description : `ktl_set_errtxt`() should be called to set the error text whenever a routine returns a failure (negative) status. A temporary buffer is used for formatting the error text and a call to `ktl_get_errtxt`() can appear among `ktl_set_errtxt`()'s parameters.

Function value : char `*`: address of current `KTL` error text

Invocation :

char * ktl_set_errtxt (fmt, va_alist)

Parameters :

> `char* fmt;`
a `printf`() format string, followed by extra arguments as implied by the string.

Appendix C KTL high-level routines

The KTL high-level routines define the application programmer's interface to KTL. In general, application programmers should call only these and the kernel routines.

C.1 ktl_close() --- Close a connection to a server.

Abstract : Close a connection to a server.

Description : ktl_close() closes a connection to a server. Ideally, every successful ktl_open() call should have a corresponding ktl_close() call. In practice, ktl_close() calls are often omitted!

Method

1. Check that the handle is not NULL and that its close() function is not NULL.
2. Free storage associated with the handle.
3. Invoke the service-specific close() function to close the connection to the server.
4. Free the handle.

Function value : int: 1 if all OK, -1 otherwise

Invocation :

int ktl_close (handle)

Parameters :

> KTL_HANDLE *handle;
handle returned by ktl_open().

C.2 ktl_dispatch() --- Dispatch a KTL event for a specified service.

Abstract : Dispatch a KTL event for a specified service.

Description : ktl_dispatch() is called whenever an event may have occurred on a handle. It is NOT an error for it to be called when there are in fact no events to be dispatched on the handle. It can often be convenient to call ktl_dispatch() for all handles whenever an event occurs on any handle.

Method :

1. Check that the handle is not NULL and that its event() and respond() functions are not NULL.
2. Call the service/style-specific event() routine to check for events on one of the service's file descriptors and, if there are any, dispatch them.

3 Otherwise, if this service has enabled timer events (ie the `KTL_USETIMER` flag was specified when `ktl_open()` was called), check for timer events on this handle and, if there are any, dispatch them.

4 Timer and service/style-specific events are each dispatched in exactly the same way.

5 For each event, its type (command or response) is first determined. The `event()` routine may have specified the event type but, if not, if the application has expressed interest in the object name corresponding to the event, it is a command event. Otherwise it is a response event.

6 If it is a command event, a new context is created. If it is a response event, its sequence number must match an existing context (if negative, as it is for timer events) or sub-context (if positive).

7 Next the callback is invoked, and then `KTL` uses the context state to decide whether to delete the context, plus calls the service/style-specific `respond()` routine to send a response to the initiator of the command. Thus the rules for sending responses are service/style-specific. Existing keyword style implementations send no responses and message style implementations send responses after the first and last callback invocations (but the callback can override this policy).

Function value : int: number of dispatched events if all `OK`, -1 otherwise

Invocation :

int ktl_dispatch (handle)

Parameters :

> `KTL_HANDLE*handle;`
handle returned by `ktl_open()`.

C.3 ktl_ioctl() --- Control or return information about a service and its objects.

Abstract : Control or return information about a service and its objects.

Description : `ktl_ioctl()` is used in very much the same way as the Unix `ioctl()` system call. It is possible to add functionality to the `KTL` interface in an upwards-compatible way without adding new routines to the interface. It can also be used to send information to and receive information from service/style-specific code.

Method :

1 Check that the handle is not `NULL` and that its `ioctl()` function is not `NULL`.

2 Invoke the handle's `ioctl()` function to return the requested information.

3 If this service is using the `KTL` timer routines `ktl_open()` was called with the `KTL_USE-TIMER` flag), and if `ktl_ioctl()` was called with the `KTL_FDSET` flag, include the relevant timer fd in the returned `fd_set`.

Examples :

```
fd_set fdset;
status = ktl_ioctl( handle, KTL_FDSET, &fdset );
```

returns the `fd_set` (file descriptor set) in use by the service corresponding the supplied handle.

```
char buff[256];
KTL_POLYMORPH poly;
status = ktl_ioctl( handle, ''FRED'', KTL_FORMAT|KTL_ASCIN|
                                KTL_BINOUT, buff, &poly );
```

converts an `ASCII` value for the `FRED` keyword to binary. This conversion is keyword-dependent and may even include units conversion.

Notes : The flags parameter is used to determine which information should be returned by `ktl_-ioctl`(). The top two bytes of flags are service-specific and are not used by generic `KTL`. The third (0xff00) byte is regarded as a bit-mask and consists of mode bits which affect the interpretation of the low (0xff) byte (the defaults given in parentheses are mnemonic names, all of which translate to zero):

KTL_CONTINUOUS can be applied to `KTL_READ` to enable or disable a continuous read of an object (default `KTL_ONESHOT`)

KTL_DISABLE can be applied with `KTL_CONTINUOUS` to disable continuous reads or writes (default `KTL_ENABLE`)

KTL_NOPRIME can be applied with `KTL_CONTINUOUS` and `KTL_ENABLE` in order to disable "pump-priming", that is the reading of an initial value for the object in question (default `KTL_PRIME`)

KTL_BININ can be applied to `KTL_FORMAT` to indicate that the input is in binary (default `KTL_ASCIN`)

KTL_BINOUT can be applied to `KTL_FORMAT` to indicate that the output is to be in binary (default `KTL_ASCOUT`)

The fourth (0xff) byte is regarded as an enumeration, whose interpretation is modified by the above mode values. The following values for the fourth byte control or return information about the service and not about any particular object:

KTL_FDSET returns the set of file descriptors used by this service. If the service is using the `KTL` timer routines, this includes the `KTL` timer fd. This `fd_set` can be used to build an `fd_set` to pass to the `select`() system call. Provide a single parameter after the flags, of type `fd_set*` and pointing to an allocated `fd_set` array, to receive the `fd_set` (`fd_set` is defined in `<sys/types.h>`).

KTL_OBJECTS returns a list of all objects known to the service. The meaning of "object" is style-dependent. For the keyword style it means "keyword" and for the message style it means ``message type''. For other styles it could mean anything else. Provide two parameters after the flags, the first the address of an integer to receive the number of objects, the second the address of a variable which is to point at an array of object names (so, for example, for the keyword style this parameter will be of type `char***`). Free this array after use.

KTL_SETLOGFILE sets the name of a service- dependent log file. The effect of this is up to each service. Provide a single parameter after the flags, of type `char*`, containing the name of the log file.

KTL_GETLOGFILE gets the name of a service-dependent log file. The effect of this is up to each service. Provide a single parameter after the flags, of type `char **`, to receive a pointer to the name of the log file. Free it after use.

KTL_IMPLCAPS returns capabilities of the service / style implementation. It is used internally by generic `KTL` but applications can call it if they like. It returns a structure which currently contains only a single component: whether the implementation supports `ktl_read()` with the `KTL_NOTIFY` flag. Provide a single parameter after the flags, of type `KTL_IMPL_CAPS*` and pointing to an allocated `KTL_IMPL_CAPS` structure, to receive the capabilities `KTL_IMPL_CAPS` is defined in `ktl.h`).

KTL_NAMEDESCR returns details of the object names supported by this style implementation. It is used internally by generic `KTL` but applications can call it if they like. It returns a structure which contains information about how to format, compare, duplicate and free object names. Provide a single parameter after the flags, of type `KTL_NAME_DESCR*` and pointing to an allocated `KTL_NAME_DESCR` structure, to receive the capabilities (`KTL_NAME_DESCR` is defined in `ktl.h`).

KTL_DATADESCR returns details of the object data values supported by this style implementation. It is used internally by generic `KTL` but applications can call it if they like. It returns a structure which contains information about how to free data values, and also how to represent internal `KTL` events (rechedules and timeouts) in the style of the style (sic). Provide a single parameter after the flags, of type `KTL_DATA_DESCR*` and pointing to an allocated `KTL_DATA_DESCR` structure, to receive the capabilities (`KTL_DATA_DESCR` is defined in `ktl.h`).

The remaining values of the fourth byte all control or return information about a single object. They all expect the parameter after the flags to be an object name (the actual type is style-dependent). Generic applications may first have used the above `KTL_OBJECTS` to obtain a list of all objects.

KTL_READ makes an object readable or unreadable. It is used internally by generic `KTL` but applications can call it if they like. It supports the `KTL_CONTINUOUS` (`KTL_ONESHOT`), `KTL_DISABLE` (`KTL_ENABLE`) and `KTL_NOPRIME` (`KTL_PRIME`) mode flags. No parameters after the object name are required.

KTL_WRITE makes an object writeable or unwriteable. In all other respects, it is exactly like `KTL_READ`.

KTL_FORMAT converts object values between binary and `ASCII`. It supports all combinations of `KTL_BININ` (`KTL_ASCIN`) and `KTL_BINOUT` (`ASC_BINOUT`). Provide two parameters after the object name. The first is either of type `char*` (for `KTL_ASCIN`) or style-dependent (for `KTL_BININ`; in keyword style it is of type `KTL_POLYMORPH *`). The second is either of type `char**` (for `KTL_ASCOUT`) or style-dependent (for `KTL_BINOUT`; in keyword style it is of type `KTL_POLYMORPH*`). A returned `ASCII` value may contain newline characters but will not be terminated with a newline character. It should be freed after use. Whether or not to free a returned binary value is style-dependent (for keyword style, any arrays pointed to by the polymorph should be freed).

KTL_ALIASES returns a list of an object's aliases. In other respects it is very like `KTL_OBJECTS`. Provide two parameters after the object name, the first the address of an integer to receive the number of aliases, the second the address of a variable which is to point at an array of alias names (so, for example, for the keyword style this parameter will be of type `char***`). Free this array after use.

KTL_TYPE returns an object's type and size. It supports `KTL_BINOUT` (`KTL_ASCOUT`). Provide two parameters after the object name. The first is either of type `char**` (for `KTL_ASCOUT`) or style-dependent (for `KTL_BINOUT`; in keyword style it is of type `KTL_DATATYPE*`). The second is of type `int*` and is the address of an integer to receive the object's size in multiples of its type size (eg, 10 for an array of 10 integers or floats, but 78 for a message of length 78 bytes). A returned `ASCII` value should be freed after use.

KTL_UNITS returns an object's units. It supports `KTL_BINOUT` (`KTL_ASCOUT`). Provide two parameters after the object name, the first the address of an integer to receive the number of units values, the second the address of a variable which is to point at an array of units, either binary or `ASCII`. Free this array after use.

KTL_RANGE returns an object's range. Its interface is not yet specified.

KTL_FLAGS returns flags describing an object' s existence and state. Its interface is not yet specified.

KTL_BRIEFHELP returns a single-line string (with no newline character) giving brief help on an object. The single parameter after the object name is of type `char**` and will receive a copy of the help string. It should be freed after use.

KTL_FULLHELP returns full help (possibly containing newline characters but not terminated with a newline character) on an object. It may return the same information as `KTL_BRIEFHELP` and their interfaces are identical.

Function value : int: 1 if all `OK`, -1 otherwise

Invocation :

int ktl_ioctl(handle, flags, arg1, arg2, arg3, arg4, arg5, arg6, arg7, arg8)

Parameters :

`> KTL_HANDLE *handle;`
handle returned by `ktl_open`().

`> int flags;`
flags defining operation to perform, followed by up to four flags-dependent arguments. Refer to notes above for a description of the flags and of the parameters expected by each.

C.4 ktl_open() --- Open a connection to a server.

Abstract : Open a connection to a server.

Description : `ktl_open`() is normally the first `KTL` routine which is called by an application. It returns a handle which is used in all subsequent calls to high-level `KTL` routines.

Method :

1 Invoke `ktl_open1()` to create and initialize the `KTL` handle.

2 Invoke `ktl_open2()` to open the service/style-dependent shareable library and find the service/style-dependent functions.

3 Invoke `ktl_open3()` to invoke the service/style-dependent `open()` and `ioctl()` functions (`open()` to open the connection to the server and `ioctl()` to populate handle structures with information about the service/style implementation.

Notes : `ktl_open()` is broken out into `ktl_open1()`, `ktl_open2()` and `ktl_open3()` to allow the use of statically linked applications which do not activate shareable libraries at run-time. This can be useful, especially for debugging and perhaps also when shipping `KTL` applications to remote sites which do not have standard Keck directory structures set up. To link statically, the call to `ktl_open2()` can be replaced with a call to `ktl_register()`, which directly registers service/style-specific functions with the by-then-created handle.

Refer to the descriptions of `ktl_open1()` for further details on the `KTL` handle, and to `ktl_open2()` for further details on `KTL` shareable libraries.

Function value : int: 1 if all `OK`, -1 otherwise

Invocation :

int ktl_open (service, style, flags, handle)

Parameters :

`> char *service;`
name of underlying system to which to connect. Typically this is an instrument name, or the name of some well-defined server such as “dcs” or “catalog”.

`> char *style;`
name of style. Determines how the application interacts with the server. Only “keyword” and “message” styles have as yet been implemented but new ones can be implemented without altering `KTL`.

`> int flags;`
bit mask of `KTL` flags. The top two bytes are service/style-specific. The third (0xff00) byte defined a bit-mask:

KTL_USETIMER enable use of the `KTL` timer routines for this service. Necessary if the service plans to use `KTL` timeout and reschedule facilities (`KTL_NOTIMER` is also defined and is zero)

The low (0xff) byte is an enumerated type for which no further options are supported.

`< KTL_HANDLE **handle;`
address of a variable to receive a pointer to an allocated `KTL` handle. This handle is largely opaque to the application and should be used in all subsequent high-level `KTL` calls.

C.5 ktl_open1() --- Create and initialize a KTL handle.

Abstract : Create and initialize a KTL handle.

Description : ktl_open1() does the first part of ktl_open()'s job.

Method : Allocate the handle, copy the service and style names to it, and initialize all remaining fields to 0/NULL as appropriate.

Notes : The KTL handle contains all information necessary in order for the generic KTL system to access service/style-specific code. By and large, applications should treat it as an opaque pointer. However its definition is in fact visible to application code and it is legitimate to access its service, style and flags components to retrieve the values of the parameters of the same names on the ktl_open() call which created it. It is also legitimate to use the timer_handle component when directly manipulating timers. It is NOT legitimate to assume anything about the meaning of the the specific component.

Function value : int: 1 if all OK, -1 otherwise

Invocation :

int ktl_open1 (service, style, handle)

Parameters :

`> char *service;`
name of underlying system to which to connect. Typically this is an instrument name, or the name of some well-defined server such as "dcs" or "catalog".

`> char *style;`
name of style. Determines how the application interacts with the server. Only "keyword" and "message" styles have as yet been implemented but new ones can be implemented without altering KTL.

`< KTL_HANDLE **handle;`
address of a variable to receive a pointer to an allocated KTL handle. This handle is largely opaque to the application and should be used in all subsequent high-level KTL calls.

C.6 ktl_open2() --- Open shareable library and find functions.

Abstract : Open shareable library and find functions.

Description : ktl_open2() does the second part of ktl_open()'s job.

Method :

1 Concatenate the service name, "_", and the style name to give a first attempt at the name of the shareable library containing the service/style-specific routines.

2 Open the shareable library (see notes below for exactly how the shareable library is located).

3. Find the addresses of the seven service/style-dependent functions in the shareable library, storing the addresses in the handle. The entry points can be in either of two alternative forms. The first (and preferred) form is `ktl_<service>_<style>_<name>`, where `<service>` and `<style>` are the service and style parameters to `ktl_open`() and `<name>` is the actual service/style-dependent function, one of `open`, `ioctl`, `read`, `write`, `close`, `event` or `respond`. The alternative form is `<style>_<name>`.

Examples : Refer to the notes below for explanations of these examples. All assume that the `KTL_VERSION` macro has the value zero (as it does at the time of writing):

1. service= “hires”, style= “keyword”, `KTL_LIBRARY_PATH` undefined, `HIRES_KEYWORD` undefined: results in a library name of “./lib-hires_-keyword.-so.0.0”.
2. same as above but `KTL_LIBRARY_PATH`= “/kroot/bin”: results in a library name of “/kroot/-bin/-lib-hires_-keyword.so.0.0”.
3. same as 1 but `HIRES_KEYWORD`= “hires_keyword.1”: results in an error because the major versions mismatch, but it would have used a library name of “./lib-hires_-keyword.-so.1.0”.
4. same as 1 but `HIRES_KEYWORD`= “hires_keyword.0.1”: results in a library name of “./libhires_keyword.0.1”.

Notes : As noted above, the first attempt at the shareable library name is `<service>_` `<style>`. This is passed to the low-level `ktl_load_find`() routine which performs the following steps in locating the shareable library:

1. It attempts to translate the environment variable `KTL_LIBRARY_PATH`. If it fails, it assumes the value “.” (current directory). This is the directory in which the shareable library will be looked for. Note that at the time of writing `KTL_LIBRARY_PATH` can not have more than a single directory in its value - ie it is not really a path.
2. It converts the supplied library name to upper-case and attempts to translate it as an environment variable. If the translation fails, the original lower-case value is used.
3. This is where it begins to get Sun-specific (this is, as far as I know, the only Sun-specific bit in all of `KTL`). The resulting name is parsed as a string of the form “`lib<name>`.-so-.`<maj>`.`<min>`”, which is Sun shareable library name syntax. All the trailing three fields are optional and `<maj>` (major version) defaults to the value of the `KTL_VERSION` macro (defined in ktl.h).
4. The resulting major version is compared with the value of `KTL_VERSION` (they must match; it is an error if they don’t).
5. The full shareable library name is generated, and it is opened using the Sun-specific `dlopen`() call.
6. Note that in the absence of a specified minor version, it would be better to search for the existing file with the highest minor version, but this is not done.

Function value : int: 1 if all `OK`, -1 otherwise

Invocation :

int ktl_open2 (handle)

Parameters :

> `KTL_HANDLE *handle;`
handle returned by `ktl_open1`().

C.7 ktl_open3() --- Invoke service/style-dependent open functions.

Abstract : Invoke service/style-dependent open functions.

Description : `ktl_open3`() does the third and last part of `ktl_open`()'s job.

Method :

1. Check that the handle is not `NULL` and that its open() and `ioctl`() functions are not `NULL`.
2. Copy the `flags` parameter to the handle.
3. If the `KTL_USETIMER` flag is set, create a new timer handle.
4. Invoke the service/style-specific `open`() function to connect to the source of events on this handle.
5. Invoke the service/style-specific `ioctl`() function to return implementation capabilities, object name, and object data descriptions.

Function value : int: 1 if all `OK`, -1 otherwise

Invocation :

int ktl_open3(handle, flags)

Parameters :

> `KTL_HANDLE *handle;`
handle returned by `ktl_open1`().

> `int flags;`
bit mask of `KTL` flags. The top two bytes are service/style-specific. The third (0xff00) byte defined a bit-mask:

KTL_USETIMER enable use of the `KTL` timer routines for this service. Necessary if the service plans to use `KTL` timeout and reschedule facilities (`KTL_NOTIMER` is also defined and is zero)

The low (0xff) byte is an enumerated type for which no further options are supported.

C.8 ktl_read() --- Read a `KTL` keyword or message, continuously or once only.

Abstract : Read a `KTL` keyword or message, continuously or once only.

Description : ktl_read() occupies two slightly different roles. Firstly it handles the enabling and disabling of continuous reads of KTL objects (with a callback being invoked whenever a value becomes available). Secondly it handles one-shot reads. In the latter case, the caller can block awaiting the response, can perform other processing and then later block awaiting the response, or can ask that a callback be invoked when the response is received.

Method :

1 Check that the handle is not NULL and that its ioctl() and read() functions are not NULL.

If the KTL_CONTINUOUS flag is set:

2 If the KTL_ENABLE flag is set (the default; it is zero), invoke the handle's ioctl() function to enable continuous reads of the specified object and, if the KTL_PRIME flag is set (the default; it is zero), arrange for an initial value of the object to be supplied (via callback invocation). Add the object to this handle's interest list.

3 If the KTL_DISABLE flag is set, invoke the handle's ioctl() function to disable continuous reads of the specified object. Remove this object from the handle's interest list.

If the KTL_ONESHOT flag is set (the default; it is zero), switch on the low (enumerated) byte of the flags:

4 If it is KTL_NOTIFY, create a sub-context, invoke the handle's read() function (passing it the supplied flags), and return the sub-context's unique (positive) sequence number as the function value.

5 Otherwise (so no callback is to be invoked on completion), simply call the handle's read() function (passing it the supplied flags) and return its function value as the function value (should be a unique positive sequence number for KTL_NOWAIT).

Notes : The flags parameter is used to determine what ktl_read() should do and, under some circumstances, the interpretation of some of its parameters. As with all high-level KTL routines, the top two bytes of flags are service-specific and are not used by generic KTL.

The third (0xff00) byte is regarded as a bit-mask and consists of mode bits which affect the interpretation of the low (0xff) byte (the defaults given in parentheses are mnemonic names, all of which translate to zero):

KTL_CONTINUOUS enables or disables a continuous read of an object (default KTL_ONESHOT)

KTL_DISABLE can be used with KTL_CONTINUOUS to disable continuous reads of an object (default KTL_ENABLE)

KTL_NOPRIME can be used with KTL_CONINUOUS and KTL_ENABLE to prevent the automatic reading of an initial value for an object (default KTL_PRIME)

The fourth (0xff) byte is regarded as an enumeration, and is used when KTL_CONTINUOUS is not set, in which case it should have one of the following values:

KTL_WAIT indicates that ktl_read() should block until the object value is available. Use of KTL_WAIT is easiest to program but should be avoided in event-driven applications such as X user interfaces.

KTL_NOWAIT indicates that ktl_read() should initiate whatever transaction with the server is necessary and then return a unique sequence number to the caller, which can be used in a subsequent ktl_read() / KTL_WAITFOR call.

KTL_WAITFOR indicates that ktl_read() should block awaiting completion of a transaction previously initiated using ktl_read() / ktl_write() / KTL_NOWAIT. When KTL_WAITFOR is specified, the object_name parameter is the sequence number returned by the ktl_read() / ktl_write() call in question (cast to int).

KTL_NOREPLY is like KTL_NOWAIT except that the server is to be told to send no responses and no subsequent ktl_read() / KTL_WAITFOR call will be made.

KTL_NOTIFY is like KTL_NOWAIT except that a new sub-command context is created and the callback specified via the context parameter will be invoked whenever a server response to this request is received.

Note that not all service/style implementations will necessarily support all these flags. In particular, existing keyword style implementations do not support ktl_read() / KTL_NOTIFY (for historical reasons), and existing message style implementations only support ktl_read() / KTL_WAITFOR (since ktl_write() should be used to initiate message style transactions).

Function value : int: (positive) sequence number if OK and KTL_NOWAIT or KTL_NOTIFY is specified, 1 otherwise but OK, -1 not OK

Invocation :

int ktl_read(handle, flags, object_name, client_data, call_data, context)

Parameters :

> KTL_HANDLE *handle;
handle returned by ktl_open().

> int flags;
flags defining operation to perform. Refer to the notes above for a description of the flags.

> KTL_OBJECT_NAME object_name;
object name; actual type is dependent on style. When KTL_WAITFOR is specified, it is the sequence number returned by the matching ktl_read() / ktl_write() / KTL_NOWAIT call (cast to int).

> void *client_data;
client data for callback; this is used only when KTL_NOTIFY is specified and will be passed as the callback's client_data parameter.

< void *call_data;
object value (strange parameter name is for compatibility with callback parameter names); this is used only when KTL_WAIT or KTL_WAITFOR is specified. Actual type is dependent on style.

> KTL_CONTEXT *context;

command context pointer; used only when KTL_CONTINUOUS (with KTL_ENABLE) or KTL_NOTIFY are specified. With KTL_CONTINUOUS, only its callback component is used; with KTL_NOTIFY, if calling ktl_read() from a callback it will be the callback's context parameter; otherwise (eg, if calling from main program or X callback) it must have been created using ktl_context_create().

C.9 ktl_register() --- Register service/style-dependent functions.

Abstract : Register service/style-dependent functions.

Description : ktl_register() can be used to avoid the use of shareable libraries for service/style-specific code. Rather than using ktl_open() (which itself calls ktl_open1(), ktl_open2() and ktl_-open3()), ktl_open1() can be called to create the KTL handle, ktl_register() can be called to register the service/style-specific functions, and ktl_open3() can be used to open a connection to the server.

Method :

1. Check that the handle is not NULL.
2. Step through the list of functions, validating their names and copying their addresses to the handle.

Notes : Each KTL_REGISTER_ARG entry consists of the function name, one of (case-insensitive) open, ioctl, read, write, close, event, respond, and the function address. The list is terminated by an entry with a NULL function name.

The following is an example of a valid declaration of a KTL_REGISTER_ARG array:

```
KTL_REGISTER_ARG regarg[] = {
      {"open",    keyword_open },
      {"ioctl",   keyword_ioctl },
      {"read",    keyword_read },
      {"write",   keyword_write },
      {"close",   keyword_close },
      {"event",   keyword_event },
      {"respond", keyword_respond},
      {NULL}
};
```

Function value : int: 1 if all OK, -1 otherwise

Invocation :

int ktl_register(handle, functions)

Parameters :

> KTL_HANDLE *handle;
handle returned by ktl_open().

> KTL_REGISTER_ARG *functions;
NULL terminated list of function names and addresses. Refer to the above for more information.

C.10 ktl_write() --- Write a KTL keyword or message.

Abstract : Write a KTL keyword or message.

Description : ktl_write() is in many respects complementary to ktl_read(). However, there is no concept of “continuous writes” and ktl_write() does not support the KTL_CONTINUOUS flag. For the keyword style (and any future styles where there is an obvious distinction between reading and writing an object), ktl_read() is used to read the value of an object and ktl_write() is used to write the value of an object. For the message style, there is no distinction between a message that reads values and a message that writes values (the difference is in the semantics of the messages and KTL does not concern itself with this); hence for the message style ktl_write() is used to initiate all transactions and ktl_read() is used only with KTL_CONTINUOUS and KTL_WAITFOR.

Method :

1 Check that the handle is not NULL and that its write() function is not NULL.

2 Mask off the low byte of the flags and switch on it.

3 If it is KTL_NOTIFY, create a sub-context, invoke the handle’s write() function (passing it the supplied flags), and return the sub-context’s unique (positive) sequence number as the function value.

4 Otherwise (so no callback is to be invoked on completion), simply call the handle’s write() function (passing it the supplied flags) and return its function value as the function value (should be a unique positive sequence number for KTL_NOWAIT).

Notes : The flags parameter is used to determine what ktl_write() should do. As with all high-level KTL routines, the top two bytes of flags are service-specific and are not used by generic KTL.

The fourth (0xff) byte is regarded as an enumeration, and should have one of the following values:

KTL_WAIT indicates that ktl_write() should block until the object value has been written. Use of KTL_WAIT is easiest to program but should be avoided in event-driven applications such as X user interfaces.

KTL_NOWAIT indicates that ktl_write() should initiate whatever transaction with the server is necessary and then return a unique sequence number to the caller, which can be used in a subsequent ktl_read() / KTL_WAITFOR call.

KTL_NOREPLY is like KTL_NOWAIT except that the server is to be told to send no responses and no subsequent ktl_write() / KTL_WAITFOR call will be made.

KTL_NOTIFY is like KTL_NOWAIT except that a new sub-command context is created and the callback specified via the context parameter will be invoked whenever a server response to this request is received.

Function value : int: (positive) sequence number if OK and KTL_NOWAIT or KTL_NOTIFY is specified, 1 otherwise but OK, -1 not OK

Invocation :

int ktl_write (handle, flags, object_name, client_data, call_data, context)

Parameters :

> `KTL_HANDLE *handle;`
handle returned by `ktl_open`().

> `int flags;`
flags defining operation to perform. Refer to the notes above for a description of the flags.

> `KTL_OBJECT_NAME object_name;`
object name; actual type is dependent on style.

> `void *client_data;`
client data for callback; this is used only when `KTL_NOTIFY` is specified and will be passed as the callback's `client_data` parameter.

< `void *call_data;`
object value (strange parameter name is for compatibility with callback parameter names); this is used only when `KTL_WAIT` is specified. Actual type is dependent on style.

> `KTL_CONTEXT *context;`
command context pointer; used only when `KTL_NOTIFY` is specified. If calling `ktl_write`() from a callback it will be the callback's context parameter; otherwise (eg, if calling from main program or `X` callback) it must have been created using `ktl_context_create`().

Appendix D KTL context routines

The KTL context routines manage creation, manipulation and deletion of command contexts and sub-contexts.

Application programmers need to access one KTL context enumerated type (the command state) and two structures (the KTL sub-context and the KTL context).

The only routine which application programmers will normally need to call is ktl_context_create(). This is necessary when an internal event (such as an X event) has occurred and it is desired to create a context.

D.1 KTL_COMMAND_STATE --- used by callbacks to tell KTL what to do next.

Abstract : used by callbacks to tell KTL what to do next.

Description : The KTL command state is held in context->state, is always KTL_INITIAL_STATE for a callback's first entry, and is changed only by the callback (except when it is set to KTL_ERROR_STATE). It indicates to KTL the circumstances under which the callback should next be invoked.

Enum values :

KTL_INITIAL_STATE, initial state: always set to this for a callback's first entry. On exit from a callback treated the same way as KTL_FINAL_STATE.

KTL_RESCHEDULE_STATE, reschedule state: set the state to this in order to re-invoke the callback immediately (but after servicing any queued events).

KTL_DELAY_STATE, delay state: like KTL_RESCHEDULE_STATE but re-invoke the callback after context->timeout milliseconds.

KTL_MESSAGE_STATE, message state: re-invoke the callback when a response is received or on expiry of context->timeout milliseconds (-1 means no timeout).

KTL_FINAL_STATE, final state: command is complete and context should be deleted.

KTL_ERROR_STATE error state: KTL detected error, callback had context->flags KTL_RESCHEDULE_ON_ERROR flag, and KTL has re-invoked callback.

D.2 KTL_CONTEXT --- structure created when command event occurs.

Abstract : structure created when command event occurs.

Description : A command context (or context) is created whenever a command event (event in which an application has expressed interest) occurs. It contains information about the command event and its source, plus tracks the progress of the command and of any sub-commands (ktl_read() or ktl_write() calls with the KTL_NOTIFY flag) that it may initiate.

Contexts can also be created directly using `ktl_context_create`(). This is useful when internal events (such as `X` events) need to initiate transactions with other tasks.

Struct fields : set on context creation and never subsequently changed.

`int (*callback)();`
: pointer to the callback function associated with the command which created the context.

`void *create_handle;`
: pointer to the `KTL` handle which created this context.

`void *client_data;`
: arbitrary piece of data to be passed to callback invocations associated with this context.

`void *call_data;`
: pointer to event data (message) which caused creation of this context.

`void *timer_entry;`
: opaque timer entry, used for rescheduling and timeouts.

`int sequence_number;`
: unique negative command sequence number (decremented by 1 for each context allocated).

Struct fields : with well-defined values on each entry to callbacks and not altered in callbacks.

`KTL_SUBCONTEXT *subcontext;`
: pointer to sub-context corresponding to current event, `NULL` if none (eg, timer event).

`void *event_handle;`
: pointer to handle on which event arrived, `NULL` if none (eg, timer event).

`int done;`
: boolean indicating whether this is the last response associated with a sub-context, or else is a timeout in `KTL_MESSAGE_STATE` state.

`int status;`
: status corresponding to current event. For associated with a sub-context, this is extracted from the message which caused the event. For timer events it is -1 for timeout in `KTL_MESSAGE_STATE`, 1 otherwise. \

Struct fields : with well-defined values on context creation and altered subsequently only in callbacks, except as noted.

`KTL_COMMAND_STATE old_state;`
: always contains the command state on entry to a callback (used for deciding whether to send responses.

`KTL_COMMAND_STATE state;`
: current command state guaranteed to be `KTL_INITIAL_STATE` on command's initial entry, `KTL_ERROR_STATE` on error if "reschedule on error" flag is set and otherwise altered only in callbacks.

`int local_state;`

current local command state guaranteed to be 0 on a command's initial entry, -1 on error as above and otherwise altered only in callbacks.

`void *user_area;`
: pointer guaranteed to be `NULL` on a command's initial entry and altered only in callbacks; it is the user's responsibility to free it if nec- essary.

`int timeout;`
: timeout period in milliseconds, used when the command state is `KTL_DELAY_STATE` or `KTL_MESSAGE_STATE` and altered only in callbacks; initially -1 (infinite timeout).

Struct fields : with well-defined values on each entry to callbacks and

`potentially altered in callbacks - only directly relevant to`
: message styles.

`void *response_msg;`
: pointer to service/style-specific message which is to be returned to the initiator of this command; `NULL` on entry.

`int flags;`
: flags controlling sending of response etc; 0 on entry; high two bytes are service/style-specific.

`KTL_INHIBIT_REPLY`	inhibits sending a default reply.
`KTL_INHIBIT_FREE`	inhibits freeing of `context->response_msg`
`KTL_RESCHEDULE_ON_ERROR`	re-invokes the callback if `ktl_dispatch()` encounters an error sending a reply.
`KTL_DELETE_ON_ERROR`	deletes the context if `ktl_dispatch()` encounters an error sending a reply.

D.3 `KTL_SUBCONTEXT` --- structure created when `KTL_NOTIFY` flag is specified.

Abstract : structure created when `KTL_NOTIFY` flag is specified.

Description : A sub-command context (or sub-context) is created whenever `ktl_read()` or `ktl_-write()` is called with the `KTL_NOTIFY` flag. It contains a unique sequence number which allows `KTL` to match up responses with sub-contexts and thus contexts. The sub-context is automatically deleted when the transaction initiated with `ktl_read()` or `ktl_write()` completes.

When a callback is invoked as a result of a response message, `context->subcontext` points to the associated sub-context structure.

Struct fields : none should be altered by callbacks.

`KTL_CONTEXT *context;`
: pointer to parent context structure.

`int done;`

boolean indicating whether this is the last response for this sub-context (also available in `context->done`).

`int sequence_number;`
unique sub-command sequence number (incremented by 1 for each new sub-context and allocated `ktl_seq_num()`).

`caddr_t object_name;`
name of object corresponding to this sub-context (interpretation is style-dependent).

`void *client_data;`
arbitrary piece of data supplied by caller and passed as the `client_data` parameter to callbacks associated with this sub-context.

D.4 ktl_context_create() --- create a command context.

Abstract : create a command context.

Description : `ktl_context_create()` creates a command context, which survives for the life of a command, tracks sub-commands initiated by a command, and contains state information allowing both `KTL` and callbacks to keep track of what is going on.

Function value : int: 1 if all `OK`, -1 otherwise

Invocation :

int ktl_context_create (handle, callback, client_data, call_data, context)

Parameters :

`> void *handle;`
handle returned by `ktl_open()`.

`> int (*callback)();`
callback that handles context state changes.

`> void *client_data;`
user data that gets passed to the callback.

`> void *call_data;`
call data resulting in context creation.

`< KTL_CONTEXT **context;`
address of variable to receive pointer to the allocated context structure.

Appendix E KTL timer routines

The KTL timer routines provide a set of general-purpose timer management facilities using only the Unix signal() and alarm() facilities (this means that timer resolution is one second; a higher resolution would require use of less portable facilities such as the SunOS ualarm() routine).

These routines are used internally within KTL but can be used independently by applications programmers without interfering with KTL.

ktl_timer_open() returns a handle which is passed to as many ktl_timer_create() calls as required, each of which creates a timer entry associated with that handle. The resulting family of timers exist independently of timers associated with other handles. ktl_timer_delete() deletes a timer and ktl_timer_close() deletes all timers associated with a handle, and then the handle itself.

ktl_timer_arm() arms a timer, specifying an interval in milliseconds before that timer should fire. One of its parameters is a piece of user data which is made available when the timer fires. ktl_timer_event() returns a list of user data items for timers which have fired.

The implementation uses SIGALRM signals: ktl_timer_alarm() is the signal handler and traverses a single list of timer entries, decrementing their countdown fields and, when a timer is ready to fire, it writes a message to the pipe corresponding to its parent handle (each handle has its own pipe). These messages are read by ktl_timer_event().

It is intended that users of these routines will use the ktl_timer_fd() routine to retrieve the fd (file descriptor) on which the above-mentioned messages will arrive, and add it to the fd_set supplied to the select() system call. When select() returns, ktl_timer_event() will be called to determine whether any timer events occurred (this is how KTL uses the routines).

E.1 ktl_timer_alarm() --- KTL SIGALRM signal handler routine.

Abstract : KTL SIGALRM signal handler routine.

Description : ktl_timer_alarm() should execute at whatever rate KTL requires wakeups to be serviced, typically 1 Hz and possibly as fast as 10 Hz in the future.

Function value : int: 1 if all OK, -1 otherwise

Invocation :

void ktl_timer_alarm()

Parameters : None

E.2 ktl_timer_arm() --- KTL timer arm routine.

Abstract : KTL timer arm routine.

Description : `ktl_timer_arm`() arranges for a timer to fire in a specified number of milliseconds. A negative number of milliseconds disarms the timer.

Function value : int: 1 if all `OK`, -1 otherwise

Invocation :

int ktl_timer_arm (entry, milliseconds, user_data)

Parameters :

> `KTL_TIMER_ENTRY *entry;`
timer entry pointer

> `int milliseconds;`
number of milliseconds before firing

> `void *user_data;`
opaque user data (not used with negative numbers of milliseconds)

E.3 ktl_timer_close() --- Close (delete) a `KTL` timer family.

Abstract : Close (delete) a `KTL` timer family.

Description : `ktl_timer_close`() closes (deletes) a timer family which was opened by `ktl_timer_open`(). It also deletes any timer entries associated with the supplied timer handle.

Function value : int: 1 if all `OK`, -1 otherwise

Invocation :

int ktl_timer_close(handle)

Parameters :

> `KTL_TIMER_HANDLE *handle;`
timer handle pointer

E.4 ktl_timer_create() --- Create a `KTL` timer associated with an existing handle.

Abstract : Create a `KTL` timer associated with an existing handle.

Function value : int: 1 if all `OK`, -1 otherwise

Invocation :

int ktl_timer_create (handle, entry)

Parameters :

> `KTL_TIMER_HANDLE *handle;`

timer handle pointer

`< KTL_TIMER_ENTRY **entry;`
address of variable to receive timer entry

E.5 ktl_timer_delete() --- Delete a KTL timer.

Abstract : Delete a KTL timer.

Function value : int: 1 if all OK, -1 otherwise

Invocation :

int ktl_timer_delete (entry)

Parameters :

`> KTL_TIMER_ENTRY *entry;`
timer entry pointer

E.6 ktl_timer_event() --- KTL timer event routine.

Abstract : KTL timer event routine.

Description : `ktl_timer_event()` reads all timer messages associated with the supplied handle and converts them into an array of user data values.

Function value : int: 1 if all OK, -1 otherwise

Invocation :

int ktl_timer_event (handle, maxevent, nevent, user_data)

Parameters :

`> KTL_TIMER_HANDLE *handle;`
timer handle pointer

`> int maxevent;`
max # events to return (-1 means all and 0 means to inquire number of waiting events)

`< int *nevent;`
address of var to receive # events; always returned as number of available events

`< void ***user_data;`
address of var to receive pointer to user data array (free after use); no more than maxevent events are returned

E.7 kt_timer_fd() --- KTL timer pipe fd enquiry routine.

Abstract : KTL timer pipe fd enquiry routine.

Function value : int: input pipe fd, -1 if handle is `NULL`.

Invocation :

int ktl_timer_fd(handle)

Parameters :

> `> KTL_TIMER_HANDLE *handle;`
> timer handle pointer

E.8 ktl_timer_open() --- Open (create) a `KTL` timer family.

Abstract : Open (create) a `KTL` timer family.

Description : `ktl_timer_open()` opens (creates) a `KTL` timer family. Once the family has been opened, `ktl_timer_create()` can be used to create timer entries.

Function value : int: 1 if all `OK`, -1 otherwise

Invocation :

int ktl_timer_open(handle)

Parameters :

> `< KTL_TIMER_HANDLE **handle;`
> address of variable to receive timer handle pointer

Appendix F KTL list routines

The KTL list routines provide a set of generic list-handling facilities which are used by KTL and which can be used by application programmers.

F.1 ktl_list_add() --- Add an entry to the head of the linked list.

Abstract : Add an entry to the head of the linked list.

Function value : int: 1 if all OK, -1 otherwise

Invocation :

int ktl_list_add (dptr, head)

Parameters :

`> KTL_LIST_DATA *dptr;`
pointer to generic data.

`! KTL_LIST_ENTRY **head;`
addr of pointer to first list entry.

F.2 ktl_list_add_tail() --- Add an entry to the tail of the linked list.

Abstract : Add an entry to the tail of the linked list.

Function value : int: 1 if all OK, -1 otherwise

Invocation :

int ktl_list_add_tail (dptr, head)

Parameters :

`> KTL_LIST_DATA *dptr;`
pointer to generic data.

`! KTL_LIST_ENTRY **head;`
addr of pointer to first list entry.

F.3 ktl_list_count() --- Count the number of entries on the linked list.

Abstract : Count the number of entries on the linked list.

Function value : int: always 1

Invocation :

int ktl_list_count (head, count)

Parameters :

> `> KTL_LIST_ENTRY *head;`
> pointer to first list entry.

> `< int_*count;`
> count of list entries.

F.4 ktl_list_count_ptr() --- Count list entry with pointer field matching provided pointer.

Abstract : Count list entry with pointer field matching provided pointer.

Function value : int: always 1

Invocation :

int ktl_list_count_ptr (ptr, head, offset, count)

Parameters :

> `> void *ptr;`
> the ptr to search for.

> `> KTL_LIST_ENTRY *head;`
> addr of pointer to first list entry.

> `> int offset;`
> offset in bytes into the data area.

> `< int *count;`
> count of structures which have match.

F.5 ktl_list_delete() --- Delete an entry from a linked list.

Abstract : Delete an entry from a linked list.

Function value : int: 1 if all `OK`, -1 otherwise

Invocation :

int ktl_list_delete (dptr, head)

Parameters :

> `> KTL_LIST_ENTRY *dptr;`
> pointer to entry to be deleted.

> `! KTL_LIST_ENTRY **head;`
> addr of pointer to first list entry.

F.6 ktl_list_delete_all() --- Delete an entire linked list.

Abstract : Delete an entire linked list.

Function value : int: always 1

Invocation :

int ktl_list_delete_all (head)

Parameters :

`! KTL_LIST_ENTRY **head;`
addr of pointer to first list entry.

F.7 ktl_list_find() --- Find list entry using supplied routine to match data field

Abstract : Find list entry using supplied routine to match data field with supplied search key.

Function value : int: 1 if all `OK`, -1 otherwise

Invocation :

int ktl_list_find (key, compare, head, offset, dptr)

Parameters :

`> char *key;`
the data to search for.

`> int (*compare)();`
data comparison routine (same interface as `strcmp`()).

`>KTL_LIST_ENTRY *head;`
addr of pointer to first list entry.

`> int offset;`
offset in bytes into the data area.

`< KTL_LIST_DATA **dptr;`
pointer to the found list entry.

F.8 ktl_list_find_entry() --- Find list entry whose data address matches the provided pointer.

Abstract : Find list entry whose data address matches the provided pointer.

Function value : int: 1 if all `OK`, -1 otherwise

Invocation :

int ktl_list_find_entry (dptr, head, eptr)

Parameters :

> `> KTL_LIST_DATA *dptr;`
> the data entry to search for.

> `> KTL_LIST_ENTRY *head;`
> addr of pointer to first list entry.

> `< KTL_LIST_ENTRY **eptr;`
> pointer to the found list entry.

F.9 ktl_list_find_int() -- Find list entry with data field matching provided int.

Abstract : Find list entry with data field matching provided int.

Function value : int: 1 if all `OK`, -1 otherwise

Invocation :

int ktl_list_find_int (idata, head, offset, dptr)

Parameters :

> `> int idata;`
> the int to search for.

> `> KTL_LIST_ENTRY *head;`
> addr of pointer to first list entry.

> `> int offset;`
> offset in bytes into the data area.

> `< KTL_LIST_DATA **dptr;`
> pointer to the found list entry.

F.10 ktl_list_find_string() --- Find list entry with data field matching provided string.

Abstract : Find list entry with data field matching provided string.

Function value : int: 1 if all `OK`, -1 otherwise

Invocation :

int ktl_list_find_string (string, head, offset, dptr)

Parameters :

> `> char *string;`
> the string to search for.

> `> KTL_LIST_ENTRY *head;`
> addr of pointer to first list entry.

> `int offset;`
  offset in bytes into the data area.

< `KTL_LIST_DATA **dptr;`
  pointer to the found list entry.

F.11 ktl_list_first() --- Find first list entry.

Abstract : Find first list entry.

Function value : `KTL_LIST_DATA *`: data pointer from the list's first entry, `NULL` if the list is empty. The first list entry is returned as a parameter, ready to be passed to `ktl_list_next()`.

Invocation :

`KTL_LIST_DATA *` ktl_list_first (head, eptr)

Parameters :

> `KTL_LIST_ENTRY *head;`
  pointer to first list entry

< `KTL_LIST_ENTRY **eptr;`
  address of variable to receive first list entry

F.12 ktl_list_next() --- Find next list entry.

Abstract : Find next list entry.

Function value : `KTL_LIST_DATA *`: data pointer from the list's next entry, `NULL` if there are no more entries.

Invocation :

`KTL_LIST_DATA*` ktl_list_next (eptr)

Parameters :

! `KTL_LIST_ENTRY **eptr;`
  on entry points to current entry; on exit points to next entry.