W. M. KECK OBSERVATORY KSD/KTL_API_8
Keck Software Document 8
Version 7

Al Conrad and William Lupton
26-Dec-91

KTL: the Keck Task Library

Contents

1 Introduction **3**

2 Historical Background **3**

3 Descriptions of KTL functions **4**
3.1 ktl_init — initialize KTL 4
3.2 ktl_open — open a connection to a server 6
3.3 ktl_read — read a keyword or a message 7
3.4 ktl_write — set a keyword or send a message 10
3.5 ktl_handle_event — handle an event 12
3.6 ktl_close — close a connection to a server 12
3.7 ktl_shut — shut down KTL 13
3.8 ktl_log — log a message 13
3.9 The KTL_HANDLE structure 14
3.10 Function return values 14
3.11 Callback function parameters 15
3.12 Other functions 15

4 Handling context **16**
4.1 ktl_context_init — initialize the context facility 16
4.2 ktl_context_open — open up the context facility for a service 17
4.3 ktl_context_create — create and initialize a context structure 17
4.4 ktl_context_lookup — look up the context structure corresponding to an event . . 18
4.5 ktl_context_postproc — perform post-processing after callback invocation 18
4.6 ktl_context_respond — respond to initiator of command 18
4.7 ktl_context_nextentry — arrange for next callback entry 19
4.8 ktl_context_delete — delete a context structure 20
4.9 ktl_context_close — close down the context facility for a service 20
4.10 ktl_context_shut — shut down the context facility 20
4.11 ktl_subcontext_create — create and initialize a sub-context structure 21
4.12 ktl_subcontext_lookup — look up the sub-context structure corresponding to an event 21

4.13 `ktl_subcontext_delete` — delete a sub-context structure . . . 22
4.14 The `KTL_CONTEXT` structure . . . 22
4.15 The `KTL_SUBCONTEXT` structure . . . 23

5 Implementing `KEYWORD`-style `KTL` for the `MUSIC` service 24
5.1 `ktl_read/KTL_WAIT` . . . 24
5.2 `ktl_read/KTL_EXPRESS_INTEREST` . . . 24
5.3 `ktl_write` . . . 24
5.4 `ktl_handle_event` . . . 24

6 Implementing `KTL` for the `RPC` service 25

7 Implementing `KTL` for the `MESSAGE` style 25
7.1 Main program . . . 25
7.2 `ktl_init` . . . 25
7.3 `ktl_open` . . . 26
7.4 `ktl_read` . . . 26
7.5 `ktl_write` . . . 27
7.6 `ktl_handle_event` . . . 27
7.7 `ktl_close` . . . 28
7.8 `ktl_shut` . . . 28

8 Outstanding work 28

9 Changes to this document 29

A `KTL` Identifier Names 30

1 Introduction

This document describes the Keck Task Library (`KTL`). `KTL` will be used in the near future for:

- `X` windows user interface for the optical instruments.
- `X` windows user interface and server tasks for the autoguider.
- `X` windows user interface for `LWS`.

`KTL` applications have the following form (with corresponding function names given in parentheses):

1. Initialize (`ktl_init`).
2. Open connections to one or more servers (`ktl_open`).
3. Express interest in one or more events (`ktl_read`).
4. Loop forever handling incoming events (`ktl_handle_event`).

`KTL` also provides functions for interacting with other parts of an instrumentation system, and for closing server connections.

§2 gives some historical background. The `KTL` functions are described in §3. §4 explains what is meant by context and how it is handled. §5–§7 describe specific implementations of `KTL` services. Refer to the bibliography for background information.

This document is just a proposal. We hope that there will be no substantial changes but we do not rule it out. Please feed back any remarks, no matter how trivial, to the authors: `AConrad@Keck.Hawaii.Edu` and `WLupton@Keck.Hawaii.Edu`.

2 Historical Background

Two existing packages preceded `KTL`:

- `CCX`: The command context layer above `DCS` message passing, and
- `KICS`: the keyword layer above instrument message passing.

Although both of these packages provide a layer above message passing, their aims are distinct:

A `CCX` task is event driven, can be *any* task in the system, and does *not* hide details of the message system.

A `KICS` task is *not* event driven, is *specifically* a user interface task, and *does* hide details of the message system.

The KTL design began as an extension to KICS for X windows. An X windows task is event driven, so it is not surprising that the X windows extension to KICS took on much of the look of CCX. At the same time, there are those who argue that a CCX *user interface* task should hide the details of the message system.

Thus we have KTL. A KTL task is event driven and is either:

1. a user interface task which uses KTL to access the control system via keywords (referred to as the KEYWORD style), or

2. a task of any type which uses KTL to handles messages directly (referred to as the MESSAGE style).

Figure 1: Use of Message vs Keyword Style.

Figure 1 shows how the two KTL styles fit into a typical application. Figure 2 shows how a very preliminary version of KTL has been used for a very preliminary display of optical instrument CCD status.

3 Descriptions of KTL functions

3.1 `ktl_init` — initialize KTL

`ktl_init` initializes KTL. It should be called once at the top of the application's main program. It exists to perform service/style-independent initialization such as argument processing and the initialization of the context facilities.

`ktl_init` uses similar conventions to those of the X windows system for processing command-line arguments. Typically it will be passed the same `argc` and `argv` arguments that are passed to C main programs, except that a *pointer* to `argc` is passed. It will remove any arguments that it understands, leaving the remaining arguments to be processed by the application. If the KTL application is also an X application, X should be given first stab at the argument list.

Figure 2: Use of KTL for CCD status display. Note that the program only makes use of the KTL KEYWORD style. If the MESSAGE style is incorporated, the layers labelled MUSIC will be repackaged in the KTL MESSAGE style.

The calling sequence for `ktl_init` is:

```
int
ktl_init( argc, argv )
int *argc;
char *argv[];
```

The `argc` parameter is a *pointer* to the number of arguments pointed to by `argv`. On exit this number will be modified to be the number of arguments still to be processed.

The `argv` parameter is a pointer to an array of pointers to arguments. `ktl_init` will process argument strings that it understands and will adjust `argv` such that it only contains pointers to arguments which are still to be processed. Arguments understood by `ktl_init` are:

`-host hostname`	This host's name as far as KTL is concerned; defaults to the string returned by `gethostname()`.
`-task taskname`	This task's name as far as KTL is concerned; defaults to the name part of `argv[0]`, which is by convention the application name.

The argument names can be abbreviated as long as they are not ambiguous, but this is not advisable, since abbreviations may conflict with X argument names (the only current potential conflict is between `title` and `task`) or application-specific argument names. The white space between the argument name and its value is optional.

3.2 `ktl_open` — open a connection to a server

`ktl_open` opens a connection to a server, specifying a service (which indicates the underlying message system and possibly other information) and a style (which indicates whether the application will work in terms of keywords or messages).

In the following table, the service/style combinations which will definitely be implemented are indicated with a $\times$. The numbers in the other entries indicate the likely implementation orders for the remaining combinations.

	KEYWORD style	MESSAGE style
MUSIC services	$\times$	2
RPC services	$\times$	3
DCS services	1	$\times$

The syntax *service/style* refers to a particular service and style; *e.g.* DCS/MESSAGE refers to the DCS service and the MESSAGE style.

Note that KTL may simultaneously be open for an unrestricted number of service/style combinations.

The calling sequence for `ktl_open` is:

```
int
ktl_open( service, style, flags, handle )
char *service;
char *style;
unsigned int flags;
KTL_HANDLE **handle;
```

The `service` parameter is the name of the service. Possible services are:

`MUSIC`: `MUSIC` messages are used.

`RPC`: `RPC`s are used.

`DCS`: `DCS` messages are used (even though MUSIC may be the underlying message system).

Where a service requires additional information, an alias may be used, which will be looked up in a standard place and used to derive both the service and the additional information. Where `MUSIC` is available, the standard `MUSIC` `services` file will be used. For example,

Optical instrument user interfaces will use the `MUSIC/KEYWORD` service/style, but `KTL` also needs to know the actual instrument. Thus the actual `service` parameter will be `hires` or `lris`, resulting in separate `hires` and `lris` handles.

The `DCS` service needs to know details of Internet addresses, port numbers etc. To read commands from the `DCS`, the actual `service` parameter might be `DCS_COMMAND`, which would be translated to give the `DCS` service and the relevant additional information.

The `style` parameter indicates the programming style employed by the application. Not all services need to support all styles. Possible styles are:

`KEYWORD`: details of the underlying message system are hidden and all events are identified by keyword name. Applications written in this style will normally be user interfaces and cannot handle requests from other tasks to write and read keyword values.

`MESSAGE`: messages are handled directly in application code (the actual message structure is implied by the service). Applications written in this style will normally be low-level tasks which both service requests from other tasks and issue requests of their own.

The `flags` parameter is a bit-mask of flags which provide extra information about the requested event. Flags in the low two bytes are generic to all services/styles but need not be supported by all services/styles. No generic flags are currently defined. Flags in the high two bytes are specific to individual services/styles.

The `handle` parameter is the address of a variable to receive a pointer to a structure containing information derived from the `service`, `style` and `flags` parameters. It contains all information necessary to communicate over the underlying message system and its contents is described in §3.9.

3.3 `ktl_read` — read a keyword or a message

The `ktl_read` function is the successor to the `KICS` function `cread`. It provides the functionality of `cread` plus extra functionality to support new `KTL` facilities. Note that the only uses of `ktl_read` for the MESSAGE style are with KTL_WAITFOR to await completion of a transaction previously initiated with ktl_write, and with KTL_EXPRESS_INTEREST and KTL_CANCEL_INTEREST to control behavior in response to asynchronous events.

Several reading modes are available and are controlled by the flags parameter. In the simplest case ktl_read is called with KTL_WAIT and blocks until the specified keyword value is available.

There are two non-blocking alternatives to KTL_WAIT. In the first, ktl_read is called once with KTL_NOWAIT and again with KTL_WAITFOR. In this case, the MESSAGE and KEYWORD styles differ:

a keyword value is always available and any latency is the result of system overhead,

a message is not available until it is generated asynchronously by another process.

The situation is analogous to the C library function read applied to a file (in the keyword case) and a socket (in the message case).

In the other non-blocking alternative ktl_read is called with KTL_NOTIFY and specifies a callback function which is called when the keyword value is available.

Finally, the ktl_read mode which will be used most commonly in X displays is the continuous read mode (KTL_EXPRESS_INTEREST). In this mode, a supplied callback is invoked whenever the keyword value changes (KEYWORD style) or whenever a message of the specified type arrives (MESSAGE style).

The calling sequence for ktl_read is:

```
int
ktl_read( handle, flags, event_info, client_data, call_data, context );
KTL_HANDLE *handle;
unsigned int flags;
KTL_EVENT_INFO event_info;
void *client_data;
KTL_CALL_DATA *call_data;
KTL_CONTEXT *context;
```

The handle parameter is the handle containing the information necessary to communicate over the underlying message system.

The flags parameter is a bit-mask of flags which provide extra information about the read operation. Flags in the low two bytes are generic to all services/styles but need not be supported by all services/styles. All flags are relevant to the KEYWORD style. Only the KTL_EXPRESS_INTEREST, KTL_CANCEL_INTEREST and KTL_WAITFOR flags are relevant to the MESSAGE style. Note that the existing cread behavior is achieved using the KTL_WAIT flag.

Possible values (think of them as being processed in this order) are:

KTL_EXPRESS_INTEREST: Enable callback invocation on every occurrence of the asynchronous event defined by event_info. For the KEYWORD style, the first event will occur immediately so that tasks written in this style automatically receive initial values for all keywords in which they have expressed interest. Ignore the remaining generic flags and the `client_data` and `call_data` parameters. Use only the `callback` component of the `context` parameter.[1]

`KTL_CANCEL_INTEREST`: Disable callback invocation on occurrence of asynchronous events defined by `event_info`. Ignore the remaining generic flags and the `client_data`, `call_data` and `context` parameters.

`KTL_WAIT`: Don't return until the keyword defined by `event_info` is available, returning data in the structure whose address is passed in the `call_data` parameter. Meaningless for the `MESSAGE` style. Ignore the remaining generic flags and the `client_data` and `context` parameters.

`KTL_NOWAIT`: Return immediately after the keyword request, if any, has been sent. Any replies necessary to support a subsequent call with `KTL_WAITFOR` will be sent. Meaningless for the `MESSAGE` style. Ignore the remaining generic flags and the `client_data`, `call_data` and `context` parameters.

`KTL_NOREPLY`: Return immediately after the keyword request, if any, has been sent. No replies will be sent and a subsequent call with `KTL_WAITFOR` may block for ever. Meaningless for the `MESSAGE` style. Ignore the remaining generic flags and the `client_data`, `call_data` and `context` parameters.

`KTL_WAITFOR`: Don't return until the keyword or message defined by `event_info` is available, returning data in the structure whose address is passed in the `call_data` parameter. A call with `KTL_NOWAIT` followed by a call with `KTL_WAITFOR` has exactly the same visible effect as a single call with `KTL_WAIT`. Can also be used to wait for completion of transactions initiated using `ktl_write` with `KTL_NOWAIT`. Ignore all other generic flags and the `client_data` and `context` parameters.

`KTL_NOTIFY`: Return as soon as the keyword request, if any, has been sent. Meaningless for the `MESSAGE` style. Ignore the `call_data` parameter, and use the `client_data` and `context` parameters to set up a callback function to be called when the keyword value is available.

The `event_info` parameter defines the asynchronous event for which callback invocation is being enabled or disabled, or the keyword or message which is to be read.[2]

For the `KEYWORD` style it is a `char *` and points to the short (`FITS`) version of the keyword as specified in [**?**, Keck keyword definitions].

For the `MESSAGE` style with `KTL_WAITFOR` it is the unique sequence number returned as the function value of the `ktl_write` call which initiated the transaction.

For the `MESSAGE` style with all other flags (only `KTL_EXPRESS_INTEREST` and `KTL_CANCEL_INTEREST` make sense) it is a message type (`MUSIC` subject for the `MUSIC` service, `DCS` message type for the `DCS` service).

It will be useful if all styles provide a standard "match-all" `event_info`, *e.g.* `*` for the `KEYWORD` style and `-1` for the `MESSAGE` style.

---

[1] `context` can be thought of as a union of a callback pointer and a context structure; thus a callback pointer can be passed as the `context` parameter.

[2] In implementation terms, it is a union passed by value. For all machines currently in use it will be of length 4 bytes.

The `client_data` parameter is an arbitrary piece of data which is supplied as one of the parameters to the callback function.

The `call_data` parameter allows access to the received keyword value or message. For the `KEYWORD` style, it is the address of a `KTL_POLYMORPH` structure. For the `MESSAGE` style, it is the address of a structure containing a service-specific message header and a pointer to the message body. In both cases, `ktl_read` may `malloc` storage for the keyword value / message body, and the caller must free any such storage.

The `context` parameter is a pointer to a context structure. `KTL_EXPRESS_INTEREST` requires only the first component of this structure. The only other flag which requires it is `KTL_NOTIFY`. If `ktl_read` is itself being called from a `KTL` callback function, this should be the callback function's `context` parameter. Otherwise (a common case will be in an `X` callback), a context must be created using the `ktl_context_create` function (§4.3). Refer to §3.11 for a description of the callback function's parameters. Refer to §4 for a description of how context is handled.

The function return value is the unique sub-context sequence number pertaining to this transaction, or zero if no sub-context needs to be allocated. Note that all valid sub-context sequence numbers are greater than zero.

3.4 `ktl_write` — set a keyword or send a message

The `ktl_write` function is the successor to the `KICS` function `cset`. It provides the functionality of `cset` plus extra functionality to support new `KTL` facilities. It is in some respects complementary to `ktl_read`.

Several writing modes are available and are controlled by the `flags` parameter. The simplest case `ktl_write` is called with `KTL_WAIT` and blocks until the specified keyword has been set or completion message has been received.

There are two non-blocking alternatives to `KTL_WAIT`. In the first, `ktl_write` is called with `KTL_NOWAIT` and then `ktl_read` is called with `KTL_WAITFOR`.

In the other non-blocking alternative `ktl_write` is called with `KTL_NOTIFY` and specifies a callback function which is called when the keyword value has been set or message associated with this transaction has been received.

The calling sequence for `ktl_write` is:

```
int
ktl_write( handle, flags, event_info, client_data, call_data, context );
KTL_HANDLE *handle;
unsigned int flags;
KTL_EVENT_INFO event_info;
void *client_data;
KTL_CALL_DATA *call_data;
KTL_CONTEXT *context;
```

The `handle` parameter is the handle containing the information necessary to communicate over the underlying message system.

The `flags` parameter is a bit-mask of flags which provide extra information about the operation. Flags in the low two bytes are generic to all services/styles but need not be supported by all services/styles.

Possible values (think of them as being processed in this order) are:

`KTL_WAIT`: Don't return until the keyword value has been set or the completion message has been received. Ignore the remaining generic flags and the `client_data` and `context` parameters.

`KTL_NOWAIT`: Return immediately the operation has been initiated. Any replies necessary to support a subsequent call with `KTL_WAITFOR` will be sent. Ignore the remaining generic flags and the `client_data` and `context` parameters.

`KTL_NOREPLY`: Return immediately the operation has been initiated. No replies will be sent and a subsequent `ktl_read` call with `KTL_WAITFOR` may block for ever. Ignore the remaining generic flags and the `client_data` and `context` parameters.

`KTL_NOTIFY`: Return as soon as the operation has been initiated, but use the `client_data` and `context` parameters to set up a callback function to be called on completion of the operation.

The `event_info` parameter specifies, for the `KEYWORD` style, the name of the keyword whose value is to be set, and for the `MESSAGE` style, the type of the message to be sent.

The `client_data` parameter is an arbitrary piece of data which is supplied as one of the parameters to the callback function.

The `call_data` parameter is a pointer to the data corresponding to `event_info`. For the `KEYWORD` style, it is a pointer to a `KTL_POLYMORPH` structure. For the `MESSAGE` style, it is a pointer to a message. Information cannot be returned in this structure.

The `context` parameter is a pointer to a context structure. This is used only in association with the `KTL_NOTIFY` flag. If `ktl_write` is itself being called from a `KTL` callback function, it should be the callback function's `context` parameter. Otherwise (a common case will be in an `X` callback), a context must be created using the `ktl_context_create` function (§4.3). Refer to §3.11 for a description of the callback function's parameters. Refer to §4 for a description of how context is handled.

The function return value is the unique sub-context sequence number pertaining to this transaction, or zero if no sub-context needs to be allocated. Note that all valid sub-context sequence numbers are greater than zero. This sequence number can be used to perform multiple operations in parallel without blocking and without using callback functions. Use `ktl_write` with `KTL_NOWAIT` to initiate the transactions, keeping a record of the sequence numbers of active transactions. Use these sequence numbers as the `event_info` parameters to `ktl_read` with `KTL_WAITFOR`.

3.5 `ktl_handle_event` — handle an event

The `ktl_handle_event` function provides the mechanism by which a particular event results in the invocation of one of the callbacks previously specified via `ktl_read` or `ktl_write`.

If the callback was specified with `ktl_read / KTL_EXPRESS_INTEREST` then `ktl_handle_event` creates a new context when the event occurs and passes it to the callback. Otherwise `ktl_handle_event` passes the context that was supplied to `ktl_read / ktl_write` to the completion callback.

The calling sequence for `ktl_handle_event` is:

```
int
ktl_handle_event( handle )
KTL_HANDLE *handle;
```

The `handle` **parameter** is the handle corresponding to the server from which a message has arrived. `ktl_handle_event` reads and handles this message.

The function `ktl_handle_event` is called from the main loop of an application as follows:

```
forever
  begin
    select( [ X socket, KTL socket ], timeout = forever )

    if a message is waiting on the X socket
      event <- XtNextEvent()
      XtDispatchEvent( event )

    if a message is waiting on a KTL device
      ktl_handle_event( handle )
  end
```

Actually it should be possible to use `XtAppAddInput` and the standard `XtAppMainLoop`.

3.6 `ktl_close` — close a connection to a server

`ktl_close` closes a connection to a server and releases all storage allocated by `ktl_open`.

The calling sequence for `ktl_close` is

```
int
ktl_close( handle )
KTL_HANDLE *handle;
```

The `handle` **parameter** is the value returned from the corresponding call to `ktl_open`. On successful completion of the function, it will be invalid.

3.7 ktl_shut — shut down KTL

ktl_init shuts down KTL. It should be called once at the bottom of the application's main program. It exists to perform service-independent shutdown, of which the only current example is the shutdown of the context facilities.

The calling sequence for ktl_shut is

```
int
ktl_shut()
```

3.8 ktl_log — log a message

ktl_log provides a standard way to log informational and error messages not only to standard output and standard error but also to a centralized log file.[3] Various macros hiding the flags and/or status parameters could be provided.

The calling sequence for ktl_log is

```
void
ktl_log( flags, status, fmt, ... )
unsigned int where;
char *fmt;
```

The flags parameter is a bit-mask which indicates the destination(s) of the message, the type of the message and controls other behavior of the function. Possible destination bits could include:

KTL_STDOUT: log to standard output

KTL_STDERR: log to standard error

KTL_STDLOG: log to the standard log file

A given message can therefore go to 0–3 different places. The implementation may involve sending a message to a reporting task.

Possible type bits could include:

KTL_INFO: message is informational

KTL_WARN: message is a warning

KTL_ERROR: message is an error

KTL_DEBUG: message is debugging output

Application-specific bits should also be defined. When and if needed, a function could be provided for setting a task's logging level, for example to disable KTL_DEBUG output. There would be a static logging level which ktl_log would AND with the supplied message type flags in order to decide whether to output the message.

Possible other bits could include:

[3]The interface proposed here is tentative but does combine the good features of several existing systems.

KTL_ERRTXT: message is to replace the string returned by ktl_set_errtxt.

One could define various standard combinations of these bits and give them names. That way, not all users would have to be aware of the individual bits.

The status parameter is a status value which may optionally be used by the error reporting mechanism.

The fmt parameter is a printf format string. It is followed by the parameters implied by the format string's contents. ktl_log will probably call sprintf.

3.9 The KTL_HANDLE structure

The first part of the KTL_HANDLE structure is generic. This is followed by service/style-specific components, accessed indirectly via a pointer. A possible definition of the structure (omitting some low-level details) is:

int valid: Boolean indicating whether the handle is valid (perhaps some magic values are called for as well).

KTL_SERVICE service: enumerated type describing the service from the ktl_open call.

KTL_STYLE style: enumerated type describing the style from the ktl_open call.

unsigned int flags: flags from the ktl_open call.

void *normal_fd: file descriptor on which ktl_handle_event should listen for normal messages.

void *timeout_fd: file descriptor on which ktl_handle_event should listen for timeout messages. Refer to §4.2 and §7.6 for discussion of this.

void *outgoing_list: pointer to list of outgoing connections (can be used in service/style-specific way).

void *interest_list: pointer to asynchronous event interest list (can be used in service/style-specific way but with any luck the event interest list structure will be generic).

KTL_SPECIFIC_HANDLE *specific: pointer to service/style-specific information.

3.10 Function return values

All functions return ints which indicate whether they succeeded or failed: positive or zero for success, negative for failure. Actual error codes need to be defined. Text describing the most recent error is obtained via:

```
char *ktl_get_errtxt();
```

which takes no parameters.

Note that these success/failure conventions are not consistent with the DCS convention that odd numbers indicate success and even numbers indicate failure.

3.11 Callback function parameters

All callback functions have the same calling sequence, regardless of service or whether they are as a result of calls to `ktl_read` or `ktl_write`. The calling sequence is modeled on that of `X` callbacks, with `event_info` taking the place of `Widget` and with an extra parameter to handle context, which is not explicitly supported by `X`. It is important to understand how context is handled; refer to §4.

The calling sequence for callback functions is:

```
int
foo( event_info, client_data, call_data, context )
EVENT_INFO event_info;
void *client_data;
KTL_CALL_DATA *call_data;
KTL_CONTEXT *context;
```

The `event_info` parameter specifies, for the `KEYWORD` style, the name of the keyword corresponding to the event, and for the `MESSAGE` style, the type of the message which has been received.

The `client_data` parameter is the same arbitrary piece of data that was passed to `ktl_read` or `ktl_write`.

The `call_data` parameter contains data corresponding to `event_info`. For the `KEYWORD` style, this is a pointer to a `KTL_POLYMORPH` structure. For the `MESSAGE` style, it is a pointer to a message. `context->status` indicates the success or failure of the operation which resulted in this event.

The `context` parameter is a pointer to a context structure. If the callback is as a result of a call to `ktl_read / KTL_EXPRESS_INTEREST` this structure will have just been allocated. Otherwise it will the same context pointer that was passed to the `ktl_read / ktl_write` call. Refer to §4 for a fuller description.

The function return value is used by `ktl_handle_event` in two ways. Firstly, if a failure status (§3.10) is returned it will be as if `context->state` were returned as `KTL_FINAL_STATE` and the context will be deleted. Secondly, if a response is sent on exit from the callback function, its status component (assuming that the service supports such a concept) will be the function return value.

3.12 Other functions

`ktl_get_errtxt` has been mentioned. There needs to be a corresponding `ktl_set_errtxt` (unless setting the error text is a side-effect of `ktl_log`, as tentatively proposed in §3.8).

`KTL include` file naming conventions need to be generated, as do a set of conventions on how to manage the service/style-specific code. Presumably all non-generic `KTL` functions will ultimately consist of a `switch` on `handle->service`.

4 Handling context

The KTL context facilities keep track of *active commands*. A command becomes active when an event occurs in which interest has been expressed by `ktl_read / KTL_EXPRESS_INTEREST` and at this point a context structure is created. Some commands complete immediately but others use `ktl_read / ktl_write` to interact with other parts of the system and are not deemed complete until all those interactions are themselves complete. The context facilities allow

- Management of timeouts on interactions,
- Tracking of which interactions are still active,
- Mapping of response events back to the callback functions which initiated the interactions.

Context is handled by a self-contained set of KTL functions which can be utilized by all KTL services/styles and which can be called by application functions (all but one or two are generic). A brief description of these functions and their interfaces is given in the following sections.

Refer to §7 for an illustration of how these context functions can be used to implement the MESSAGE style.

4.1 `ktl_context_init` — initialize the context facility

`ktl_context_init` is called by `ktl_init`. It sets defaults, initializes free lists and so on. Note that global variables are used to maintain internal state rather than forcing programmers to carry around handle variables. This is done to avoid the need to communicate such handles to callback functions. Under systems such as VxWorks, which have a single shared address space, special care will have to be taken to maintain handles which are unique to tasks (one can either use task variables or else can use the task id to index into a table).

It is unlikely that application functions would want to call this function.

The calling sequence for `ktl_context_init` is:

```
int
ktl_context_init( context_specific, subcontext_specific )
int context_specific;
int subcontext_specific;
```

The `context_specific` **parameter** is the length in bytes of the service/style-specific part of the `KTL_CONTEXT` structure. If multiple services/styles are to be used, this must be the maximum such length since all context structures are allocated off the same free list.

The `subcontext_specific` **parameter** is the equivalent length for the `KTL_SUBCONTEXT` structure.

4.2 `ktl_context_open` — open up the context facility for a service

`ktl_context_open` is called from `ktl_open`. It sets up the context facility for a given service. The only thing that I can currently see it doing is creating a pipe device to receive reschedule and timeout messages. There has to be one such device per service and its `fd` will be written to a component of the `handle` structure. Refer to §3.9 for a description of the handle structure.

It is unlikely that application functions would want to call this function.

The calling sequence for `ktl_context_open` is:

```
int
ktl_context_open( handle )
KTL_HANDLE *handle;
```

The `handle` **parameter** is the pointer to the service-specific handle structure which is returned by `ktl_open`.

4.3 `ktl_context_create` — create and initialize a context structure

`ktl_context_create` is called by `ktl_handle_event` when it handles an event in which interest has been expressed by `ktl_read / KTL_EXPRESS_INTEREST`. It creates a new context structure and initializes its components, returning a pointer to the allocated structure.

One of the structure components is a unique sequence number which is `-1` for the first allocated context, `-2` for the next, and so on. This sequence number is placed in a special component of all messages sent by `ktl_context_nextentry` and is returned by all responses to such messages, thus allowing the originating context to be determined.

This function will be called from application functions which are not `KTL` callbacks (of which `X` callbacks will be the most common example).

The calling sequence for `ktl_context_create` is:

```
int
ktl_context_create( handle, callback, client_data, context )
KTL_HANDLE *handle;
int (*callback)();
void *client_data;
KTL_CONTEXT **context;
```

The `handle` **parameter** is the pointer to the service-specific handle structure which is returned by `ktl_open`.

The `callback` **parameter** is a pointer to the callback function which is associated with the command on whose behalf the context is being created.

The `client_data` **parameter** is an arbitrary piece of data which is supplied as one of the parameters of callback functions associated with this context.

The `context` **parameter** is the address of a variable to receive a pointer to a structure which contains the information necessary to maintain the context. This is a public structure and is described in §4.14.

4.4 `ktl_context_lookup` — look up the context structure corresponding to an event

`ktl_context_lookup` is called by `ktl_handle_event` when it handles an event which has occurred as a result of `ktl_context_nextentry`. It searches for an active context structure whose sequence number component matches the sequence number from the message which caused the event.

This function will not normally be called from application functions.

The calling sequence for `ktl_context_lookup` is:

```
int
ktl_context_lookup( sequence, context )
int sequence;
KTL_CONTEXT **context;
```

The `sequence` parameter is an integer which uniquely identifies an active context. It will normally have been taken from a special sequence number component of the message that caused the event.

The `context` parameter is the address of a variable to receive the context pointer.

4.5 `ktl_context_postproc` — perform post-processing after callback invocation

`ktl_context_postproc` is called by `ktl_handle_event` immediately after it has called the programmer-supplied callback function. It checks that the callback has not invalidated the context structure and sets the context state to `KTL_FINAL_STATE` if it is still `KTL_INITIAL_STATE` or if the callback function returned bad (negative) status.

This function will not normally be called from application functions.

The calling sequence for `ktl_context_postproc` is:

```
int
ktl_context_postproc( status, context )
int status;
KTL_CONTEXT *context;
```

The `status` parameter is the status (function value) returned by the programmer-supplied callback function.

The `context` parameter is a pointer to the current context structure.

4.6 `ktl_context_respond` — respond to initiator of command

This function is not generic.

`ktl_context_respond` is called by `ktl_handle_event` immediately after it has called `ktl_context_postproc`. It decides whether a response is to be sent (this decision has to be service/style-specific) and, if so, sends it. Note that its `handle` parameter is always that from the `ktl_context_create` call (the `create_handle`), which is not necessarily the one corresponding to the event which is currently being handled (the `event_handle`).

This function will not normally be called from application functions.

The calling sequence for `ktl_context_respond` is:

```
int
ktl_context_respond( handle, context )
KTL_HANDLE *handle;
KTL_CONTEXT *context;
```

The `handle` parameter is the handle pointer which was passed to the `ktl_context_create` call which created the current context.

The `context` parameter is a pointer to the current context structure.

4.7 `ktl_context_nextentry` — arrange for next callback entry

This function may not be generic.

`ktl_context_nextentry` is called by `ktl_handle_event` just before it exits. It handles immediate reschedules, reschedules on expiry of specified intervals, and reschedules when interactions with other tasks time out. Note that its `handle` parameter is always that from the `ktl_context_create` call (the `create_handle`), which is not necessarily the one corresponding to the event which is currently being handled (the `event_handle`).

This function will not normally be called from application functions.

The calling sequence for `ktl_context_nextentry` is:

```
int
ktl_context_nextentry( handle, context )
KTL_HANDLE *handle;
KTL_CONTEXT *context;
```

The `handle` parameter is the handle pointer which was passed to the `ktl_context_create` call which created the current context.

The `context` parameter is a pointer to the current context.

4.8 `ktl_context_delete` — delete a context structure

`ktl_context_delete` is called by `ktl_handle_event` when it determines that a command is complete. When a context is deleted it should not have any associated sub-contexts; if it has, then a warning message will be output for each one using `ktl_log` and it will be deleted before the parent context is deleted.

This function will not normally be called from application functions.

The calling sequence for `ktl_context_delete` is:

```
int
ktl_context_delete( context )
KTL_CONTEXT *context;
```

The `context` **parameter** is a pointer to the context to be deleted.

4.9 `ktl_context_close` — close down the context facility for a service

`ktl_context_close` is called by `ktl_close`. It deletes the pipe device that was created by `ktl_context_open`.

This function will not normally be called from application functions.

The calling sequence for `ktl_context_close` is:

```
int
ktl_context_close( handle )
```

The `handle` **parameter** is the pointer to the service-specific handle structure which is returned by `ktl_open`.

4.10 `ktl_context_shut` — shut down the context facility

`ktl_context_shut` is called by `ktl_shut`. When it is called, there should be no active contexts; if there are, then a warning message will be output for each one and it will be deleted before the facility is closed.

This function will not normally be called from application functions.

The calling sequence for `ktl_context_shut` is:

```
int
ktl_context_shut()
```

4.11 `ktl_subcontext_create` — create and initialize a sub-context structure

`ktl_subcontext_create` is called by `ktl_write` when it initiates interactions with lower-level tasks. It creates a new sub-context structure and initializes its components, returning a pointer to the allocated structure.

One of the structure components is a unique sequence number which is `1` for the first allocated sub-context, `2` for the next, and so on. This sequence number is placed in a special component of all messages sent by `ktl_write` and is returned by all responses to such messages, thus allowing the originating sub-context and context to be determined (the sub-context structure points to its parent context structure).

This function will not normally be called from application functions.

The calling sequence for `ktl_subcontext_create` is:

```
int
ktl_subcontext_create( context, client_data, subcontext )
KTL_CONTEXT *context;
void *client_data;
KTL_SUBCONTEXT **subcontext;
```

The `context` **parameter** is a pointer to the parent context structure.

The `client_data` **parameter** is an arbitrary piece of data which is supplied as one of the parameters of callback functions associated with this sub-context.

The `subcontext` **parameter** is the address of a variable to receive a pointer to a structure which contains the sub-context. This is a public structure and is described in §4.15.

4.12 `ktl_subcontext_lookup` — look up the sub-context structure corresponding to an event

`ktl_subcontext_lookup` is called by `ktl_handle_event` when it handles an event which has occurred as a result of `ktl_write`. It searches for an active sub-context structure whose sequence number component matches the sequence number from the message which caused the event.

This function will not normally be called from application functions.

The calling sequence for `ktl_subcontext_lookup` is:

```
int
ktl_context_lookup( sequence, subcontext )
int sequence;
KTL_SUBCONTEXT **subcontext;
```

The `sequence` **parameter** is an integer which uniquely identifies an active sub-context. It will normally have been taken from a special sequence number component of the message that caused the event.

The `subcontext` **parameter** is the address of a variable to receive the subcontext pointer.

4.13 `ktl_subcontext_delete` — delete a sub-context structure

`ktl_subcontext_delete` is called by `ktl_handle_event` when it determines that an interaction with a low-level task is complete.

This function will not normally be called from application functions.

The calling sequence for `ktl_subcontext_delete` is:

```
int
ktl_subcontext_delete( subcontext )
KTL_SUBCONTEXT *subcontext;
```

The `subcontext` **parameter** is a pointer to the sub-context to be deleted.

4.14 The `KTL_CONTEXT` structure

The first part of the `KTL_CONTEXT` structure is generic. This is followed by service/style-specific components accessed indirectly via a pointer (the advantage of using a pointer is that the primary structure is independent of service/style). A possible definition of the structure (omitting some low-level details) is:

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

> *This has to be the first component of the structure, or else a* `union` *has to be used. Refer to the discussion of the* `KTL_EXPRESS_INTEREST` *flag in* §*3.3.*

`int valid:` Boolean indicating whether the context structure is valid (perhaps some magic values are called for as well).

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

`void *source_task:` service/style-specific means of identifying the source of the message which resulted in the creation of this context.

`int source_sequence_number:` sequence number from message which resulted in the creation of this context (must be used in all responses).

`void *client_data:` arbitrary piece of data supplied to the `ktl_read / KTL_EXPRESS_INTEREST` function which resulted in this context being created.

`KTL_HANDLE *create_handle:` pointer to handle which created this context structure.

`KTL_HANDLE *event_handle:` pointer to handle on which current event arrived.

`KTL_SUBCONTEXT *subcontext:` pointer to sub-context corresponding to current event, `NULL` if none.

`int status`: status corresponding to current event.

`KTL_COMMAND_STATE state`: enumerated type indicating the command's state: `KTL_INITIAL_STATE`, `KTL_RESCHEDULE_STATE`, `KTL_DELAY_STATE`, `KTL_MESSAGE_STATE`, `KTL_FINAL_STATE`, `KTL_ERROR_STATE`.

`int timeout`: timeout period in milliseconds, used when the command state is `KTL_DELAY_STATE` or `KTL_MESSAGE_STATE`.

`int local_state`: integer guaranteed to be `0` on a command's initial entry, `-1` on error and otherwise altered only in the callback function.

`void *user_area`: pointer guaranteed to be `NULL` on a command's initial entry and altered only in the callback function.

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

`int inhibit_reply`: whether to inhibit sending a service/style-specific default reply message.

`int inhibit_free`: whether to inhibit `free`ing of `response_msg`.

`int reschedule_on_error`: whether to give control back to the callback if `ktl_handle_event` encounters an error.

`int delete_on_error`: whether to delete the context if `ktl_handle_event` encounters an error.

`KTL_SPECIFIC_CONTEXT *specific`: pointer to service/style-specific context information (which will be allocated immediately following this pointer). `NULL` if there is no such information.

4.15 The `KTL_SUBCONTEXT` structure

The first part of the `KTL_SUBCONTEXT` structure is generic. This is followed by service/style-specific components accessed indirectly via a pointer (the advantage of using a pointer is that the primary structure is independent of service/style). A possible definition of the structure (omitting some low-level details) is:

`int valid`: Boolean indicating whether the sub-context structure is valid (perhaps some magic values are called for as well).

`int sequence_number`: unique sub-command sequence number (incremented by 1 for each sub-context allocated).

`KTL_CONTEXT *context`: pointer to parent context.

`KTL_SPECIFIC_SUBCONTEXT *specific`: pointer to service/style-specific sub-context information (which will be allocated immediately following this pointer). `NULL` if there is no such information.

5 Implementing KEYWORD-style KTL for the MUSIC service

5.1 ktl_read/KTL_WAIT

Call `cread`.

5.2 ktl_read/KTL_EXPRESS_INTEREST

The `ktl_read` function adds entries to a list of `interest` structures. Each `interest` structure contains:

- keyword name
- event type
- callback address
- client data
- sequence number

As we will see in §5.4 below, this list is read by `ktl_handle_event` to determine which callback function is to be invoked whenever an event is received in the application main loop.

The `ktl_read` function creates an `interest` entry, consisting of the keyword, callback, and client data which were passed as arguments, and adds it with `event_type = ON_CHANGE` to the list of `interest` entries. It then sends a `DESIRED_MESSAGES` message to traffic specifying interest in a broadcast read from a table (*e.g.*, the infoman broadcast `INFO_CHANGE_MADE`). It then call `cread` and invokes the callback with the initial value.

5.3 ktl_write

As described in §3.4 above, `KTL` introduces a successor to the `KICS` function `cset` called `ktl_write`. If the `KTL_NOTIFY` flag bit is not set then the old `cset` is called with `client_data` and `context` omitted and the other parameters in the appropriate order but otherwise unchanged.

If the `KTL_NOTIFY` flag bit is set then the supplied callback is added to the interest table with `event_type` set to `ON_COMPLETION` and `sequence_number` set to a unique number. A subprocess is forked. The subprocess is supplied with the `our_msg_id` field from the `MUSIC_handle` and the unique sequence number. The subprocess makes a new connection to `traffic` and calls `cset` with the `KTL_WAIT` option. When `cset` returns, the subprocess sends a `CSET_COMPLETED` `MUSIC` message to `our_msg_id` which contains the keyword, sequence number, and completion status. The context pointer is saved in the interest structure before the fork.

5.4 ktl_handle_event

The `ktl_handle_event` function handles broadcast events indicating asynchronous change and `CSET_COMPLETED` messages generated by `ktl_write`. Broadcast messages are handled by looking up a function to be called from a table keyed by broadcast subject. The function retrieved from the broadcast table extracts the keyword and keyword value from the `MUSIC` message. The keyword is used to find the interest record in the interest table. ktl_handle_event then passes the keyword value to the callback. In addition a newly created context is passed to the callback.

The CSET_COMPLETED message contains the keyword, sequence number, and completion status. The keyword and sequence number are used to find the interest record in the interest table. The completion status and the saved context are passed to the callback.

6 Implementing KTL for the RPC service

The RPC service may only support the KEYWORD style?

TBD/DEB

7 Implementing KTL for the MESSAGE style

This section concentrates on how the DCS service and MESSAGE style can be implemented (this combination results in a system which has the same functionality as the CCX system currently running under VMS and VxWorks within the DCS system).

7.1 Main program

This main program illustrates use of just the DCS service without any X events. ktl_handle_event will block until a message arrives, but this is OK since there is only one source of messages:

```
#define FLAGS 0

ktl_init ( &argc, argv )
ktl_open( "DCS", "MESSAGE", NULL, FLAGS, &handle )

for (i = 0; i < NUM_COMMANDS; i++)
    ktl_read( handle, KTL_EXPRESS_INTEREST, msg_type[i], NULL, NULL, callback[i] )

forever
    ktl_handle_event( handle )
```

7.2 ktl_init

ktl_init initializes the DCS message system and the KTL context facilities:

```
ktl_init( argc, argv )
{
    process arguments to obtain this_host_name and this_task_name
    msg_system_init( this_host_name )
    msg_init( this_task_name )
    ktl_context_init()
}
```

7.3 `ktl_open`

`ktl_open` allocates a `KTL_HANDLE` structure, creates (by default) this task's input device, initializes the list of known outgoing connections (outgoing connections are created on the fly within `ktl_write`), initializes the list of events in which interest has been expressed, and initializes the context facilities for this service:

```
ktl_open( service, style, flags, handle )
{
    *handle = allocate handle structure
    initialize (*handle)->outgoing_list
    initialize (*handle)->interest_list

    msg_open( msg_task_name( NULL ), CREATE, &(*handle)->this_task )

    ktl_context_open( *handle )
}
```

7.4 `ktl_read`

`ktl_read` for the `MESSAGE` style is meaningful only when called with the `KTL_EXPRESS_INTEREST` or `KTL_CANCEL_INTEREST` flags:

`KTL_EXPRESS_INTEREST`: add an entry to the list of events in which interest has been expressed. Each entry consists of the event info, the client data and the callback address. An entry for an existing event info overrides the earlier entry:

`KTL_CANCEL_INTEREST`: remove an entry from the list of events in which interest has been expressed. It simply searches for an entry with a matching event info and deletes it by some means.

```
ktl_read( handle, flags, event_info, client_data, call_data, context )
{
    scan handle->interest_list looking for event_info
    if flags & KTL_EXPRESS_INTEREST
        if found
            replace entry
        else
            create new entry
    else if flags & KTL_CANCEL_INTEREST
        if found
            delete entry
}
```

7.5 ktl_write

`ktl_write` allocates a sub-context, copies its sequence number to the corresponding message component, checks that a connection is open to the destination task, and sends the message (assuming that neither of the `KTL_WAIT` and `KTL_WAITFOR` flags is set):

```
ktl_write( handle, flags, event_info, client_data, call_data, context )
{
    ktl_subcontext_create( context, client_data, &subcontext )
    call_data->sequence_number = subcontext->sequence_number

    if no entry for call_data->destination in handle->outgoing_list
        open connection to call_data->destination
        add entry to handle->outgoing list
    send_message( call_data->destination, call_data )
}
```

7.6 ktl_handle_event

`ktl_handle_event` reads its message, searches the interest list for an entry that matches the incoming event info. If it finds an entry, it creates a new context. If not, it looks up the incoming message's sequence number field in the active context list (if it's negative) or the active sub-context list (if it's positive). Zero event info is an error. This gives the context. Next it invokes the callback function, calls the generic context post-processing function, handles sending replies back to the caller, calls the generic context next-entry function, and, finally, handles deletion of sub-context and context.

Note that reading the message may require doing a `select` on both the normal and the timeout file ids. There have to be two file descriptors when normal messages arrive over an Internet socket.

```
ktl_handle_event( handle )
{
    receive_message( handle, &call_data )
    event_info = call_data->msg_type

    if entry = handle->interest_list entry matching event_info
        ktl_context_create( handle, entry->callback, entry->client_data, &context )
        subcontext = NULL
        client_data = context->client_data

    else if call_data->sequence_number < 0
        ktl_context_lookup( sequence, &context )
        subcontext = NULL
        client_data = context->client_data

    else if call_data->sequence_number > 0
        ktl_subcontext_lookup( sequence, &subcontext )
```

```
        context = subcontext->context
        client_data = subcontext->client_data
    else
        error

    context->status = call_data->status
    status = context->callback( event_info, client_data, call_data, context )

    ktl_context_postproc( status, context )
    ktl_context_respond( context->create_handle, context )
    ktl_context_nextentry( context->create_handle, context )

    if subcontext != NULL and call_data->flags & FINAL
        ktl_subcontext_delete( subcontext )
    if context->state == KTL_FINAL_STATE
        ktl_context_delete( context )
}
```

7.7 ktl_close

Tidy up.

7.8 ktl_shut

Tidy up.

8 Outstanding work

Bear in mind efficient support: for KTL_NOTIFY in the KEYWORD style. The proposed implementation is inefficient but it is sufficient for an initial implementation. A more efficient approach is to propagate the sequence number down through the various tasks and return it in the response. This approach is complicated by the fact that MUSIC headers are not propagated through internal layers on the crate software. This might be eased by using the negative indexing trick.

Define conventions on reserved keyword names: such as ERROR to express interest in error events.

Ensure that KTL can reconnect: when server systems reboot.

Finalize service and style names:

Resolve message representation issues: Current MUSIC applications never explicitly use a message structure; all access is via the mget* and mput* functions. If one used XDR with MUSIC, the message body pointer in call_data could be a pointer to an XDR buffer which is handed to MUSIC in its entirety via a single call to mput. This would not fit in easily with DCS messages, where the header and body are always a contiguous structure.

Review `call_data` parameters: when writing a message using `ktl_write` it is natural to pass a pointer to the entire message, including the header. When reading a message using `ktl_read` this is unsatisfactory since there is no indication of the size of the user's buffer. The proposed solution is close in philosophy to that of the `KTL_POLYMORPH` union but is messy in that it represents messages differently in the two different cases. One could simply pass the address of a pointer to `ktl_read` but that is not consistent with the current use of `KTL_POLYMORPH`.

Try to use a generic interest list:

9 Changes to this document

Version 1 was the original version written by Al Conrad and describing the `CSET++` extensions to `FIORD` for use in writing user interfaces. It was distributed only to David Bruce of the `LWS` project.

Version 2 was also written by Al Conrad and was the first version to describe the `KTL` (at that time known as `KCX`) functions. It was distributed only within `CARA`.

Version 3 was upgraded by William Lupton and extended the descriptions, adding information on maintenance of context. It was distributed only within `CARA`.

Version 4 was a further upgrade by William Lupton. The `KCX` prefix was changed to `KTL`, the concept of "style" was introduced, use of context was made mandatory, and several omissions were rectified.

Version 5 was yet another upgrade by William Lupton, following internal discussion by the CARA software group and various thought experiments on how `KTL` functions might be used for various applications.

Version 6 was a revision by Al Conrad. Two figures were added and the prose describing `ktl_read` was replace.

Version 7 was yet another revision by William Lupton. This was the first version since version 1 which was distributed outside CARA.

References

[1] Al Conrad. Cset/cread. HIRES/LRIS CDR Document number 2.2A, Lick Observatory, January 1991.

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

[3] William Lupton. Ccx (command context) system manual. Keck Software Document KSD 6, W. M. Keck Observatory, December 1991.

[4] William Lupton. Message system manual. Keck Software Document KSD 5, W. M. Keck Observatory, November 1991.

A KTL Identifier Names

The following table enumerates all of KTL identifier names.

Function Names

`ktl_close`
`ktl_context_init`
`ktl_context_open`
`ktl_context_create`
`ktl_context_lookup`
`ktl_context_postproc`
`ktl_context_nextentry`
`ktl_context_delete`
`ktl_context_close`
`ktl_context_shut`
`ktl_get_errtxt` (same as `KICS get_errtxt`)
`ktl_handle_event`
`ktl_init`
`ktl_log`
`ktl_open`
`ktl_read`
`ktl_shut`
`ktl_subcontext_create`
`ktl_subcontext_lookup`
`ktl_subcontext_delete`
`ktl_write`

Types

`KTL_CONTEXT`
`KTL_CALL_DATA`
`KTL_EVENT_INFO`
`KTL_HANDLE`
`KTL_POLYMORPH`
`KTL_SERVICE`
`KTL_STYLE`
`KTL_SUBCONTEXT`

Flag bits

`KTL_CANCEL_INTEREST`
`KTL_EXPRESS_INTEREST`
`KTL_NOTIFY`
`KTL_NOREPLY`
`KTL_NOWAIT`
`KTL_WAIT`
`KTL_WAITFOR`