```
queue_delete          ( qid )
queue_ident           ( name, nid, qid )
queue_send            ( qid, msg_buff, msg_length )
queue_jump            ( qid, msg_buff, msg_length )
queue_broadcast       ( qid, msg_buff, msg_length, count )
queue_receive         ( qid, msg_buff, buff_length, options, time_out,
                        msg_length )
queue_flush           ( qid, count )
queue_info            ( qid, max_buff, length, options, messages_waiting,
                        tasks_waiting )
```

Event Operations

```
event_send            ( tid, event )
event_receive         ( event, options, time_out, event_received )
```

Exception Operations

```
exception_catch       ( bit_number, new_xsr, new_mode, old_xsr, old_mode)
exception_raise       ( tid, exception )
exception_return      ( )
```

Clock Operations

```
clock_set             ( clock )
clock_get             ( clock )
clock_tick            (  )
```

Timer Operations

```
timer_wake_after      ( ticks )
timer_wake_when       ( clock )
timer_event_after     ( ticks, event, tmid )
timer_event_when      ( clock, event, tmid )
timer_event_every     ( ticks, event, tmid )
timer_cancel          ( tmid )
```

Interrupt Operations

```
int_enter             ( )
int_return            ( )
```

Miscellaneous Operations

```
int_to_ext            ( int_addr, port, ext_addr )
ext_to_int            ( ext_addr, port, int_addr )
```

```
#ifndef ORKID_H
#define ORKID_H 1
/*
```

D. ORKID: C LANGUAGE BINDING

This file contains the C language binding standard for VITA's "Open Real-time Kernel Interface Definition", henceforth called **ORKID.** The file is in the format of a C language header file, and is intended to be a common starting point for system developers wishing to produce an **ORKID** compliant kernel.

The **ORKID** C language binding consists of four sections, containing type specifications, function declarations, completion status codes and special symbol codes. The character sequence ??? has been used throughout wherever the coding is implementation dependent.

Of the four sections in this standard, only the function declarations are completely defined. In the other sections, only the type names and constant symbols are defined by this standard - all types and values are implementation dependent.

Both ANSI C and non-ANSI C have been used for this header file. Defining the symbol __ANSI__ will cause the ANSI versions to be used, otherwise the non-ANSI versions will be used. Full prototyping has been employed for the ANSI function declarations.

```
*/
```

```
/*
```

ORKID TYPE SPECIFICATIONS

This section of the **ORKID** C language binding contains typedef definitions for the types used in operation arguments in the main **ORKID** standard. The names are the same as those in the **ORKID** standard. Only the names, and in clock_buff the order of the structure members, are defined by this standard. The actual types are implementation dependent.

```
*/

typedef unsigned int prio ;
typedef unsigned int word ;
typedef unsigned int bit_field ;
typedef ??? task_id ;
typedef ??? node_id ;
typedef ??? region_id ;
typedef ??? pool_id ;
typedef ??? sema_id ;
typedef ??? queue_id ;
typedef ??? timer_id ;
typedef ??? cb_year ;
typedef ??? cb_month ;
typedef ??? cb_day ;
typedef ??? cb_hours ;
typedef ??? cb_minutes ;
typedef ??? cb_seconds ;
typedef ??? cb_ticks ;
typedef ??? cb_time_zone ;
typedef ??? clock_buff ;
```

/*

ORKID OPERATION DECLARATIONS

This section of the **ORKID** C language binding contains function declarations for all the operations defined in the main **ORKID** standard, and is subdivided according to the subsections in this standard.

Each subdivision contains a list of function declarations and a list of symbol definitions. The function names have been kept to six characters for the sake of linker compatibility. Of these six characters, the first two are always 'OK', and the third designates the **ORKID** object type on which the operation works. The symbol definitions link the full names of the operations given in the **ORKID** standard (in lower case) to the appropriate abbreviation.

The lists of function declarations are split in two. If the symbol __ANSI__ has been defined, then all the functions are declared to the ANSI C standard using full prototyping, with parameter names also included. This latter is not necessary, but not illegal. It shows the correspondence between arguments in this and the main **ORKID** standard, the names being identical. If the symbol __ANSI__ has not been defined, then the functions are declared without prototyping.

The correspondence between the C types and arguments and those defined in the **ORKID** standard are mostly obvious. However, the following comments concerning task_start/restart and exception_catch are perhaps necessary.

A task start address is translated into a function with one argument -a pointer to anything. The task's startup arguments are given as a pointer to anything and a length. The actual arguments will be contained in a programmer defined data type, a copy of which will be passed to the new task. The following is an example of a declaration of a task's main program and a call to start that task (the necessary task creation call is not included):

```
typedef struct { int arg1, arg2, arg3 } argblock ; /*can contain
argblock *argp ;                                    anything*/


void taskmain( argblock *taskargs, int arg_size ) { ... } ;      /*main
task program*/


status = oktsta( tid, taskmain, *argp, size_of( argblock ) ) ;

/*start the task*/
```

An XSR address also becomes a function with one argument - this time a bitfield. The previous XSR address output parameter becomes a pointer to such a function. The following is an example of the declaration of an XSR and a call to exception_catch to set it up:

```
void taskxsr( bit_field exception_caught ) { ... } ;  /*XSR
declaration*/

void (*oldxsr)() ;


status = okxcat( taskxsr, NOXSR, oldxsr ) ;  /*set up taskxsr as XSR*/
                                             with NOXSR mode parameter
*/
```