XILINX
ALL PROGRAMMABLE

UG646 June 4, 2014

Xilkernel (v6.1)

Overview

Xilkernel is a small, robust, and modular kernel. It is a free software library that you receive with the Xilinx® Software Development Kit (SDK). Xilkernel:

- Allows a high degree of customization, letting you tailor the kernel to an optimal level both in terms of size and functionality.
- Supports the core features required in a lightweight embedded kernel, with a POSIX API.
- Works on MicroBlaze™ processors.

Xilkernel IPC services can be used to implement higher level services (such as networking, video, and audio) and subsequently run applications using these services.

Why Use a Kernel?

The following are a few of the deciding factors that can influence your choice of using a kernel as the software platform for your next application project:

- Typical embedded control applications comprise various *tasks* that need to be performed in a particular sequence or schedule. As the number of control tasks involved grows, it becomes difficult to organize the sub-tasks manually, and to time-share the required work. The responsiveness and the capability of such an application decreases dramatically when the complexity is increased.
- Breaking down tasks as individual applications and implementing them on an operating system (OS) is much more intuitive.
- A kernel enables the you to write code at an abstract level, instead of at a small, micro-controller-level standalone code.
- Many common and legacy applications rely on OS services such as file systems, time management, and so forth. Xilkernel is a thin library that provides these essential services. Porting or using common and open source libraries (such as graphics or network protocols) might require some form of these OS services also.

Key Features

Xilkernel includes the following key features:

- High scalability into a given system through the inclusion or exclusion of functionality as required.
- Complete kernel configuration and deployment within minutes from inside of SDK.
- Robustness of the kernel: system calls protected by parameter validity checks and proper return of POSIX error codes.
- A POSIX API targeting embedded kernels, win core kernel features such as:
  - Threads with round-robin or strict priority scheduling.
  - Synchronization services: semaphores and mutex locks.
  - IPC services: message queues and shared memory.
  - Dynamic buffer pool memory allocation.
  - Software timers.
  - User-level interrupt handling.
- Static thread creation that startup with the kernel.

© 2014 Xilinx, Inc. XILINX, the Xilinx logo, Virtex, Spartan, ISE, SDK, Vivado, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. All other trademarks are the property of their respective owners.

- System call interface to the kernel.
- Exception handling for the MicroBlaze processor.
- Memory protection using MicroBlaze Memory Management (Protection) Unit (MMU) features when available.

Xilkernel Organization

The kernel is highly modular in design. You can select and customize the kernel modules that are needed for the application at hand. Customizing the kernel is discussed in detail in "Kernel Customization," page 43[1]. Figure 1 shows the various modules of Xilkernel:

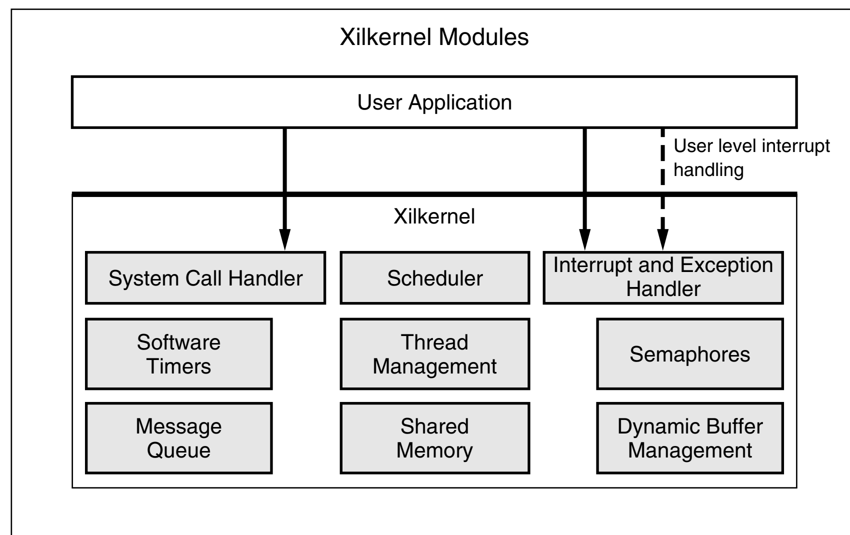

Figure 1: **Xilkernel Modules**

Building Xilkernel Applications

Xilkernel is organized in the form of a library of kernel functions. This leads to a simple model of kernel linkage. To build Xilkernel, you must include Xilkernel in your software platform, configure it appropriately, and run Libgen to generate the Xilkernel library. Your application sources can be edited and developed separately. After you have developed your application, you must link with the Xilkernel library, thus pulling in all the kernel functionality to build the final kernel image. The Xilkernel library is generated as `libxilkernel.a`. Figure 2, page 3 shows this development flow.

Internally, Xilkernel also supports the much more powerful, traditional OS-like method of linkage and separate executables. Conventional operating systems have the kernel image as a separate file and each application that executes on the kernel as a separate file. However, Xilinx recommends that you use the more simple and more elegant library linkage mode. This mode provides maximum ease of use. It is also the preferred mode for debugging, downloading, and bootloading. The separate executable mode is required only by those who have advanced requirements in the form of separate executables. The separate executable mode and its caveats are documented in "Deprecated Features," page 51.

The following are the steps for the kernel linkage mode of application development:

1. Application source C files should include the file `xmk.h` as the first file among others. For example, defining the `includexmk.h` flag makes available certain definitions and declarations from the GNU include files that are required by Xilkernel and applications.

---

1. Some of these features might not be fully supported in a given release of Xilkernel.

2. Your application software project links with the library `libxil.a`. This library contains the actual kernel functions generated. Your application links with this and forms the final kernel and application image.
3. Xilkernel is responsible for all first level interrupt and exception handling on both the MicroBlaze and PowerPC processors. Therefore, you should not directly attempt to use any of the methods of handling interrupts documented for standalone programs. Instead refer to the section on interrupt handling for how to handle user level interrupts and exceptions.
4. You can control the memory map of the kernel by using the linker script feature of the final software application project that links with the kernel. Automatic linker script generation helps you here.
5. Your application must provide a `main( )` which is the starting point of execution for your kernel image. Inside your `main( )`, you can do any initialization and setup that you need to do. The kernel remains unstarted and dormant. At the point where your application setup is complete and you want the kernel to start, you must invoke `xilkernel_main( )` that starts off the kernel, enables interrupts, and transfers control to your application processes, as configured. Some system-level features may need to be enabled before invoking `xilkernel_main( )`. These are typically machine-state features such as cache enablement, hardware exception enablement which must be "always ON" even when context switching from application to application. Make sure that you setup such system state before invoking `xilkernel_main( )`. Also, you must not arbitrarily modify such system-state in your application threads. If a context switch occurs when the system state is modified, it could lead to subsequent threads executing without that state being enabled; consequently, you must lock out context switches and interrupts before you modify such a state.

Note: Your linker script must be aware of the requirements for the kernel.

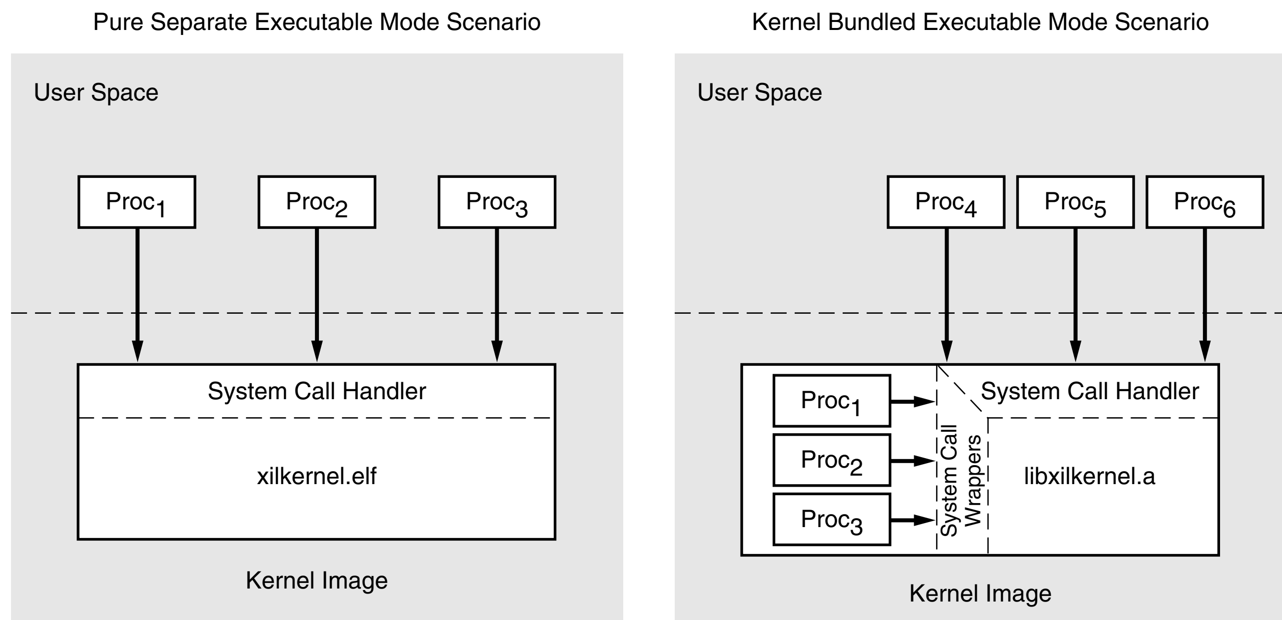

Figure 2: **Xilkernel Development Flow**

Xilkernel Process Model

The units of execution within Xilkernel are called *process contexts*. Scheduling is done at the process context level. There is no concept of thread groups combining to form, what is conventionally called a process. Instead, all the threads are peers and compete equally for resources. The POSIX threads API is the primary user-visible interface to these process contexts. There are a few other useful additional interfaces provided, that are not a part of POSIX. The interfaces allow creating, destroying, and manipulating created application threads. The actual interfaces are described in detail in "Xilkernel API," page 6. Threads are manipulated with thread identifiers. The underlying process context is identified with a process identifier *pid_t*.

Xilkernel Scheduling Model

Xilkernel supports either priority-driven, preemptive scheduling with time slicing (`SCHED_PRIO`) or simple round-robin scheduling (`SCHED_RR`). This is a global scheduling policy and cannot be changed on a per-thread basis. This must be configured statically at kernel generation time.

In `SCHED_RR`, there is a single ready queue and each process context executes for a configured time slice before yielding execution to the next process context in the queue.

In `SCHED_PRIO` there are as many ready queues as there are priority levels. Priority 0 is the highest priority in the system and higher values mean lower priority.

As shown in the following figure, the process that is at the head of the highest priority ready queue is always scheduled to execute next. Within the same priority level, scheduling is round-robin and time-sliced. If a ready queue level is empty, it is skipped and the next ready queue level examined for schedulable processes. Blocked processes are off their ready queues and in their appropriate wait queues. The number of priority levels can be configured for `SCHED_PRIO`.

For both the scheduling models, the length of the ready queue can also be configured. If there are wait queues inside the kernel (in semaphores, message queues), they are configured as priority queues if scheduling mode is `SCHED_PRIO`. Otherwise, they are configured as simple first-in-first-out (FIFO) queues.

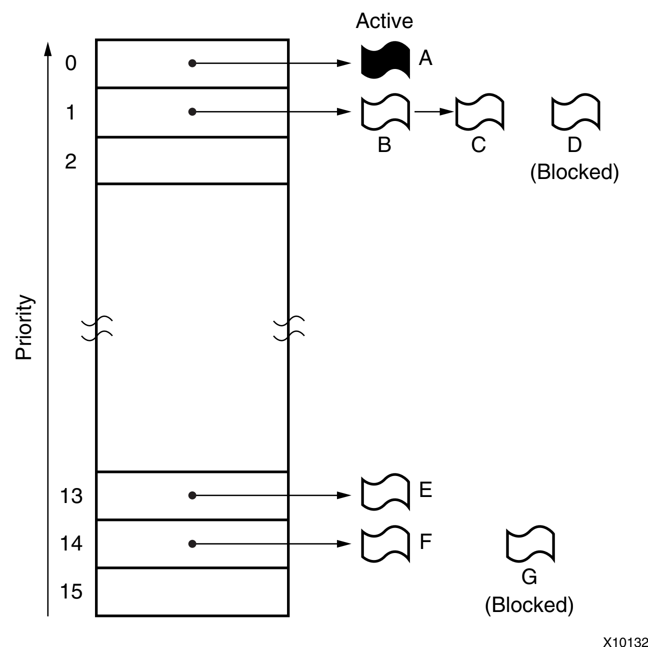

Figure 3: **Priority-Driven Scheduling**

Each process context is in any of the following six states:

- `PROC_NEW` - A newly created process.
- `PROC_READY` - A process ready to execute.
- `PROC_RUN` - A process that is running.
- `PROC_WAIT` - A process that is blocked on a resource.
- `PROC_DELAY` - A process that is waiting for a timeout.
- `PROC_TIMED_WAIT` - A process that is blocked on a resource and has an associated timeout.

When a process terminates, it enters a dead state called `PROC_DEAD`. The process context state diagram is shown in the following figure.

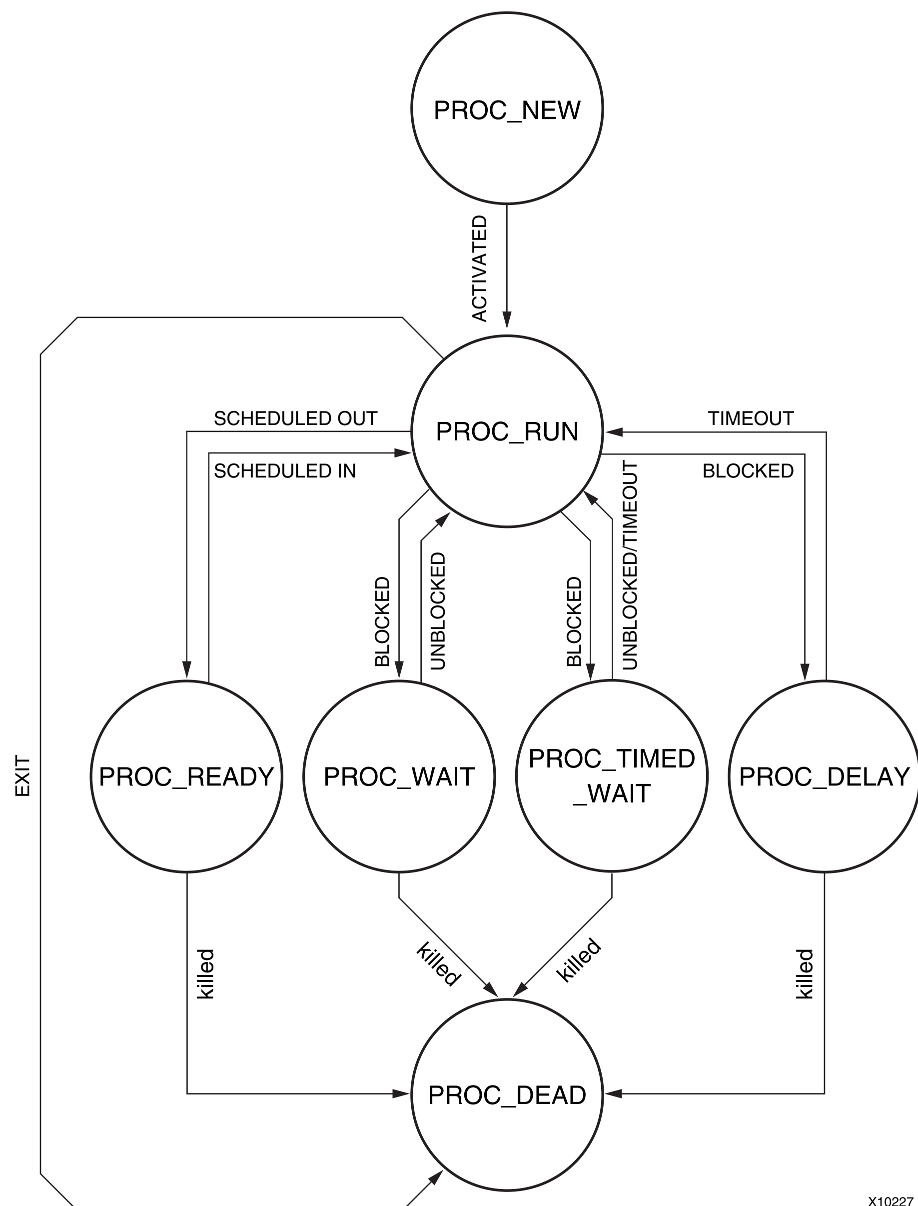

Figure 4: **Process Context States**

POSIX Interface

Xilkernel provides a POSIX interface to the kernel. Not all the concepts and interfaces defined by POSIX are available. A subset covering the most useful interfaces and concepts are implemented. Xilkernel programs can run almost equivalently on your desktop OS, like Linux or SunOS. This makes for easy application development, portability and legacy software support. The programming model appeals to those who have worked on equivalent POSIX interfaces on traditional operating systems. For those interfaces that have been provided, POSIX is rigorously adhered to in almost all cases. For cases that do differ, the differences are clearly specified. Refer to "Xilkernel API", for the actual interfaces and their descriptions.

Xilkernel Functions

Click an item below view function summaries and descriptions for:

- Thread Management
- Semaphores
- Message Queues
- Shared Memory
- Mutex Locks
- Dynamic Buffer Memory Management
- Software Timers
- Memory Protection Overview

Xilkernel API

Thread Management

Xilkernel supports the basic POSIX threads API. Thread creation and manipulation is done in standard POSIX notation. Threads are identified by a unique thread identifier. The thread identifier is of type `pthread_t`. This thread identifier uniquely identifies a thread for an operation. Threads created in the system have a kernel wrapper to which they return control to when they terminate. So, a specific exit function is not required at the end of the thread's code.

Thread stack is allocated automatically on behalf of the thread from a pool of Block Starting Symbol (BSS) memory that is statically allocated based upon the maximum number of system threads. You can also assign a custom piece of memory as the stack for each thread to create dynamically.

The entire thread module is optional and can be configured in or out as a part of the software specification. See "Configuring Thread Management," page 45 for more details on customizing this module.

Thread Management Function Summary

The following list is a linked summary of the thread management functions in Xilkernel. Click on a function to view a detailed description.

int pthread_create(pthread_t thread, pthread_attr_t* att, void*(*start_func)(void*),void* param)
void pthread_exit(void *value_ptr)
int pthread_join(pthread_t thread, void **value_ptr)
int pthread_detach(pthread_t target)
int pthread_equal(pthread_t t1, pthread_t t2)
int pthread_getschedparam(pthread_t thread, int *policy, struct sched_param *param)
int pthread_setschedparam(pthread_t thread, int policy, const struct sched_param *param)
int pthread_attr_init(pthread_attr_t* attr)
int pthread_attr_destroy (pthread_attr_t* attr)
int pthread_attr_setdetachstate(pthread_attr_t* attr, int dstate)
int pthread_attr_getdetachstate(pthread_attr_t* attr, int *dstate)
int pthread_attr_setschedparam(pthread_attr_t* attr, struct sched_param *schedpar)
int pthread_attr_getschedparam(pthread_attr_t* attr, struct sched_param* schedpar)
int pthread_attr_setstack(const pthread_attr_t *attr, void *stackaddr, size_t stacksize)
int pthread_attr_getstack(const pthread_attr_t *attr, void **stackaddr, size_t *stacksize)
pid_t get_currentPID(void)
int kill(pid_tpid)
int process_status(pid_t pid, p_stat *ps)
int xmk_add_static_thread(void* (*start_routine)(void *), int sched_priority)
int yield(void)

Thread Management Function Descriptions

The following descriptions are the thread management interface identifiers.

`int` **`pthread_create`**`(pthread_t` *`thread`*`, pthread_attr_t*` *`att`*`,`
`void*(*`*`start_func`*`)(void*),void*` *`param`*`)`

Parameters
: *`thread`* is the location at which to store the created thread's identifier.
: *`attr`* is the pointer to thread creation attributes structure.
: *`start_func`* is the start address of the function from which the thread needs to execute.
: *`param`* is the pointer argument to the thread function.

Returns
: 0 and thread identifier of the created thread in `*`*`thread`*, on success.
: -1 if *`thread`* refers to an invalid location.
: `EINVAL` if *`attr`* refers to invalid attributes.
: `EAGAIN` if resources unavailable to create the thread.

Description
: `pthread_create( )` creates a new thread, with attributes specified by *`attr`*, within a process. If *`attr`* is `NULL`, the default attributes are used. If the attributes specified by *`attr`* are modified later, the thread's attributes are not affected. Upon successful completion, `pthread_create( )` stores the ID of the created thread in the location referenced by thread. The thread is created executing *`start_routine`* with `arg` as its sole argument. If the *`start_routine`* returns, the effect is as if there was an implicit call to `pthread_exit( )` using the return value of *`start_routine`* as the exit status. This is explained in the *`pthread_exit`* description.
: You can control various attributes of a thread during its creation. See the *`pthread_attr`* routines for a description of the kinds of thread creation attributes that you can control.

Includes
: `xmk.h, pthread.h`

`void` **`pthread_exit`**`(void *`*`value_ptr`*`)`

Parameters
: *`value_ptr`* is a pointer to the return value of the thread.

Returns
: None.

Description
: The `pthread_exit( )` function terminates the calling thread and makes the value *`value_ptr`* available to any successful join with the terminating thread. Thread termination releases process context resources including, but not limited to, memory and attributes. An implicit call to `pthread_exit( )` is made when a thread returns from the creating start routine. The return value of the function serves as the thread's exit status. Therefore no explicit `pthread_exit( )` is required at the end of a thread.

Includes
: `xmk.h, pthread.h`

int **pthread_join**(pthread_t *thread*, void ***value_ptr*)

Parameters
value_ptr is a pointer to the return value of the thread.

Returns
0 on success.
ESRCH if the target thread is not in a joinable state or is an invalid thread.
EINVAL if the target thread already has someone waiting to join with it.

Description
The pthread_join( ) function suspends execution of the calling thread until the *pthread_t* (target thread) terminates, unless the target thread has already terminated.Upon return from a successful pthread_join( ) call with a non-NULL *value_ptr* argument, the value passed to the pthread_exit( ) function by the terminating thread is made available in the location referenced by *value_ptr*. When a pthread_join( ) returns successfully, the target thread has been terminated. The results of multiple simultaneous calls to pthread_join( ) specifying the same target thread are that only one thread succeeds and the others fail with EINVAL.

Note: No deadlock detection is provided.

Includes
xmk.h, pthread.h

---

pthread_t **pthread_self**(void)

Parameters
None.

Returns
On success, returns thread identifier of current thread.
Error behavior not defined.

Description
The pthread_self( ) function returns the thread ID of the calling thread.

Includes
xmk.h, pthread.h

---

int **pthread_detach**(pthread_t *target*)

Parameters
target is the target thread to detach.

Returns
0 on success.
ESRCH if target thread cannot be found.

Description
The pthread_detach( ) function indicates to the implementation that storage for the *thread* can be reclaimed when that thread terminates. If thread has not terminated, pthread_detach( ) does not cause it to terminate. The effect of multiple pthread_detach( ) calls on the same target thread is unspecified.

Includes
xmk.h, pthread.h

---

int **pthread_equal**(pthread_t *t1*, pthread_t *t2*)

Parameters
t1 and *t2* are the two thread identifiers to compare.

Returns
1 if *t1* and *t2* refer to threads that are equal.
0 otherwise.

Description
The pthread_equal( ) function returns a non-zero value if *t1* and *t2* are equal; otherwise, zero is returned. If either *t1* or *t2* are not valid thread IDs, zero is returned.

Includes
xmk.h, pthread.h

`int` **`pthread_getschedparam`**`(pthread_t` *`thread`*`, int` *`*policy`*`,`
`struct sched_param` *`*param`*`)`

Parameters

`thread` is the identifier of the thread on which to perform the operation.

`policy` is a pointer to the location where the global scheduling policy is stored.

`param` is a pointer to the scheduling parameters structure.

Returns

0 on success.

`ESRCH` if the value specified by thread does not refer to an existing thread.

`EINVAL` if param or policy refer to invalid memory.

Description

The `pthread_getschedparam( )` function gets the scheduling policy and parameters of an individual thread. For `SCHED_RR` there are no scheduling parameters; consequently, this routine is not defined for `SCHED_RR`.

For `SCHED_PRIO`, the only required member of the sched_param structure is the priority *`sched_priority`*. The returned priority value is the value specified by the most recent `pthread_getschedparam( )` or `pthread_create( )` call affecting the target thread.

It does not reflect any temporary adjustments to its priority as a result of any priority inheritance or ceiling functions.

This routine is defined only if scheduling type is `SCHED_PRIO`.

Returns

`xmk.h, pthread.h`

`int` **`pthread_setschedparam`**`(pthread_t` *`thread`*`, int` *`policy`*`,`
`const struct sched_param` *`*param)`*

Parameters

`thread` is the identifier of the thread on which to perform the operation.

`policy` is ignored.

`param` is a pointer to the scheduling parameters structure.

Returns

0 on success.

`ESRCH` if *`thread`* does not refer to a valid thread.

`EINVAL` if the scheduling parameters are invalid.

Description

The `pthread_setschedparam( )` function sets the scheduling policy and parameters of individual threads to be retrieved. For `SCHED_RR` there are no scheduling parameters; consequently this routine is not defined for `SCHED_RR`. For `SCHED_PRIO`, the only required member of the sched_param structure is the priority *`sched_priority`*. The priority value must be a valid value as configured in the scheduling parameters of the kernel. The policy parameter is ignored.

Note: This routine is defined only if scheduling type is `SCHED_PRIO`.

Includes

`xmk.h, pthread.h`

int **pthread_attr_init**(pthread_attr_t* *attr*)

Parameters	*attr* is a pointer to the attribute structure to be initialized.
Returns	0 on success. 1 on failure. EINVAL on invalid *attr* parameter.
Description	The pthread_attr_init( ) function initializes a thread attributes object *attr* with the default value for all of the individual attributes used by a given implementation. The function contents are defined in the sys/types.h header. ***Note:*** This function does not make a call to the kernel.
Includes	xmk.h, pthread.h

int **pthread_attr_destroy** (pthread_attr_t* *attr*)

Parameters	*attr* is a pointer to the thread attributes that must be destroyed.
Returns	0 on success. EINVAL on errors.
Description	The pthread_attr_destroy( ) function destroys a thread attributes object and sets *attr* to an implementation-defined invalid value. Re-initialize a destroyed *attr* attributes object with pthread_attr_init( ); the results of otherwise referencing the object after it is destroyed are undefined. ***Note:*** This function does not make a call to the kernel.
Includes	xmk.h, pthread.h

int **pthread_attr_setdetachstate**(pthread_attr_t* *attr*, int *dstate*)

Parameters	*attr* is the attribute structure on which the operation is to be performed. *dstate* is the detachstate required.
Returns	0 on success. EINVAL on invalid parameters.
Description	The detachstate attribute controls whether the thread is created in a detached state. If the thread is created detached, then when the thread exits, the thread's resources are detached without requiring a pthread_join( ) or a call pthread_detach( ).The application can set detachstate to either PTHREAD_CREATE_DETACHED or PTHREAD_CREATE_JOINABLE. ***Note:*** This does not make a call into the kernel.
Includes	xmk.h, pthread.h

int **pthread_attr_getdetachstate**(pthread_attr_t* *attr*, int *dstate*)

Parameters: *attr* is the attribute structure on which the operation is to be performed.
dstate is the location in which to store the detachstate.

Returns: 0 on success.
EINVAL on invalid parameters.

Description: The implementation stores either PTHREAD_CREATE_DETACHED or PTHREAD_CREATE_JOINABLE in dstate, if the value of detachstate was valid in *attr*.

Note: This does not make a call into the kernel.

Includes: xmk.h, pthread.h

---

int **pthread_attr_setschedparam**(pthread_attr_t* *attr*, struct sched_param **schedpar*)

Parameters: *attr* is the attribute structure on which the operation is to be performed.
schedpar is the location of the structure that contains the scheduling parameters.

Returns: 0 on success.
EINVAL on invalid parameters.
ENOTSUP for invalid scheduling parameters.

Description: The pthread_attr_setschedparam( ) function sets the scheduling parameter attributes in the *attr* argument.
The contents of the *sched_param* structure are defined in the sched.h header.

Note: This does not make a call into the kernel.

Includes: xmk.h, pthread.h

---

int **pthread_attr_getschedparam**(pthread_attr_t* *attr*, struct sched_param* *schedpar*)

Parameters: *attr* is the attribute structure on which the operation is to be performed.
schedpar is the location at which to store the *sched_param* structure.

Returns: 0 on success.
EINVAL on invalid parameters.

Description: The pthread_attr_getschedparam( ) gets the scheduling parameter attributes in the *attr* argument. The contents of the param structure are defined in the sched.h header.

Note: This does not make a call to the kernel.

Includes: xmk.h, pthread.h

`int` **`pthread_attr_setstack`**`(const pthread_attr_t *attr, void *stackaddr, size_t` *`stacksize`*`)`

Parameters: *`attr`* is the attributes structure on which to perform the operation.

`stackaddr` is base address of the stack memory.

`stacksize` is the size of the memory block in bytes.

Returns: 0 on success.

`EINVAL` if the *`attr`* param is invalid or if *`stackaddr`* is not aligned appropriately.

Description: The `pthread_attr_setstack( )` function shall set the thread creation stack attributes *`stackaddr`* and *`stacksize`* in the *`attr`* object.

The stack attributes specify the area of storage to be used for the created thread's stack. The base (lowest addressable byte) of the storage is *`stackaddr`*, and the size of the storage is *`stacksize`* bytes.

The *`stackaddr`* must be aligned appropriately according to the processor EABI, to be used as a stack; for example, `pthread_attr_setstack( )` might fail with `EINVAL` if (*`stackaddr`* and 0x3) is not 0.

Note: For the MicroBlaze processor, the alignment required is 4 bytes.

Includes: `xmk.h, pthread.h`

`int` **pthread_attr_getstack**`(const pthread_attr_t *`*`attr`*`, void **`*`stackaddr`*`, size_t *`*`stacksize`*`)`

Parameters: *`attr`* is the attributes structure on which to perform the operation.

`stackaddr` is the location at which to store the base address of the stack memory.

`stacksize` is the location at which to store the size of the memory block in bytes.

Returns: 0 on success.

`EINVAL` on invalid *`attr`*.

Description: The `pthread_attr_getstack( )` function retrieves the thread creation attributes related to stack of the specified attributes structure and stores it in `stackaddr` and `stacksize`.

Includes: `xmk.h, pthread.h`

`pid_t` **`get_currentPID`**`(void)`

Parameters: None.

Returns: The process identifier associated with the current thread or elf process.

Description: Gets the underlying process identifier of the process context that is executing currently. The process identifier is needed to perform certain operations like `kill( )` on both processes and threads.

Includes: `xmk.h, sys/process.h`

int **kill**(pid_t*pid*)

Parameters: *pid* is the PID of the process.

Returns: 0 on success.
-1 on failure.

Description: Removes the process context specified by *pid* from the system. If *pid* refers to the current executing process context, then it is equivalent to the current process context terminating. A kill can be invoked on processes that are suspended on wait queues or on a timeout. No indication is given to other processes that are dependant on this process context.

Note: This function is defined only if `CONFIG_KILL` is true. This can be configured in with the enhanced features category of the kernel.

Includes: `xmk.h, sys/process.h`

---

int **process_status**(pid_t *pid*, p_stat **ps*)

Parameters: *pid* is the PID of process.
ps is the buffer where the process status is returned.

Returns: Process status in `ps` on success.
`NULL` in *ps* on failure.

Description: Get the status of the process or thread, whose pid is *pid*. The status is returned in structure p_stat which has the following fields:

- *pid* is the process ID.
- *state* is the current scheduling state of the process.

The contents of `p_stat` are defined in the `sys/ktypes.h` header.

Includes: `xmk.h, sys/process.h`

---

int **xmk_add_static_thread**(void* (**start_routine*)(void *), int *sched_priority*)

Parameters: *start_routine* is the thread start routine.
sched_priority is the priority of the thread when the kernel is configured for priority scheduling.

Returns: 0 on success and -1 on failure.

Description: This function provides the ability to add a thread to the list of startup or static threads that run on kernel start, via C code. This function must be used prior to xilkernel_main() being invoked.

Includes: `xmk.h, sys/init.h`

`int yield(void)`

Parameters	None.
Returns	None.
Description	Yields the processor to the next process context that is ready to execute. The current process is put back in the appropriate ready queue. ***Note:*** This function is optional and included only if `CONFIG_YIELD` is defined. This can be configured in with the enhanced features category of the kernel.
Includes	`xmk.h, sys/process.h`

Semaphores

Xilkernel supports kernel-allocated POSIX semaphores that can be used for synchronization. POSIX semaphores are counting semaphores that also count below zero (a negative value indicates the number of processes blocked on the semaphore). Xilkernel also supports a few interfaces for working with named semaphores. The number of semaphores allocated in the kernel and the length of semaphore wait queues can be configured during system initialization. Refer to "Configuring Semaphores," page 46 for more details. The semaphore module is optional and can be configured in or out during system initialization. The message queue module, described later on in this document, uses semaphores. This module must be included if you are to use message queues.

Semaphore Function Summary

The following list provides a linked summary of the semaphore functions in Xilkernel. You can click on a function to go to the description.

int sem_init(sem_t *sem, int pshared, unsigned value)
int sem_destroy(sem_t* sem)
int sem_getvalue(sem_t* sem, int* value)
int sem_wait(sem_t* sem)
int sem_trywait(sem_t* sem)
int sem_timedwait(sem_t* sem, unsigned_ms)
sem_t* sem_open(const char* name, int oflag,...)
int sem_close(sem_t* sem)
int sem_post(sem_t* sem)
int sem_unlink(const char* name)

Semaphore Function Descriptions

The following are descriptions of the Xilkernel semaphore functions:

`int` **`sem_init`**`(sem_t *`*`sem`*`, int` *`pshared`*`, unsigned` *`value`*`)`

Parameters — *`sem`* is the location at which to store the created semaphore's identifier.

`pshared` indicates sharing status of the semaphore, between processes.

`value` is the initial count of the semaphore.

Note: *`pshared`* is unused currently.

Returns — 0 on success.

-1 on failure and sets `errno` appropriately. The `errno` is set to `ENOSPC` if no more semaphore resources are available in the system.

Description — The `sem_init( )` function initializes the unnamed semaphore referred to by *`sem`*. The value of the initialized semaphore is *`value`*. Following a successful call to `sem_init( )`, the semaphore can be used in subsequent calls to `sem_wait( )`, `sem_trywait( )`, `sem_post( )`, and `sem_destroy( )`. This semaphore remains usable until the semaphore is destroyed. Only sem itself can be used for performing synchronization. The result of referring to copies of sem in calls to `sem_wait( )`, `sem_trywait( )`, `sem_post( )`, and `sem_destroy( )` is undefined. Attempting to initialize an already initialized semaphore results in undefined behavior.

Includes — `xmk.h, semaphore.h`

`int` **`sem_destroy`**`(sem_t*` *`sem`*`)`

Parameters — *`sem`* is the semaphore to be destroyed.

Returns — 0 on success.

-1 on failure and sets `errno` appropriately. The `errno` can be set to:

- `EINVAL` if the semaphore identifier does not refer to a valid semaphore.
- `EBUSY` if the semaphore is currently locked, and processes are blocked on it.

Description — The `sem_destroy( )` function destroys the unnamed semaphore indicated by sem. Only a semaphore that was created using sem_init( ) can be destroyed using `sem_destroy( )`; the effect of calling `sem_destroy( )` with a named semaphore is undefined. The effect of subsequent use of the semaphore *`sem`* is undefined until *`sem`* is re-initialized by another call to `sem_init( )`.

Includes — `xmk.h, semaphore.h`

`int sem_getvalue(sem_t* sem, int* value)`

Parameters
: *sem* is the semaphore identifier.
: *value* is the location where the semaphore value is stored.

Returns
: 0 on success and *value* appropriately filled in.
: -1 on failure and sets `errno` appropriately.The `errno` can be set to `EINVAL` if the semaphore identifier refers to an invalid semaphore.

Description
: The `sem_getvalue( )` function updates the location referenced by the *sval* argument to have the value of the semaphore referenced by sem without affecting the state of the semaphore. The updated value represents an actual semaphore value that occurred at some unspecified time during the call, but it need not be the actual value of the semaphore when it is returned to the calling process.
: If sem is locked, then the object to which *sval* points is set to a negative number whose absolute value represents the number of processes waiting for the semaphore at some unspecified time during the call.

Includes
: `xmk.h, semaphore.h`

`int sem_wait(sem_t* sem)`

Parameters
: *sem* is the semaphore identifier.

Returns
: 0 on success and the semaphore in a locked state.
: -1 on failure and `errno` is set appropriately. The `errno` can be set to:
- `EINVAL` if the semaphore identifier is invalid.
- `EIDRM` if the semaphore was forcibly removed.

Description
: The `sem_wait( )` function locks the semaphore referenced by *sem* by performing a semaphore lock operation on that semaphore. If the semaphore value is currently zero, then the calling thread does not return from the call to `sem_wait( )` until it either locks the semaphore or the semaphore is forcibly destroyed.
: Upon successful return, the state of the semaphore is locked and remains locked until the `sem_post( )` function is executed and returns successfully.
: ***Note:*** When a process is unblocked within the `sem_wait` call, where it blocked due to unavailability of the semaphore, the semaphore might have been destroyed forcibly. In such a case, -1 is returned. Semaphores might be forcibly destroyed due to destroying message queues that use semaphores internally. No deadlock detection is provided.

Includes
: `xmk.h, semaphore.h`

`int sem_trywait(sem_t* sem)`

Parameters
: *sem* is the semaphore identifier.

Returns
: 0 on success.
: -1 on failure and *errno* is set appropriately. The *errno* can be set to:
- `EINVAL` if the semaphore identifier is invalid.
- `EAGAIN` if the semaphore could not be locked immediately.

Description
: The `sem_trywait( )` function locks the semaphore referenced by *sem* only if the semaphore is currently not locked; that is, if the semaphore value is currently positive. Otherwise, it does not lock the semaphore and returns -1.

Includes
: `xmk.h, semaphore.h`

`int` **`sem_timedwait`**`(sem_t*` *`sem`*`, unsigned` *`ms`*`)`

Parameters — *`sem`* is the semaphore identifier.

Returns — 0 on success and the semaphore in a locked state.

-1 on failure and *`errno`* is set appropriately. The *`errno`* can be set to:

- `EINVAL` - If the semaphore identifier does not refer to a valid semaphore.
- `ETIMEDOUT` - The semaphore could not be locked before the specified timeout expired.
- `EIDRM` - If the semaphore was forcibly removed from the system.

Description — The `sem_timedwait( )` function locks the semaphore referenced by *`sem`* by performing a semaphore lock operation on that semaphore. If the semaphore value is currently zero, then the calling thread does not return from the call to `sem_timedwait( )` until one of the following conditions occurs:

- It locks the semaphore.
- The semaphore is forcibly destroyed.
- The timeout specified has elapsed.

Upon successful return, the state of the semaphore is locked and remains locked until the `sem_post( )` function is executed and returns successfully.

Note: When a process is unblocked within the `sem_wait` call, where it blocked due to unavailability of the semaphore, the semaphore might have been destroyed forcibly. In such a case, -1 is returned. Semaphores maybe forcibly destroyed due to destroying message queues which internally use semaphores. No deadlock detection is provided.

Note: This routine depends on software timers support being present in the kernel and is defined only if `CONFIG_TIME` is true.

Note: This routine is slightly different from the POSIX equivalent. The POSIX version specifies the timeout as absolute wall-clock time. Because there is no concept of absolute time in Xilkernel, we use relative time specified in milliseconds.

Includes — `xmk.h, semaphore.h`

`sem_t* sem_open(const char* name, int oflag,...)`

Parameters

name points to a string naming a semaphore object.

oflag is the flag that controls the semaphore creation.

Returns

A pointer to the created/existing semaphore identifier.

`SEM_FAILED` on failures and when *errno* is set appropriately. The *errno* can be set to:

- `ENOSPC` - If the system is out of resources to create a new semaphore (or mapping).
- `EEXIST` - if `O_EXCL` has been requested and the named semaphore already exists.
- `EINVAL` - if the parameters are invalid.

Description

The `sem_open( )` function establishes a connection between a named semaphore and a process. Following a call to `sem_open( )` with semaphore *name*, the process can reference the semaphore associated with name using the address returned from the call. This semaphore can be used in subsequent calls to `sem_wait( )`, `sem_trywait( )`, `sem_post( )`, and `sem_close( )`. The semaphore remains usable by this process until the semaphore is closed by a successful call to `sem_close( )`. The *oflag* argument controls whether the semaphore is created or merely accessed by the call to `sem_open( )`. The bits that can be set in *oflag* are:

- `O_CREAT`
  Used to create a semaphore if it does not already exist. If `O_CREAT` is set and the semaphore already exists, then `O_CREAT` has no effect, except as noted under `O_EXCL`. Otherwise, `sem_open( )` creates a named semaphore. `O_CREAT` requires a third and a fourth argument: *mode*, which is of type `mode_t`, and *value*, which is of type unsigned.
- `O_EXCL`
  If `O_EXCL` and `O_CREAT` are set, `sem_open( )` fails if the semaphore name exists. The check for the existence of the semaphore and the creation of the semaphore if it does not exist are atomic with respect to other processes executing `sem_open( )` with `O_EXCL` and `O_CREAT` set. If `O_EXCL` is set and `O_CREAT` is not set, the effect is undefined.

Note: The *mode* argument is unused currently. This interface is optional and is defined only if `CONFIG_NAMED_SEMA` is set to `TRUE`.

Note: If flags other than `O_CREAT` and `O_EXCL` are specified in the *oflag* parameter, an error is signalled.

The semaphore is created with an initial value of *value*.

After the *name* semaphore has been created by `sem_open( )` with the `O_CREAT` flag, other processes can connect to the semaphore by calling `sem_open( )` with the same value of *name*.

If a process makes multiple successful calls to `sem_open( )` with the same value for name, the same semaphore address is returned for each such successful call, assuming that there have been no calls to `sem_unlink( )` for this semaphore.

Includes

`xmk.h, semaphore.h`

`int sem_close(sem_t* sem)`

Parameters

sem is the semaphore identifier.

Returns

0 on success.

-1 on failure and sets *errno* appropriately. The *errno* can be set to:

- `EINVAL` - If the semaphore identifier is invalid.
- `ENOTSUP` - If the semaphore is currently locked and/or processes are blocked on the semaphore.

Description	The `sem_close( )` function indicates that the calling process is finished using the named semaphore *sem*. The `sem_close( )` function deallocates (that is, make available for reuse by a subsequent `sem_open( )` by this process) any system resources allocated by the system for use by this process for this semaphore. The effect of subsequent use of the semaphore indicated by *sem* by this process is undefined. The name mapping for this named semaphore is also destroyed. The call fails if the semaphore is currently locked. ***Note:*** This interface is optional and is defined only if `CONFIG_NAMED_SEMA` is true.
Includes	`xmk.h, semaphore.h`

---

int **sem_post**(sem_t* *sem*)

Parameters	*sem* is the semaphore identifier.
Returns	0 on success. -1 on failure and sets *errno* appropriately. The `errno` can be set to `EINVAL` if the semaphore identifier is invalid.
Description	The `sem_post( )` function performs an unlock operation on the semaphore referenced by the *sem* identifier. If the semaphore value resulting from this operation is positive, then no threads were blocked waiting for the semaphore to become unlocked and the semaphore value is incremented. If the value of the semaphore resulting from this operation is zero or negative, then one of the threads blocked waiting for the semaphore is allowed to return successfully from its call to `sem_wait( )`. This is either the first thread on the queue, if scheduling mode is `SCHED_RR` or, it is the highest priority thread in the queue, if scheduling mode is `SCHED_PRIO`. ***Note:*** If an unlink operation was requested on the semaphore, the post operation performs an unlink when no more processes are waiting on the semaphore.
Includes	`xmk.h, semaphore.h`

`int sem_unlink(const char* name)`

Parameters	`name` is the name that refers to the semaphore.
Returns	0 on success. -1 on failure and *errno* is set appropriately. *errno* can be set to `ENOENT` - If an entry for name cannot be located.
Description	The `sem_unlink( )` function removes the semaphore named by the string name. If the semaphore named by `name` has processes blocked on it, then `sem_unlink( )` has no immediate effect on the state of the semaphore. The destruction of the semaphore is postponed until all blocked and locking processes relinquish the semaphore. Calls to `sem_open( )` to recreate or reconnect to the semaphore refer to a new semaphore after `sem_unlink( )` is called. The `sem_unlink( )` call does not block until all references relinquish the semaphore; it returns immediately. ***Note:*** If an unlink operation had been requested on the semaphore, the unlink is performed on a post operation that sees that no more processes waiting on the semaphore. This interface is optional and is defined only if `CONFIG_NAMED_SEMA` is true.
Includes	`xmk.h, semaphore.h`

Message Queues

Xilkernel supports kernel allocated X/Open System Interface (XSI) message queues. XSI is a set of optional interfaces under POSIX. Message queues can be used as an IPC mechanism. The message queues can take in arbitrary sized messages. However, buffer memory allocation must be configured appropriately for the memory blocks required for the messages, as a part of system buffer memory allocation initialization.The number of message queue structures allocated in the kernel and the length of the message queues can be also be configured during system initialization. The message queue module is optional and can be configured in/out. Refer to "Configuring Message Queues," page 46 for more details. This module depends on the semaphore module and the dynamic buffer memory allocation module being present in the system. There is also a larger, but more powerful message queue functionality that can be configured if required. When the enhanced message queue interface is chosen, then *`malloc`* and *`free`* are used to allocate and free space for the messages. Therefore, arbitrary sized messages can be passed around without having to make sure that buffer memory allocation APIs can handle requests for arbitrary size.

Note: When using the enhanced message queue feature, you must choose your global heap size carefully, such that requests for heap memory from the message queue interfaces are satisfied without errors. You must also be aware of thread-safety issues when using `malloc( )`, `free ( )` in your own code. You must disable interrupts and context switches before invoking the dynamic memory allocation routines. You must follow the same rules when using any other library routines that may internally use dynamic memory allocation.

Message Queue Function Summary

The following list provides a linked summary of the message queues in Xilkernel. You can click on a function to go to the description.

int msgget(key_t key, int msgflg)
int msgctl(int msqid, int cmd, struct msqid_ds* buf)
int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg)
ssize_t msgrcv(int msqid, void *msgp, size_t nbytes, long msgtyp, int msgflg)

Message Queue Function Descriptions

The Xilkernel message queue function descriptions are as follows:

`int msgget(key_t key, int msgflg)`

Parameters

key is a unique identifier for referring to the message queue.

msgflg specifies the message queue creation options.

Returns

A unique non-negative integer message queue identifier.

-1 on failure and sets *errno* appropriately; errno can be set to:

- EEXIST - If a message queue identifier exists for the argument key but ((msgflg and IPC_CREAT) and msgflg & IPC_EXCL) is non-zero.
- ENOENT - A message queue identifier does not exist for the argument key and (msgflg & IPC_CREAT) is 0.
- ENOSPC - If the message queue resources are exhausted.

Description

The msgget( ) function returns the message queue identifier associated with the argument key. A message queue identifier, associated message queue, and data structure (see sys/kmsg.h), are created for the argument key if the argument key does not already have a message queue identifier associated with it, and (*msgflg* and IPC_CREAT) is non-zero.

Upon creation, the data structure associated with the new message queue identifier is initialized as follows:

- msg_qnum, msg_lspid, msg_lrpid are set equal to 0.
- msg_qbytes is set equal to the system limit (MSGQ_MAX_BYTES).

The msgget( ) function fails if a message queue identifier exists for the argument key but ((*msgflg* and IPC_CREAT) and (*msgflg* & IPC_EXCL)) is non-zero.

IPC_PRIVATE is not supported. Also, messages in the message queue are not required to be of the form shown below. There is no support for message type based message receives and sends in this implementation.

The following is an example code snippet:

```
struct mymsg {
  long mtype; /* Message type. */
  char mtext[some_size]; /* Message text. */
  }
```

Includes

xmk.h, sys/msg.h, sys/ipc.h

`int msgctl(int msqid, int cmd, struct msqid_ds* buf)`

Parameters

msqid is the message queue identifier.
cmd is the command.
buf is the data buffer

Returns

0 on success. Status is returned in *buf* for `IPC_STAT`.

-1 on failure and sets `errno` appropriately. The *errno* can be set to `EINVAL` if any of the following conditions occur:

- *msgid* parameter refers to an invalid message queue.
- *cmd* is invalid.
- *buf* contains invalid parameters.

Description

The `msgctl( )` function provides message control operations as specified by *cmd*. The values for *cmd*, and the message control operations they specify, are:

- `IPC_STAT` - Places the current value of each member of the msqid_ds data structure associated with msqid into the structure pointed to by `buf`. The contents of this structure are defined in `sys/msg.h`.
- `IPC_SET` - Unsupported.
- `IPC_RMID` - Removes the message queue identifier specified by *msqid* from the system and destroys the message queue and associated `msqid_ds` data structure. The remove operation forcibly destroys the semaphores used internally and unblocks processes that are blocked on the semaphore. It also deallocates memory allocated for the messages in the queue.

Includes

`xmk.h, sys/msg.h, sys/ipc.h`

`int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg)`

Parameters

msqid is the message queue identifier.

msgp is a pointer to the message buffer.

msgsz is the size of the message.

msgflg is used to specify message send options.

Returns

0 on success.

-1 on failure and sets *errno* appropriately. The *errno* can be set to:

- EINVAL - The value of msgid is not a valid message queue identifier.
- ENOSPC - The system could not allocate space for the message.
- EIDRM - The message queue was removed from the system during the send operation.

Description

The `msgsnd( )` function sends a message to the queue associated with the message queue identifier specified by `msqid`.

The argument *msgflg* specifies the action to be taken if the message queue is full:

If (*msgflg* and IPC_NOWAIT) is non-zero, the message is not sent and the calling thread returns immediately.

If (*msgflg* and IPC_NOWAIT) is 0, the calling thread suspends execution until one of the following occurs:

- The condition responsible for the suspension no longer exists, in which case the message is sent.
- The message queue identifier msqid is removed from the system; when this occurs a -1 is returned.

The send fails if it is unable to allocate memory to store the message inside the kernel. On a successful send operation, the *msg_lspid* and *msg_qnum* members of the message queues are appropriately set.

Includes

`xmk.h, sys/msg.h, sys/ipc.h`

```
ssize_t msgrcv(int msqid, void *msgp, size_t nbytes, long
   msgtyp, int msgflg)
```

Parameters	*msqid* is the message queue identifier. *msgp* is the buffer where the received message is to be copied. *nbytes* specifies the size of the message that the buffer can hold. *msgtyp* is currently unsupported. *msgflg* is used to control the message receive operation.
Returns	On success, stores received message in user buffer and returns number of bytes of data received. -1 on failure and sets *errno* appropriately. The *errno* can be set to: • EINVAL - If msgid is not a valid message queue identifier. • EIDRM - If the message queue was removed from the system. • ENOMSG - msgsz is smaller than the size of the message in the queue.

Description	The msgrcv( ) function reads a message from the queue associated with the message queue identifier specified by *msqid* and places it in the user-defined buffer pointed to by *msgp.*
	The argument *msgsz* specifies the size in bytes of the message. The received message is truncated to *msgsz* bytes if it is larger than *msgsz* and (*msgflg* and MSG_NOERROR) is non-zero. The truncated part of the message is lost and no indication of the truncation is given to the calling process. If MSG_NOERROR is not specified and the received message is larger than *nbytes*, then a -1 is returned signalling error.
	The argument *msgflg* specifies the action to be taken if a message is not on the queue. These are as follows:
	• If (*msgflg* and IPC_NOWAIT) is non-zero, the calling thread returns immediately with a return value of -1.
	• If (*msgflg* and IPC_NOWAIT) is 0, the calling thread suspends execution until one of the following occurs:
	  ♦ A message is placed on the queue
	  ♦ The message queue identifier *msqid* is removed from the system; when this occurs -1 is returned
	Upon successful completion, the following actions are taken with respect to the data structure associated with *msqid*:
	• *msg_qnum* is decremented by 1.
	• *msg_lrpid* is set equal to the process ID of the calling process.

Includes	xmk.h, sys/msg.h, sys/ipc.h

Shared Memory

Xilkernel supports kernel-allocated XSI shared memory. XSI is the X/Open System Interface which is a set of optional interfaces under POSIX. Shared memory is a common, low-latency IPC mechanism. Shared memory blocks required during run-time must be identified and specified during the system configuration. From this specification, buffer memory is allocated to each shared memory region. Shared memory is currently not allocated dynamically at run-time. This module is optional and can be configured in or out during system specification. Refer to “Configuring Shared Memory,” page 47 for more details.

Shared Memory Function Summary

The following list provides a linked summary of the shared memory functions in Xilkernel. You can click on a function to go to the description.

int shmget(key_t key, size_t size, int shmflg)
int shmctl(int shmid, int cmd, struct shmid_ds *buf)
void* shmat(int shmid, const void *shmaddr, int flag)
int shm_dt(void *shmaddr)

Shared Memory Function Descriptions

The Xilkernel shared memory interface is described below.

> ***Caution!*** The memory buffers allocated by the shared memory API might not be aligned at word boundaries. Therefore, structures should not be arbitrarily mapped to shared memory segments, without checking if alignment requirements are met.

`int` **`shmget`**`(key_t` *`key`*`, size_t` *`size`*`, int` *`shmflg`*`)`

Parameters	*key* is used to uniquely identify the shared memory region. *size* is the requested size of the shared memory segment. *shmflg* specifies segment creation options.
Returns	Unique non-negative shared memory identifier on success. -1 on failure and sets *errno* appropriately: *errno* can be set to: ♦ EEXIST - A shared memory identifier exists for the argument key but (*shmflg* and IPC_CREAT) and (*shmflg* and IPC_EXCL) is non-zero. ♦ ENOTSUP - Unsupported *shmflg.* ♦ ENOENT - A shared memory identifier does not exist for the argument key and (*shmflg* and IPC_CREAT) is 0.
Description	The shmget( ) function returns the shared memory identifier associated with key. A shared memory identifier, associated data structure, and shared memory segment of at least size bytes (see sys/shm.h) are created for key if one of the following is true: ♦ *key* is equal to IPC_PRIVATE. ♦ *key* does not already have a shared memory identifier associated with it and (*shmflg* and IPC_CREAT) is non-zero. Upon creation, the data structure associated with the new shared memory identifier shall be initialized.The value of *shm_segsz* is set equal to the value of size. The values of *shm_lpid*, *shm_nattch*, *shm_cpid* are all initialized appropriately. When the shared memory segment is created, it is initialized with all zero values. At least one of the shared memory segments available in the system must match *exactly* the requested size for the call to succeed. Key IPC_PRIVATE is not supported.
Includes	xmk.h, sys/shm.h, sys/ipc.h

`int` **`shmctl`**`(int` *`shmid`*`, int` *`cmd`*`, struct shmid_ds *`*`buf`*`)`

Parameters	*shmid* is the shared memory segment identifier. *cmd* is the command to the control function. *buf* is the buffer where the status is returned.
Returns	0 on success. Status is returned in buffer for IPC_STAT. -1 on failure and sets *errno* appropriately: *errno* can be set to EINVAL on the following conditions: • if *shmid* refers to an invalid shared memory segment. • if *cmd* or other params are invalid.
Description	The shmctl( ) function provides a variety of shared memory control operations as specified by cmd. The following values for *cmd* are available: • IPC_STAT: places the current value of each member of the shmid_ds data structure associated with shmid into the structure pointed to by buf. The contents of the structure are defined in sys/shm.h. • IPC_SET is not supported. • IPC_RMID: removes the shared memory identifier specified by *shmid* from the system and destroys the shared memory segment and shmid_ds data structure associated with it. No notification is sent to processes still attached to the segment.
Includes	xmk.h, sys/shm.h, sys/ipc.h

`void* shmat(int shmid, const void *shmaddr, int flag)`

Parameters	*shmid* is the shared memory segment identifier. *shmaddr* is used to specify the location, to attach shared memory segment. This is currently unused. *flag* is used to specify shared memory (SHM) attach options.
Returns	The start address of the shared memory segment on success. NULL on failure and sets *errno* appropriately. *errno* can be set to EINVAL if *shmid* refers to an invalid shared memory segment
Description	shmat( ) increments the value of *shm_nattch* in the data structure associated with the shared memory ID of the attached shared memory segment and returns the start address of the segment. shm_lpid is also appropriately set. ***Note:*** *shmaddr* and *flag* arguments are not used.
Includes	xmk.h, sys/shm.h, sys/ipc.h

`int shm_dt(void *shmaddr)`

Parameters	*shmaddr* is the shared memory segment address that is to be detached.
Returns	0 on success. -1 on failure and sets errno appropriately. The errno can be set to EINVAL if shmaddr is not within any of the available shared memory segments.
Description	The shmdt( ) function detaches the shared memory segment located at the address specified by *shmaddr* from the address space of the calling process. The value of *shm_nattch* is also decremented. The memory segment is not removed from the system and can be attached to again.
Includes	xmk.h, sys/shm.h, sys/ipc.h

Mutex Locks

Xilkernel provides support for kernel allocated POSIX thread mutex locks. This synchronization mechanism can be used alongside of the *pthread_* API. The number of mutex locks and the length of the mutex lock wait queue can be configured during system specification. PTHREAD_MUTEX_DEFAULT and PTHREAD_MUTEX_RECURSIVE type mutex locks are supported. This module is also optional and can be configured in or out during system specification. Refer to "Configuring Shared Memory," page 47 for more details.

Mutex Lock Function Summary

The following list provides a linked summary of the Mutex locks in Xilkernel. You can click on a function to go to the description.

int pthread_mutex_init(pthread_mutex_t* mutex, const pthread_mutexattr_t* attr)
int pthread_mutex_destroy(pthread_mutex_t* mutex)
int pthread_mutex_lock(pthread_mutex_t* mutex)
int pthread_mutex_trylock(pthread_mutex_t* mutex)
int pthread_mutex_unlock(pthread_mutex_t* mutex)
int pthread_mutexattr_init(pthread_mutexattr_t* attr)
int pthread_mutexattr_destroy(pthread_mutexattr_t* attr)
int pthread_mutexattr_settype(pthread_mutexattr_t* attr, int type)
int pthread_mutexattr_gettype(pthread_mutexattr_t* attr, int *type)

Mutex Lock Function Descriptions

The Mutex lock function descriptions are as follows:

```
int pthread_mutex_init(pthread_mutex_t* mutex, const
    pthread_mutexattr_t* attr)
```

Parameters	*mutex* is the location where the newly created mutex lock's identifier is to be stored. *attr* is the mutex creation attributes structure.
Returns	0 on success and mutex identifier in *`mutex`. `EAGAIN` if system is out of resources.
Description	The `pthread_mutex_init( )` function initializes the mutex referenced by *mutex* with attributes specified by *attr*. If *attr* is `NULL`, the default mutex attributes are used; the effect is the same as passing the address of a default mutex attributes object. Refer to the `pthread_mutexattr_` routines, which are documented starting on page 32 to determine what kind of mutex creation attributes can be changed. Upon successful initialization, the state of the mutex becomes initialized and unlocked. Only the mutex itself can be used for performing synchronization. The result of referring to copies of mutex in calls to `pthread_mutex_lock( )`, `pthread_mutex_trylock( )`, `pthread_mutex_unlock( )`, and `pthread_mutex_destroy( )` is undefined. Attempting to initialize an already initialized mutex results in undefined behavior. In cases where default mutex attributes are appropriate, the macro `PTHREAD_MUTEX_INITIALIZER` can be used to initialize mutexes that are statically allocated. The effect is equivalent to dynamic initialization by a call to `pthread_mutex_init( )` with parameter *attr* specified as `NULL`, with the exception that no error checks are performed. For example: `static pthread_mutex_t foo_mutex = PTHREAD_MUTEX_INITIALIZER;`
Includes	`xmk.h, pthread.h`

Note: The mutex locks allocated by Xilkernel follow the semantics of `PTHREAD_MUTEX_DEFAULT` mutex locks by default. The following actions will result in undefined behavior:

- Attempting to recursively lock the mutex.
- Attempting to unlock the mutex if it was not locked by the calling thread.
- Attempting to unlock the mutex if it is not locked.

`int pthread_mutex_destroy(pthread_mutex_t* mutex)`

Parameters	*mutex* is the mutex identifier.
Returns	0 on success. EINVAL if *mutex* refers to an invalid identifier.
Description	The `pthread_mutex_destroy( )` function destroys the mutex object referenced by *mutex;* the mutex object becomes, in effect, uninitialized. A destroyed mutex object can be reinitialized using `pthread_mutex_init( )`; the results of otherwise referencing the object after it has been destroyed are undefined. ***Note:*** Mutex lock/unlock state disregarded during destroy. No consideration is given for waiting processes.
Includes	`xmk.h, pthread.h`

`int pthread_mutex_lock(pthread_mutex_t* mutex)`

Parameters	*mutex* is the mutex identifier.
Returns	0 on success and mutex in a locked state. EINVAL on invalid *mutex* reference. -1 on unhandled errors.
Description	The mutex object referenced by *mutex* is locked by the thread calling `pthread_mutex_lock( )`. If the mutex is already locked, the calling thread blocks until the mutex becomes available. If the mutex type is `PTHREAD_MUTEX_RECURSIVE`, then the mutex maintains the concept of a lock count. When a thread successfully acquires a mutex for the first time, the lock count is set to one. Every time a thread relocks this mutex, the lock count is incremented by one. Each time the thread unlocks the mutex, the lock count is decremented by one. If the mutex type is `PTHREAD_MUTEX_DEFAULT`, attempting to recursively lock the mutex results in undefined behavior. If successful, this operation returns with the mutex object referenced by mutex in the locked state.
Includes	`xmk.h, pthread.h`

`int pthread_mutex_trylock(pthread_mutex_t* mutex)`

Parameters
: `mutex` is the mutex identifier.

Returns
: 0 on success.
: `mutex` in a locked state.
: `EINVAL` on invalid `mutex` reference,
: `EBUSY` if `mutex` is already locked.
: -1 on unhandled errors.

Description
: The mutex object referenced by `mutex` is locked by the thread calling `pthread_mutex_trlock( )`. If the mutex is already locked, the calling thread returns immediately with `EBUSY`.
: If the mutex type is `PTHREAD_MUTEX_RECURSIVE`, then the mutex maintains the concept of a lock count.
: When a thread successfully acquires a mutex for the first time, the lock count is set to one.
: Every time a thread relocks this mutex, the lock count is incremented by one. Each time the thread unlocks the mutex, the lock count is decremented by one.
: If the mutex type is `PTHREAD_MUTEX_DEFAULT`, attempting to recursively lock the mutex results in undefined behavior. If successful, this operation returns with the mutex object referenced by mutex in the locked state.

Includes
: `xmk.h, pthread.h`

`int pthread_mutex_unlock(pthread_mutex_t* mutex)`

Parameters
: `mutex` is the mutex identifier.

Returns
: 0 on success, `EINVAL` on invalid mutex reference.
: -1 on undefined errors.

Description
: The `pthread_mutex_unlock( )` function releases the mutex object referenced by `mutex`. If there are threads blocked on the mutex object referenced by `mutex` when `pthread_mutex_unlock( )` is called, resulting in the mutex becoming available, the scheduling policy determines which thread will acquire the mutex. If it is `SCHED_RR`, then the thread that is at the head of the mutex wait queue is unblocked and allowed to lock the mutex.
: If the mutex type is `PTHREAD_MUTEX_RECURSIVE`, the mutex maintains the concept of a lock count. When the lock count reaches zero, the mutex becomes available for other threads to acquire. If a thread attempts to unlock a mutex that it has not locked or a mutex which is unlocked, an error is returned.
: If the mutex type is `PTHREAD_MUTEX_DEFAULT` the following actions result in undefined behavior:
: - Attempting to unlock the mutex if it was not locked by the calling thread.
: - Attempting to unlock the mutex if it is not locked.
: If successful, this operation returns with the mutex object referenced by `mutex` in the unlocked state.

Includes
: `xmk.h, pthread.h`

`int pthread_mutexattr_init(pthread_mutexattr_t* attr)`

Parameters	*attr* is the location of the attributes structure.
Returns	0 on success. EINVAL if *attr* refers to an invalid location.
Description	The pthread_mutexattr_init( ) function initializes a mutex attributes object *attr* with the default value for all of the attributes defined by the implementation. Refer to sys/types.h for the contents of the pthread_mutexattr structure. ***Note:*** This routine does not involve a call into the kernel.
Includes	xmk.h, pthread.h

`int pthread_mutexattr_destroy(pthread_mutexattr_t* attr)`

Parameters	*attr* is the location of the attributes structure.
Returns	0 on success. EINVAL if *attr* refers to an invalid location.
Description	The pthread_mutexattr_destroy( ) function destroys a mutex attributes object; the object becomes, in effect, uninitialized. ***Note:*** This routine does not involve a call into the kernel.
Includes	xmk.h, pthread.h

`int pthread_mutexattr_settype(pthread_mutexattr_t* attr, int type)`

Parameters	*attr* is the location of the attributes structure. *type* is the type to which to set the mutex.
Returns	0 on success. EINVAL if *attr* refers to an invalid location or if *type* is an unsupported type.
Description	The pthread_mutexattr_settype( ) function sets the type of a mutex in a mutex attributes structure to the specified type. Only PTHREAD_MUTEX_DEFAULT and PTHREAD_MUTEX_RECURSIVE are supported. ***Note:*** This routine does not involve a call into the kernel.
Includes	xmk.h, pthread.h

`int pthread_mutexattr_gettype(pthread_mutexattr_t* attr, int *type)`

Parameters	*attr* is the location of the attributes structure. *type* is a pointer to the location at which to store the mutex.
Returns	0 on success. EINVAL if *attr* refers to an invalid location.
Description	The pthread_mutexattr_gettype( ) function gets the type of a mutex in a mutex attributes structure and stores it in the location pointed to by *type*.
Includes	xmk.h, pthread.h

Dynamic Buffer Memory Management

The kernel provides a buffer memory allocation scheme, which can be used by applications that need dynamic memory allocation. These interfaces are alternatives to the standard C memory allocation routines - malloc( ), free( ) which are much slower and bigger, though more powerful. The allocation routines hand off pieces of memory from a pool of memory that the user passes to the buffer memory manager.

The buffer memory manager manages the pool of memory. You can dynamically create new pools of memory. You can also statically specify the different memory blocks sizes and number of such memory blocks required for your applications. Refer to "Configuring Buffer Memory Allocation," page 47 for more details. This method of buffer management is relatively simple, small and a fast way of allocating memory. The following are the buffer memory allocation interfaces. This module is optional and can be included during system initialization.

Dynamic Buffer Memory Management Function Summary

The following list provides a linked summary of the dynamic buffer memory management functions in Xilkernel. You can click on a function to go to the description.

int bufcreate(membuf_t *mbuf, void *memptr, int nblks, size_t blksiz)
int bufdestroy(membuf_t mbuf)
void* bufmalloc(membuf_t mbuf, size_t siz)
void buffree(membuf_t mbuf, void* mem)

> ***Caution!*** The buffer memory allocation API internally uses the memory pool handed down the by the user to store a free-list in-place within the memory pool. As a result, only memory sizes greater than or equal to 4 bytes long are supported by the buffer memory allocation APIs. Also, because there is a free-list being built in-place within the memory pool, requests in which memory block sizes are not multiples of 4 bytes cause unalignment at run time. If your software platform can handle unalignment natively or through exceptions then this does not present an issue. The memory buffers allocated and returned by the buffer memory allocation API might also not be aligned at word boundaries. Therefore, your application should not arbitrarily map structures to memory allocated in this way without first checking if alignment and padding requirements are met.

Dynamic Buffer Memory Management Function Descriptions

The dynamic buffer memory management function descriptions are as follows:

`int **bufcreate**(membuf_t *mbuf, void *memptr, int nblks, size_t blksiz)`

Parameters	*mbuf* is location at which to store the identifier of the memory pool created. *memptr* is the pool of memory to use. *nblks* is the number of memory blocks that this pool should support. *blksiz* is the size of each memory block in bytes.
Returns	0 on success and stores the identifier of the created memory pool in the location pointed to by *mbuf*. -1 on errors.
Description	Creates a memory pool out of the memory block specified in *memptr*. *nblks* number of chunks of memory are defined within the pool, each of size *blksiz*. Therefore, *memptr* must point to at least *(nblks * blksiz)* bytes of memory. *blksiz* must be greater than or equal to 4.
Includes	`xmk.h, sys/bufmalloc.h`

`int **bufdestroy**(membuf_t mbuf)`

Parameters	*mbuf* is the identifier of the memory pool to destroy.
Returns	0 on success. -1 on errors.
Description	This routine destroys the memory pool identified by *mbuf*.
Includes	`xmk.h, sys/bufmalloc.h`

`void* **bufmalloc**(membuf_t mbuf, size_t siz)`

Parameters	*mbuf* is the identifier of the memory pool from which to allocate memory. *size* is the size of memory block requested.
Returns	The start address of the memory block on success. `NULL` on failure and sets *errno* appropriately: *errno* is set to: • `EINVAL` if *mbuf* refers to an invalid memory buffer. • `EAGAIN` if the request cannot be satisfied.
Description	Allocate a chunk of memory from the memory pool specified by *mbuf*. If *mbuf* is `MEMBUF_ANY`, then all available memory pools are searched for the request and the first pool that has a free block of size *siz*, is used and allocated from.
Includes	`xmk.h, sys/bufmalloc.h`

`void buffree(membuf_t mbuf, void* mem)`

Parameters	*mbuf* is the identifier of the memory pool. *mem* is the address of the memory block.
Returns	None.
Description	Frees the memory allocated by a corresponding call to `bufmalloc`. If *mbuf* is MEMBUF_ANY, returns the memory to the pool that satisfied this request. If not, returns the memory to specified pool. Behavior is undefined if arbitrary values are specified for *mem*.
Includes	`xmk.h, sys/bufmalloc.h`

Software Timers

Xilkernel provides software timer functionality, for time relating processing. This module is optional and can be configured in or out. Refer to "Configuring Software Timers," page 48 for more information on customizing this module.

The following list provides a linked summary of the interfaces are available with the software timers module. You can click on a function to go to the description.

unsigned int xget_clock_ticks( )

`unsigned int xget_clock_ticks( )`

Parameters	None.
Returns	Number of kernel ticks elapsed since the kernel was started.
Description	A single tick is counted, every time the kernel timer delivers an interrupt. This is stored in a 32-bit integer and eventually overflows. The call to `xget_clock_ticks( )` returns this tick information, without conveying the overflows that have occurred.
Includes	`xmk.h, sys/timer.h`

`time_t time(time_t *timer)`

Parameters	*timer* points to the memory location in which to store the requested time information.
Returns	Number of seconds elapsed since the kernel was started.
Description	The routine time elapsed since kernel start in units of seconds. This is also subject to overflow.
Includes	`xmk.h, sys/timer.h`

`unsigned sleep(unsigned int ms)`

Parameters	`ms` is the number of milliseconds to sleep.
Returns	Number of seconds between sleeps. 0 on complete success.
Description	This routine causes the invoking process to enter a sleep state for the specified number of milliseconds.
Includes	`xmk.h, sys/timer.h`

Interrupt Handling

Xilkernel abstracts away primary interrupt handling requirements from the user application. Even though the kernel is functional without any interrupts, the system only makes sense when it is driven by at least one timer interrupt for scheduling. The kernel handles the main timer interrupt, using it as the kernel tick to perform scheduling. The timer interrupt is initialized and tied to the vectoring code during system initialization. This kernel pulse provides software timer facilities and time-related routines also. Additionally, Xilkernel can handle multiple interrupts when connected through an interrupt controller, and works with the `axi_intc` interrupt controller core. The following figure shows a basic interrupt service in Xilkernel.

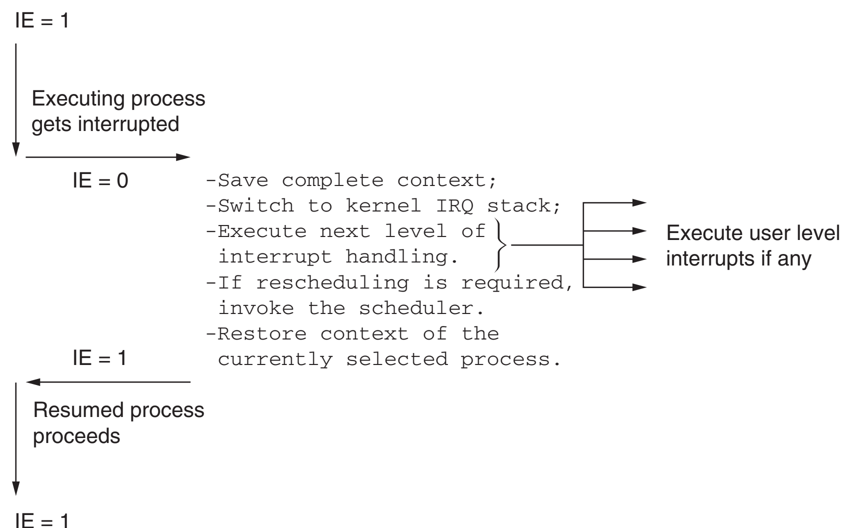

Figure 5: **Basic Interrupt Service in Xilkernel**

The interrupt handling scenario is illustrated in this diagram. Upon an interrupt:

- The context of the currently executing process is saved into the context save area.
- Interrupts are disabled from this point in time onwards, until they are enabled at the end of interrupt handling.
- This alleviates the stack burden of the process, as the execution within interrupt, does not use the user application stack.
- This interrupt context can be thought of as a special kernel thread that executes interrupt handlers in order. This thread starts to use its own separate execution stack space.
- The separate kernel execution stack is at-least 1 KB in size to enable it to handle deep levels of nesting within interrupt handlers. This kernel stack is also automatically configured to use the pthread stack size chosen by the user, if it is larger than 1 KB. If you foresee a large stack usage within your interrupt handlers, you will need to specify a large value for `pthread_stack_size`.

This ends the first level of interrupt handling by the kernel. At this point, the kernel transfers control to the second level interrupt handler. This is the main interrupt handler routine of the

interrupt controller. From this point, the handler for the interrupt controller invokes the user-specified interrupt handlers for the various interrupting peripherals.

In MicroBlaze processor kernels, if the system timer is connected through the interrupt controller, then the kernel invisibly handles the main timer interrupt (kernel tick), by registering itself as the handler for that interrupt.

Interrupt handlers can perform any kind of required interrupt handling action, including making system calls. However, the handlers must never invoke blocking system calls, or the entire kernel is blocked and the system comes to a suspended state. Use handlers wisely to do minimum processing upon interrupts.

> ***Caution!*** User level interrupt handlers must not make blocking system calls. System calls made, if any, should be non-blocking.

After the user-level interrupt handlers are serviced, the first-level interrupt handler in the kernel gets control again. It determines if the preceding interrupt handling caused a rescheduling requirement in the kernel.

If there is such a requirement, it invokes the kernel scheduler and performs the appropriate rescheduling. After the scheduler has determined the next process to execute, the context of the new process is restored and interrupts are enabled again.

When Xilkernel is used with multiple-interrupts in the system, the Xilkernel user-level interrupt handling API becomes available. The following subsection lists user-level interrupt handling APIs.

User-Level Interrupt Handling APIs

User-Level Interrupt Handling APIs Function Summary

The following list provides a linked summary of the user-level interrupt handling APIs in Xilkernel. You can click on a function to go to the description.

unsigned int register_int_handler(int_id_t id, void *handler)(void*), void *callback)
void unregister_int_handler(int_id_t id)
void enable_interrupt(int_id_t id)
void disable_interrupt(int_id_t id)
void acknowledge_interrupt(int_id_t id)

User-Level Interrupt Handling APIs Function Descriptions

The interrupt handlings API descriptions are as follows:

`unsigned int` **`register_int_handler`**`(int_id_t` *`id`*`, void *handler)(void*), void` *`*callback`*`)`

Parameters	`id` is the zero-based numeric id of the interrupt. `handler` is the user-level handler. `callback` is a callback value that can be delivered to the user-level handler.
Returns	`XST_SUCCESS` on success. error codes defined in `xstatus.h`.
Description	The `register_int_handler( )` function registers the specified user level interrupt handler as the handler for a specified interrupt. The user level routine is invoked asynchronously upon being serviced by an interrupt controller in the system. The routine returns an error on MicroBlaze processor systems if `id` is the identifier for the system timer interrupt. PowerPC processor systems have a dedicated hardware timer interrupt that exists separately from the other interrupts in the system. Therefore, this check is not performed for a PowerPC processor system.
Includes	`xmk.h, sys/intr.h`

`void` **`unregister_int_handler`**`(int_id_t` *`id`*`)`

Parameters	`id` is the zero-based numeric id of the interrupt.
Returns	None.
Description	The `unregister_int_handler( )` function unregisters the registered user-level interrupt handler as the handler for the specified interrupt. The routine does nothing and fails silently on MicroBlaze processor systems if `id` is the identifier for the system timer interrupt.
Includes	`xmk.h, sys/intr.h`

`void enable_interrupt(int_id_t id)`

Parameters	`id` is the zero-based numeric id of the interrupt.
Returns	None.
Description	The `enable_interrupt( )` function enables the specified interrupt in the interrupt controller. The routine does nothing and fails silently on MicroBlaze processor systems, if `id` is the identifier for the system timer interrupt.
Includes	`xmk.h, sys/intr.h`

`void disable_interrupt(int_id_t id)`

Parameters	`id` is the zero-based numeric id of the interrupt.
Returns	None.
Description	The `disable_interrupt( )` function disables the specified interrupt in the interrupt controller. The routine does nothing and fails silently on MicroBlaze processor systems if `id` is the identifier for the system timer interrupt.
Includes	`xmk.h, sys/intr.h`

`void acknowledge_interrupt(int_id_t id)`

Parameters	`id` is the zero-based numeric identifier of the interrupt.
Returns	None.
Description	The `acknowledge_interrupt( )` function acknowledges handling the specified interrupt to the interrupt controller. The routine does nothing and fails silently on MicroBlaze processor systems if `id` is the identifier for the system timer interrupt.
Includes	`xmk.h, sys/intr.h`

Exception Handling

Xilkernel handles exceptions for the MicroBlaze processor, treating them as faulting conditions by the executing processes/threads. Xilkernel kills the faulting process and reports using a message to the console (if verbose mode is on) as to the nature of the exception. You cannot register your own handlers for these exceptions and Xilkernel handles them all natively.

Xilkernel does *not* handle exceptions for the PowerPC processor. The exception handling API and model that is available for the Standalone platform is available for Xilkernel. You might want to register handlers or set breakpoints (during debug) for exceptions that are of interest to you.

Memory Protection

Memory protection is an extremely useful feature that can increase the robustness, reliability, and fault tolerance of your Xilkernel-based application. Memory protection requires support from the hardware. Xilkernel is designed to make use of the MicroBlaze Memory Management (Protection) Unit (MMU) features when available. This allows you to build fail-safe applications that each run within the valid sandbox of the system, as determined by the executable file and available I/O devices.

Note: Full virtual memory management is not supported by Xilkernel. Even when a full MMU is available on a MicroBlaze processor, only transparent memory translations are used, and there is no concept of demand paging.

Note: Xilkernel does not support the Memory Protection feature on PowerPC processors.

Memory Protection Overview

When the MicroBlaze parameter `C_USE_MMU` is set to >=2, the kernel configures in memory protection during startup automatically.

Note: To disable the memory protection in the kernel, add the compiler flag `-D XILKERNEL_MB_MPU_DISABLE`, to your library and application build.

The kernel identifies three types of protection violations:

1. **Code violation** — occurs when a thread tries to execute from memory that is not defined to contain program instructions.

   Note: Because Xilkernel is a single executable, all threads have access to all program instructions and the kernel cannot trap violations where a thread starts executing the kernel code directly.

2. **Data access violation** — Occurs when a thread tries to read or write data to or from memory that is not defined to be a part of the program data space. Similarly, read-only data segments can be protected by write access by all threads.

   Note: Because Xilkernel is a single executable, all threads have equal access to all data as well as the kernel data structures. The kernel cannot trap violations where a thread accesses data that it is not designated to handle.

3. **I / O violation** — occurs when a thread tries to read or write from memory-mapped peripheral I / O space that is not present in the system.

Xilkernel attempts to determine these three conceptual protection areas in your program and system during system build and kernel boot time automatically. The kernel attempts to identify code and data labels that demarcate code and data sections in your executable ELF file. These labels are typically provided by linker scripts.

For example, MicroBlaze linker scripts use the labels *`_ftext`* and *`_etext`* to indicate the beginning and the end of the `.text` section respectively.

The following table summarizes the logical sections that must be present in the linker script, the requirements on the alignment of each section, and the demarcating labels.

Table 1: **Linker Script Logical Sections**

Section	Start Label	End Label	Description
`.text`	*`_ftext`*	*`_etext`*	Executable instruction sections
`.data`	*`_fdata`*	*`_edata`*	Read-write data sections including small data sections
`.rodata`	*`_frodata`*	*`_erodata`*	Read only data sections including small data sections
`.stack`	*`_stack_end`*	*`_stack`*	Kernel stack with 1 KB guard page above and below
stack guard page (top)	*`_fstack_guard_top`*	*`_estack_guard_top`*	Top kernel stack guard page (1 KB)
stack guard page (bottom)	*`_fstack_guard_bottom`*	*`_estack_guard_bottom`*	Bottom kernel stack guard page (1 KB)

Each section must be aligned at 1 KB boundary and clearly demarcated by the specified labels. Otherwise, Xilkernel will ignore the missing logical sections with no error or warning message.

Caution! This behavior could manifest itself in your software not working as expected, because MPU translation entries will be missing for important ELF sections and the processor will treat valid requests as invalid.

Note: Each section typically has a specific type of data that is expected to be present. If the logic of the data inserted into the sections by your linker script is inappropriate, then the protection offered by the kernel could be incorrect or the level of protection could be diluted.

I/O ranges are automatically enumerated by the library generation tools and provided as a data structure to the kernel. These peripheral I/O ranges will not include read/write memory areas because the access controls for memory are determined automatically from the ELF file. During kernel boot, the enumerated I/O ranges are marked as readable and writable by the threads. Accesses outside of the defined I/O ranges causes a protection fault.

User-specified Protection

In addition to the automatic inference and protection region setup done by the kernel, you can provide your own protection regions by providing the data structures as shown in the following example. If this feature is not required, these data structures can be removed from the application code.

```
#include <mpu.h>

int user_io_nranges = 2;
xilkernel_io_range_t user_io_range[1] = {{0x25004000, 0x25004fff,
MPU_PROT_READWRITE},
                 {0x44000000, 0x44001fff, MPU_PROT_NONE}};
```

The `xilkernel_io_ranges_t` type is defined as follows:

```
typedef struct xilkernel_io_range_s {
    unsigned int baseaddr;
    unsigned int highaddr;
    unsigned int flags;
} xilkernel_io_range_t;
```

The following table lists the valid field flags that identify the user-specified access protection options:

Table 2: **Access Protection Field Flags**

Field Flag	Description
`MPU_PROT_EXEC`	Executable program instructions (no read or write permissions)
`MPU_PROT_READWRITE`	Readable and writable data sections (no execute permissions)
`MPU_PROT_READ`	Read-only data sections (no write/execute permissions)
`MPU_PROT_NONE`	(Currently no page can be protected from all three accesses at the same time. This field flag is equivalent to `MPU_PROT_READ`)

Fixed Unified Translation Look-aside Buffer (TLB) Support on the MicroBlaze Processor

The MicroBlaze processor has a fixed 64-entry Unified Translation Look-aside Buffer (TLB). Xilkernel can support up to this maximum number of TLBs only. If the maximum TLBs to enable protection for a given region are exceeded, Xilkernel will report an error during Microprocessor Unit (MPU) initialization and proceed to boot the kernel without memory protection. There is no support for dynamically swapping TLB management to provide an arbitrary number of protection regions.

Other Interfaces

Internally, Xilkernel, depends on the Standalone platform; consequently, the interfaces that the Standalone presents are inherited by Xilkernel. Refer to the “Standalone” document for information on available interfaces.

Hardware Requirements

Xilkernel is completely integrated with the software platform configuration and automatic library generation mechanism. As a result, a software platform based on Xilkernel can be configured and built in a matter of minutes. However, some services in the kernel require support from the hardware. Scheduling and all the dependent features require a periodic kernel tick and typically some kind of timer is used. Xilkernel has been designed to work with the `axi_timer` IP core. By specifying the instance name of the timer device in the software platform configuration, Xilkernel is able to initialize and use the timer cores and timer related services automatically. Refer to “Configuring System Timer,” page 48 for more information on how to specify the timer device.

Xilkernel has also been designed to work in scenarios involving multiple-interrupting peripherals. The `axi_intc` IP core handles the hardware interrupts and feeds a single IRQ line from the controller to the processor. By specifying the name of the interrupt controller peripheral in the software platform configuration, you would be getting kernel awareness of multiple interrupts. Xilkernel would automatically initialize the hardware cores, interrupt system, and the second level of software handlers as a part of its startup. You do not have to do this manually. Xilkernel handles non-cascaded interrupt controllers; cascaded interrupt controllers are not supported.

System Initialization

The entry point for the kernel is the `xilkernel_main( )` routine defined in `main.c`. Any user initialization that must be performed can be done before the call to `xilkernel_main( )`. This includes any system-wide features that might need to be enabled before invoking `xilkernel_main( )`. These are typically machine-state features such as cache enablement or hardware exception enablement that must be "always ON" even when context switching between applications. Make sure to set up such system states before invoking `xilkernel_main( )`. Conceptually, the `xilkernel_main( )` routine does two things: it initializes the kernel via `xilkernel_init( )` and then starts the kernel with `xilkernel_start( )`. The first action performed within `xilkernel_init( )` is kernel-specific hardware initialization. This includes registering the interrupt handlers and configuring the system timer, as well as memory protection initialization. Interrupts/exceptions are not enabled after completing `hw_init( )`. The `sys_init( )` routine is entered next.

The `sys_init( )` routine performs initialization of each module, such as processes and threads, initializing in the following order:

1. Internal process context structures
2. Ready queues
3. pthread module
4. Semaphore module
5. Message queue module
6. Shared memory module
7. Memory allocation module
8. Software timers module
9. Idle task creation
10. Static pthread creation

After these steps, `xilkernel_start( )` is invoked where interrupts and exceptions are enabled. The kernel loops infinitely in the *idle task*, enabling the scheduler to start scheduling processes.

Thread Safety and Re-Entrancy

Xilkernel, by definition, creates a multi-threaded environment. Many library and driver routines might not be written in a thread-safe or re-entrant manner. Examples include the C library routines such as `printf( )`, `sprintf( )`, `malloc( )`, `free( )`. When using any library or driver API that is not a part of Xilkernel, you must make sure to review thread-safety and re-entrancy features of the routine. One common way to prevent incorrect behavior with unsafe routines is to protect entry into the routine with locks or semaphores.

Restrictions

The MicroBlaze processor compiler supports a `-mxl-stack-check` switch, which can be used to catch stack overflows. However, this switch is meant to work only with single-threaded applications, so it cannot be used in Xilkernel.

Kernel Customization

Xilkernel is highly customizable. As described in previous sections, you can change the modules and individual parameters to suit your application. The SDK **Board Support Package Settings** dialog box provides an easy configuration method for Xilkernel parameters. Refer to the "Embedded System and Tools Architecture Overview" chapter in the *Embedded Systems Tools Reference Manual (UG111)* for more details. To customize a module in the kernel, a parameter with the name of the category set to `TRUE` must be defined in the Microprocessor Software Specification (MSS) file. An example for customizing the pthread is shown as follows:

```
parameter config_pthread_support = true
```

If you do not define a configurable *`config_`* parameter for the module, that module is not implemented. You do not have to manually key in these parameters and values. When you input information in the **Board Support Package Settings** dialog box, SDK generates the corresponding Microprocessor Software Specification (MSS) file entries automatically.

The following is an MSS file snippet for configuring OS Xilkernel for a PowerPC processor system. The values in the snippet are sample values that target a hypothetical board:

```
BEGIN OS
PARAMETER OS_NAME = xilkernel
PARAMETER OS_VER = 5.02.a
PARAMETER STDIN = RS232
PARAMETER STDOUT = RS232
PARAMETER proc_instance = microblaze0
PARAMETER config_debug_support = true
PARAMETER verbose = true
PARAMETER systmr_spec = true
PARAMETER systmr_freq = 100000000
PARAMETER systmr_interval = 80
PARAMETER sysintc_spec = system_intc
PARAMETER config_sched = true
PARAMETER sched_type = SCHED_PRIO
PARAMETER n_prio = 6
PARAMETER max_readyq = 10
PARAMETER config_pthread_support = true
PARAMETER max_pthreads = 10
PARAMETER config_sema = true
PARAMETER max_sem = 4
PARAMETER max_sem_waitq = 10
PARAMETER config_msgq = true
PARAMETER num_msgqs = 1
PARAMETER msgq_capacity = 10
PARAMETER config_bufmalloc = true
PARAMETER config_pthread_mutex = true
PARAMETER config_time = true
PARAMETER max_tmrs = 10
PARAMETER enhanced_features = true
PARAMETER config_kill = true
PARAMETER mem_table = ((4,30),(8,20))
PARAMETER static_pthread_table = ((shell_main,1))
END
```

The configuration parameters in the MSS specification impact the memory and code size of the Xilkernel image. Kernel-allocated structures whose count can be configured through the MSS must be reviewed to ensure that your memory and code size is appropriate to your design.

For example, the maximum number of process context structures allocated in the kernel is determined by the sum of two parameters; `max_procs` and `max_pthreads`. If a process context structures occupies *x* bytes of `bss` memory, then the total `bss` memory requirement for process contexts is `(max_pthreads*x)` bytes. Consequently, such parameters must be tuned carefully, and you need to examine the final kernel image with the GNU size utility to ensure that your memory requirements are met. To get an idea the contribution each kernel-allocated structure makes to memory requirements, review the corresponding header file. The specification in the MSS is translated by Libgen and Xilkernel Tcl files into C-language configuration directives in two header files: `os_config.h` and `config_init.h`. Review these two files, which are generated in the main processor include directory, to understand how the specification gets translated.

Configuring STDIN and STDOUT

The standard input and output peripherals can be configured for Xilkernel. Xilkernel can work without a standard input and output also. These peripherals are the targets of input-output APIs like `print`, `outbyte`, and `inbyte`. The following table provides the attribute descriptions, data types, and defaults.

Table 3: **STDIN/STDOUT Configuration Parameters**

Attribute	Description	Type	Defaults
`stdin`	Instance name of stdin peripheral.	string	none
`stdout`	Instance name of stdout peripheral.	string	none

Configuring Scheduling

You can configure the kernel scheduling policy by configuring the parameters shown in the following table.

Table 4: **Scheduling Parameters**

Attribute	Description	Type	Defaults
`config_sched`	Configure scheduler module.	boolean	true
`sched_type`	Type of Scheduler to be used. Allowed values: 2 - `SCHED_RR` 3 - `SCHED_PRIO`	enum	`SCHED_RR`
`n_prio`	Number of priority levels if scheduling is `SCHED_PRIO`.	numeric	32
`max_readyq`	Length of each ready queue. This is the maximum number of processes that can be active in a ready queue at any instant in time.	numeric	10

Configuring Thread Management

Threads are the primary mechanism for creating process contexts. The configurable parameters of the thread module are listed in the following table.

Table 5: **Thread Module Parameters**

Attribute	Description	Type	Defaults
`config_pthread_support`	Need pthread module.	boolean	true
`max_pthreads`	Maximum number of threads that can be allocated at any point in time.	numeric	10
`pthread_stack_size`	Stack size for dynamically created threads (in bytes).	numeric	1000

Table 5: **Thread Module Parameters *(Cont'd)***

Attribute	Description	Type	Defaults
`static_pthread_table`	Statically configure the threads that startup when the kernel is started. This is defined to be an array with each element containing the parameters `pthread_start_addr` and `pthread_prio`. ***Note:*** If you are specifying function names for `pthread_start_addr`, they must be functions in your source code that are compiled with the C dialect. They *cannot* be functions compiled with the C++ dialect.	array of 2-tuples	none
`pthread_start_addr`	Thread start address.	Function name (string)	none
`pthread_prio`	Thread priority.	numeric	none

Configuring Semaphores

You can configure the semaphores module, the maximum number of semaphores, and semaphore queue length. The following table shows the parameters used for configuration.

Table 6: **Semaphore Module Parameters**

Attribute	Description	Type	Defaults
`config_sema`	Need Semaphore module.	boolean	false
`max_sem`	Maximum number of Semaphores.	numeric	10
`max_sem_waitq`	Semaphore Wait Queue Length.	numeric	10
`config_named_sema`	Configure named semaphore support in the kernel.	boolean	false

Configuring Message Queues

Optionally, you can configure the message queue module, number of message queues, and the size of each message queue. The message queue module depends on both the semaphore module and the buffer memory allocation module. The following table shows the parameter definitions used for configuration. Memory for messages must be explicitly specified in the *`malloc`* customization or created at run-time.

Table 7: **Message Queue Module Parameters**

Attribute	Description	Type	Defaults
`config_msgq`	Need Message Queue module.	boolean	false
`num_msgqs`	Number of message queues in the system.	numeric	10
`msgq_capacity`	Maximum number of messages in the queue.	numeric	10
`use_malloc`	Provide for more powerful message queues which use *`malloc`* and *`free`* to allocate memory for messages.	boolean	false

Configuring Shared Memory

Optionally, you can configure the shared memory module and the size of each shared memory segment. All the shared memory segments that are needed must be specified in these parameters.The following table shows the parameters used for configuration.

Table 8: **Shared Memory Module Parameters**

Attribute	Description	Type	Defaults
`config_shm`	Need shared memory module.	boolean	false
`shm_table`	Shared memory table. Defined as an array with each element having `shm_size` parameter.	array of 1-tuples	none
`shm_size`	Shared memory size.	numeric	none
`num_shm`	Number of shared memories expressed as the `shm_table` array size.	numeric	none

Configuring Pthread Mutex Locks

Optionally, you can choose to include the pthread mutex module, number of mutex locks, and the size of the wait queue for the mutex locks. The following table shows the parameters used for configuration.

Table 9: **Pthread Mutex Module Parameters**

Attribute	Description	Type	Defaults
`config_pthread_mutex`	Need pthread mutex module.	boolean	false
`max_pthread_mutex`	Maximum number of pthread mutex locks available in the system.	numeric	10
`max_pthread_mutex_ waitq`	Length of each the mutex lock wait queue.	numeric	10

Configuring Buffer Memory Allocation

Optionally, you can configure the dynamic buffer memory management module, size of memory blocks, and number of memory blocks. The following table shows the parameters used for configuration.

Table 10: **Memory Management Module Parameters**

Attribute	Description	Type	Defaults
`config_bufmalloc`	Need buffer memory management.	boolean	false
`max_bufs`	Maximum number of buffer pools that can be managed at any time by the kernel.	numeric	10
`mem_table`	Memory block table. This is defined as an array with each element having `mem_bsize`, `mem_nblks` parameters.	array of 2-tuples	none
`mem_bsize`	Memory block size in bytes.	numeric	none
`mem_nblks`	Number of memory blocks of a size.	numeric	none

Configuring Software Timers

Optionally, you can configure the software timers module and the maximum number of timers supported. The following table shows the parameters used for configuration.

Table 11: **Software Timers Module Parameters**

Attribute	Description	Type	Defaults
config_time	Need software timers and time management module.	boolean	false
max_tmrs	Maximum number of software timers in the kernel.	numeric	10

Configuring Enhanced Interfaces

Optionally, you can configure some enhanced features/interfaces using the following parameters shown in the following table.

Table 12: **Enhanced Features**

Attribute	Description	Type	Defaults
config_kill	Include the ability to kill a process with the kill( ) function.	boolean	false
config_yield	Include the yield( ) interface.	boolean	false

Configuring System Timer

You can configure the timer device in the system for MicroBlaze processor kernels. Additionally, you can configure the timer interval for PowerPC and PIT timer based MicroBlaze processor systems. The following table shows the available parameters .

Table 13: **Attributes for Copying Kernel Source Files**

Attribute	Description	Type	Defaults
systmr_dev[1]	Instance name of the system timer peripheral.	string	none
systmr_freq	Specify the clock frequency of the system timer device. For the axi_timer, it is the frequency of the AXI bus to which the axi_timer is connected.	numeric	100000000
systmr_interval	Time interval per system timer interrupt.	numeric (milliseconds)	10

1. MicroBlaze only.

Configuring Interrupt Handling

You can configure the interrupt controller device in the system kernels. Adding this parameter automatically configures multiple interrupt support and the user-level interrupt handling API in the kernel. This also causes the kernel to automatically initialize the interrupt controller. The following table shows the implemented parameters.

Table 14: **Attributes for Copying Kernel Source Files**

Attribute	Description	Type	Defaults
`sysintc_spec`	Specify the instance name of the interrupt controller device connected to the external interrupt port.	string	null

Configuring Debug Messages

You can configure that the kernel outputs debug/diagnostic messages through its execution flow. Enabling the parameter in the following table makes the DBG_PRINT macro available, and subsequently its output to the standard output device:

Table 15: **Attribute for Debug Messages**

Attribute	Description	Type	Defaults
`debug_mode`	Turn on kernel debug messages.	boolean	false

Coping Kernel Source Files

You can copy the configured kernel source files to your repository for further editing and use them for building the kernel. The following table shows the implemented parameters:

Table 16: **Attributes for Copying Kernel Source Files**

Attribute	Description	Type	Defaults
`copyoutfiles`	Need to copy source files.	boolean	false
`copytodir`	User repository directory. The path is relative to *project_directory*/*system_name*/libsrc/xilkernel_v6_1/*src_dir*.	path string	"../*copyoflib*"

Debugging Xilkernel

The entire kernel image is a single file that can serve as the target for debugging with the SDK GNU Debugger (GDB) mechanism. User applications and the library must be compiled with a **-g**. Refer to the *Embedded System Tools Reference Manual (UG111)* for documentation on how to debug applications with GDB.

Note: This method of debugging involves great visibility into the kernel and is intrusive. Also, this debugging scheme is *not* kernel-user application aware.

Memory Footprint

The size of Xilkernel depends on the user configuration. It is small in size and can fit in different configurations. The following table shows the memory size output from GNU size utility for the kernel. Xilkernel has been tested with the GNU Compiler Collection (GCC) optimization flag of -O2; the numbers in the table are from the same optimization level.

Table 17: **User Configuration and Xilkernel Size**

Configuration	MicroBlaze (in kb)	PowerPC (in kb)
Basic kernel functionality with multi-threading only.	7	16
Full kernel functionality with round-robin scheduling (no multiple interrupt support and no enhanced features).	16	26
Full kernel functionality with priority scheduling (no multiple interrupt support and no enhanced features).	16.5	26.5
Full kernel functionality with all modules (threads, support for both ELF processes, priority scheduling, IPC, synchronization constructs, buffer malloc, multiple and user level interrupt handling, drivers for interrupt controller and timer, enhanced features).	22	32

Xilkernel File Organization

Xilkernel sources are organized as shown in the table below:

Table 18: **Organization of Xilkernel Sources**

`root/`				Contains the `/data` and the `/src` folders.
	`data/`			Contains Microprocessor Library Definition (MLD) and Tcl files that determine XilKernel configuration.
	`src/`			Contains all the source.
		`include/`		Contains header files organized similar to `/src`.
		`src/`		Non-header source files.
			`arch/`	Architecture-specific sources.
			`sys/`	System-level sources.
			`ipc/`	Sources that implement the IPC functionality.

Modifying Xilkernel

You can further customize Xilkernel by changing the actual code base. To work with a custom copy of Xilkernel, you must first copy the Xilkernel source folder `xilkernel_v6_1` from the SDK installation and place it in a software repository; for example, `<..../mylibraries/bsp/xilkernel_v6_1>`. If the repository path is added to the tools, Libgen picks up the source folder of Xilkernel for compilation.

Refer to "Xilkernel File Organization," page 50 for more information on the organization of the Xilkernel sources. Xilkernel sources have been written in an elementary and intuitive style and include comment blocks above each significant function. Each source file also carries a comment block indicating its role.

Deprecated Features

ELF Process Management (Deprecated)

A deprecated feature of Xilkernel is the support for creating execution contexts out of separate Executable Linked Files (ELFs).

You might do this if you need to create processes out of executable files that lay on a file system (such as XilFATFS or XilMFS). Typically, a loader is required, which Xilkernel does not provide. Assuming that your application does involve a loader, then given a entry point in memory to the executable, Xilkernel can then create a process. The kernel does not allocate a separate stack for such processes; the stack is set up as a part of the CRT of the separate executable.

Note: Such separate executable ELF files, which are designed to run on top of Xilkernel, must be compiled with the compiler flag `-xl-mode-xilkernel` for MicroBlaze processors. For PowerPC processors, you must use a custom linker script, that does not include the `.boot` and the `.vectors` sections in the final ELF image. The reason that these modifications are required is that, by default, any program compiled with the SDK GNU tool flow, could potentially contain sections that overwrite the critical interrupt, exception, and reset vectors section in memory. Xilkernel requires that its own ELF image initialize these sections and that they stay intact. Using these special compile flags and linker scripts, removes these sections from the output image for applications.

The separate executable mode has the following caveats:

- Global pointer optimization is not supported.

  Note: This is supported in the default kernel linkage mode. It is not supported only in this separate executable mode.

- Xilkernel does not feature a loader when creating new processes and threads. It creates process and thread contexts to start of from memory images assumed to be initialized. Therefore, if any ELF file depends on initialized data sections, then the next time the same memory image is used to create a process, the initialized sections are invalid, unless some external mechanism is used to reload the ELF image before creating the process.

Note: This feature is deprecated. Xilinx encourages use of the standard, single executable file application model.

Refer to the "Configuring ELF Process Management (Deprecated)," page 52 for more details. An ELF process is created and handled using the following interfaces.

`int` **`elf_process_create`**`(void* `*`start_addr`*`, int `*`prio`*`)`

Parameters	*`start_addr`* is the start address of the process. *`prio`* is the starting priority of the process in the system.
Returns	• The PID of the new process on success. • -1 on failure.
Description	Creates a new process. Allocates a new PID and Process Control Block (PCB) for the process.The process is placed in the appropriate ready queue.
Includes	`xmk.h, sys/process.h`

`int elf_process_exit(void)`

Parameters	None.
Returns	None.
Description	Removes the process from the system. ***Caution!*** Do not use this function to terminate a thread.
Includes	`xmk.h, sys/process.h`

Configuring ELF Process Management (Deprecated)

You can select the maximum number of processes in the system and the different functions needed to handle processes. The processes and threads that are present in the system on system startup can be configured statically. The following table provides a list of available parameters:

Table 19: **Process Management Parameters**

Attribute	Description	Type	Defaults
`config_elf_process`	Need ELF process management. ***Note:*** Using `config_elf_process` requires enhanced_features=true in the kernel configuration.	boolean	true
`max_procs`	Maximum number of processes in the system.	numeric	10
`static_elf_process_table`	Configure startup processes that are separate executable files. This is defined to be an array with each element containing the parameters `process_start` and `process_prio`.	Array of 2-tuples	none
`process_start_addr`	Process start address.	Address	none
`process_prio`	Process priority.	Numeric	none