Open Kernel Labs™
Be open. Be safe.

OKL4 Microkernel Reference Manual

API Version 03_{16}

Document Number: OK 10000:2006 (revision 12)
Software Version: 3.0
Date: September 11, 2008

This publication is distributed by Open Kernel Labs Pty Ltd, Australia.

THIS DOCUMENT IS PROVIDED “AS IS” WITHOUT ANY WARRANTIES, INCLUDING ANY WARRANTY OF MERCHANTABILITY, NON-INFIRNGEMENT, FITNESS FOR ANY PARTICULAR PURPOSE, OR ANY WARRANTY OTHERWISE ARISING OF ANY PROPOSAL, SPECIFICATION OR SAMPLE.

Authors:

Open Kernel Labs

Contact Details:

Open Kernel Labs Pty Ltd
Attention: Open Kernel Labs

Suite 3, 540 Botany Road
Alexandria, NSW 2015
Australia

email: **enquiries@ok-labs.com**
web: **http://www.ok-labs.com/**

Contents

PART A — GENERIC INTERFACE

PART B — COMMON BINARY INTERFACE

PART C — ARCHITECTURE BINDINGS

PART D — LANGUAGE BINDINGS

Preface

Open Kernel Labs L4 microkernel (OKL4) is a flexible, high performance microkernel suitable for many systems from embedded through to enterprise. This manual provides a detailed description of the OKL4 microkernel and its API.

The *OKL4 Microkernel Programming Manual* covers all features of the OKL4 microkernel at four different levels of abstraction and defines the computing concepts relevant to programming in an OKL4 environment.

It should be noted that this manual is not intended as a formal specification of OKL4 as clarity of description has been given preference over mathematical strictness. However, efforts are underway at NICTA to produce formal semantics of the L4 microkernel. It is intended that the description provided herein be as complete and unambiguous as possible without resorting to rigorous mathematical models of the microkernel's operation.

1 Introduction

OKL4 is a highly flexible, general purpose, high performance microkernel. It provides a minimal layer of hardware abstraction on which various *operating system personalities* can be built using modules. It isolates each component in the system from effects of programming errors or malicious code contained in other components. OKL4 applies the principle of *minimality with respect to security*. As such, a service (feature) is included in the microkernel only if it is impossible to provide that service outside the microkernel without sacrificing security, or if including the feature in the microkernel provides significant benefits without increasing the complexity of the microkernel.

OKL4 allows for trust and security models to be implemented using the OKL4 API. OKL4 uses hardware and software provided mechanisms to enforce these trust and security models by providing time, resource/memory and communication protection and *fault isolation*.

A system can be divided into *components* which abstract over shared system resources. The microkernel confines the effects of a programming error within a component or a malicious attempt by a component to violate the integrity of other components in the system to the misbehaving component. Though a misbehaving component may attempt to interact with other components in the system, the microkernel provides mechanisms to isolate components from the effects of misbehaviour within other components. Note that as the microkernel does not perform any error recovery, error recovery and any similar behaviour remains the responsibility of the operating system personality.

Fundamentally, a component may only affect other components in the system by accessing system resources shared between those components. As such, the basic function of the microkernel is to provide abstractions that facilitate secure management of all shared resources within the system, that is, their distribution, sharing and exchange between components. OKL4 divides shared system resources into two orthogonal classes, providing a separate component abstraction for each class. *Time* is managed through the *thread* component abstraction described in Chapter A-2 and *memory* is managed through the *address space* component abstraction described in Chapter A-3.

Each thread in the system is isolated from effects of misbehaviour within other threads with respect to the use of processing time. Similarly, each address space in the system is isolated from effects of misbehaviour within other address spaces with respect to the use of memory-mapped resources such as physical memory (RAM) and hardware device registers. OKL4 provides mechanisms for managing time, memory and communication between system components. *Scheduling* facilitates the sharing of processing time between threads as described in Chapter A-5, *Address Spaces* control the sharing of memory resources as described in Chapter A-3 and *IPC* which controls messaging between threads. as described in Chapter A-6.

In order to bootstrap resource management in the system, OKL4 defines a set of *microkernel protocols* for every case of inter-component interaction where the microkernel is one of the active communication parties. These protocols are defined in Chapter A-13 of this manual. Collectively, OKL4's thread and address space abstractions, the scheduling and IPC facilities and the microkernel protocols form the OKL4 *application programming interface* (API) defined in the remainder of this document.

Some hardware architectures (most notably IA-32) provide a separate third class of system resources which represent communication channels between hardware components such as processors and devices. On IA-32, these are known as I/O ports. Under OKL4, any such system resources are treated as a special case of a memory-mapped resource through a suitable extension to the address space abstraction.

As a side-effect of this design philosophy, OKL4 provides a high degree of portability for operating system code written using its API by virtualizing many of the low-level hardware features such as the memory management hardware that are the source of most significant differences between individual hardware architectures and therefore are the cause of most of the problems when porting an operating system to a range of hardware designs.

It should be noted that portability is not one of the OKL4 design goals. The OKL4 API does not abstract over architectural differences between hardware platforms unless such an abstraction is a side-effect of its security measures, or that it may be provided without an increase in the complexity of the microkernel or a decrease of its performance.

1.1 Outline of This Manual

The remainder of this chapter summarises the basic terminology and notation used throughout the manual. Most terminology should be familiar to most readers with a system-programming background. The remainder of the manual provides a description of the OKL4 application programming interface (API) and is divided into four parts corresponding to the four levels of abstraction of the OKL4 API:

- Part A describes the basic OKL4 concepts and defines the *generic OKL4 interface* to the OKL4 microkernel without the specific details of its architecture-specific features or low level API.

- Part B describes the *common binary interface* and provides details on common parts of the OKL4 API suitable for a range of hardware architectures. In particular, the common binary interface defines most encoding details of the OKL4 data types described in Chapter A-9. It complements the language and architecture bindings defined in the remaining two parts of the manual.

- Part C describes a mapping of the OKL4 API onto each hardware architecture on which OKL4 has been ported, thus describing OKL4 API bindings for the corresponding low-level assembly language. This part may interest authors of tools such as compilers and interface definition languages that optimise code interacting with the OKL4 microkernel by generating local specialised OKL4 interface elements from their high-level description. It also contains description of OKL4 API extensions available on some hardware architectures. This part may also be of interest to system programmers familiar with a particular hardware architecture and interested in utilising a specific architectural feature in the OKL4 environment.

- Part D describes the C programming language bindings to the OKL4 microkernel API. This part is of interest both to programmers designing OKL4 operating system personalities and those writing programs for such systems. It is expected that high-level abstractions to the OKL4 API will be provided by the operating system personality, reducing the need for application developers to deal with low level microkernel interfaces.

1.2 Definitions

The following terminology is used to describe the OKL4 API concepts in this manual:

architecture-defined refers to API elements and/or behaviour that differs between individual hardware architectures.

data type (or *type*) is an interface element describing the encoding of a class of objects as a sequence of bits for storage in memory or a hardware register. A type consists of one or more *data constructors*.

data constructor (or *constructor*) describes one of the alternative pieces of information stored in an object of a particular type. A constructor consists of a name that identifies the constructor and a list of objects known as *data fields*. C programmers may find it convenient to think of a data type as a `union` containing constructors, where each data constructor is a `struct`. Note however that data types are rarely implemented as C unions containing structures, in OKL4.

data field (or *field*) describes an individual component of a data constructor. All fields in an object created with the corresponding constructor are valid simultaneously. C programmers may find it convenient to think of data fields as individual members of a `struct` type.

hardware architecture (or *architecture*) refers to the design of the particular central processing unit (CPU) on which a port of the OKL4 microkernel is intended to execute. For example: ARM, MIPS and PowerPC are three different hardware architectures.

hardware platform (or *platform*) refers to a particular hardware system construction incorporating one or more CPUs, general purpose memory and any number of peripheral devices.

hardware thread refers to a processing unit which in turn belongs to a *processing domain*, or simply *domain*. Typically a domain represents a discrete processor and a hardware thread, one of the execution contexts on the processor, however OKL4 may divide up processing units in alternative configurations.

architecture-defined refers to behaviour that may vary between different ports of the OKL4 microkernel. Such behaviour is described in the documentation C section of this manual.

object width (or *width*) refers to the minimum number of bits required to represent all valid values of an object in binary notation. An object of width *k* is said to be a *k-bit object*.

processing unit is a hardware facility for executing a sequence of machine instructions. Intuitively, a processing unit corresponds to a single general-purpose hardware execution unit such as a single-core processor or a hardware thread on a multi-core processor.

reserved refers to an interface element that may cause undefined behaviour if modified or used in a way other than permitted under a current version of the OKL4 API. The element may be assigned specific behaviour in a future version.

undefined behaviour refers to behaviour that is deemed erroneous, but not detected by the OKL4 microkernel. However, a thread that behaves in an undefined manner shall not implicitly affect the behaviour of threads executing in other address spaces.

32-bit architecture refers to a hardware architecture that uses 32-bit sized general purpose registers upon which the base instruction set operates.

64-bit architecture refers to a hardware architecture that uses 64-bit sized general purpose registers upon which the base instruction set operates.

word refers to a unit of data, of defined bit length, which equals the hardware architecture general purpose register size. ie. 32-bits on 32-bit architectures and 64-bits on 64-bit architectures.

1.3 Notation

In this manual, all OKL4 API elements are typeset in sans-serif font with individual words capitalised. All OKL4 language and architecture binding elements are typeset in `monospaced` font. Individual OKL4 API elements are typeset as follows:

DATATYPES Names of data types are typeset in a bold small-caps font.

DATACONSTRUCTORS Names of data constructors are typeset in a medium-weight small-caps font.

CONSTRUCTOR.*DataFields* Names of individual fields within a data types are typeset in a mixed-case slanted font. They usually follow the name of the API element containing the field and are separated from it by a dot.

DataFields When the name of the API element is clear from the context, it may be omitted for brevity.

InputParameters$_{\text{IN}}$ Names of input parameters to system calls are also typeset in a mixed-case slanted font and annotated with the subscript “IN”.

OutputParameters$_{\text{OUT}}$ Similarly, output parameters from system calls are annotated with the subscript “OUT” and typeset in a mixed-case slanted font.

ApiElement	All other API elements are typeset in a mixed-case medium-weight sans-serif font. The meaning of the element is made clear by its context in the description and preserved throughout the manual by ensuring that no two elements share the same name.

Numeric constants are presented in decimal, binary or hexadecimal notation as appropriate. Upper-case letters A–F are used to represent digits 10–15 in hexadecimal constants. Large numbers include a point-separator (·) for readability. Binary constants are annotated with the subscript 2 and hexadecimal constants are annotated with the subscript 16. For example: 127, $\texttt{1011101}_2$ or $\texttt{7A66}\cdot\texttt{71FF}_{16}$.

1.4 Credits

The OKL4 microkernel described in this manual reflects the outstanding research on microkernels and operating systems conducted by Jochen Liedtke. Only his vision of system design made this work possible. Jochen defined the state of art in microkernel design for nearly a decade. We thank him for his support and try to continue the work in his spirit.

Much of the information in this manual has been obtained from the source code for the current OKL4 implementation, developed at Open Kernel Labs and based on previous implementations from the University of Karlsruhe, the University of New South Wales and National ICT Australia (NICTA). These further being influenced by experience with earlier versions developed at GMD, IBM Research, and Dresden University of Technology. We thank all people involved in those projects for their outstanding engineering work.

This document was started by Patryk Zadarnowski who spent considerable time to capture information from previous documents and the L4 source code. Further contributions, editing and restructuring by: Ben Leslie, Carl van Schaik, Daniel Potts, Gernot Heiser and Sevwandi Fernando.

This document would be impossible without the prior L4 documentation effort by Carl van Schaik from NICTA and Uwe Dannowski, Joshua LeVasseur, Espen Skoglund and Volkmar Uhlig from the L4Ka Team, with contributions from Alan Au, Marcus Brinkmann, Kevin Elphinstone, Philip Derrin, Bryan Ford, Andreas Haeberlen, Hermann Härtig, Gernot Heiser, Michael Hohmuth, Trent Jaeger, Ben Leslie, Jork Löser, Frank Mehnert, Yoonho Park, Daniel Potts, Marc Salem, Sebastian Schönberg, Cristan Szmajda, Eric Tallet, Harvey Tuch, Marcus Völp, Neal Walfield, Alex Webster, Adam Wiggins, Simon Winwood, Jean Wolter.

Finally, special credits are due to Gernot Heiser for his vision of operating system teaching and the *L4 User Manual* developed by Alan Au and Ihor Kuz out of Gernot's lecture notes for his course on Advanced Operating Systems at the University of New South Wales.

PART A

Generic Interface

A-1 Overview

This part of the manual presents the *generic OKL4 interface* (API) and serves two distinct purposes. Firstly, it describes the environment available to programs running on top of the OKL4 microkernel. Secondly, the generic OKL4 API presents a portable view of the microkernel from the perspective of a user program that is independent of the programming language used and hardware architecture.

The generic OKL4 interface does not specify how the required behaviour is achieved on a particular architecture or how the API facilities are accessed using a particular programming language. As the encoding of API objects in memory is described in Part B, it is not covered in this part of the manual.

A-1.1 The OKL4 Programming Environment

The *OKL4 programming environment* provides a layer of abstraction over the underlying architecture that isolates individual system components from the effects of programming errors or misbehaviour within other components. As portability of source code is important to the implementation of robust system software, the OKL4 API attempts to minimize differences between architectures where it may be achieved without increasing the complexity of the microkernel implementation or a degradation of its performance.

In order to provide the required level of hardware abstraction, OKL4 divides computing resources into two orthogonal classes: memory and time. These abstractions together form the *OKL4 programming environment*. The primary purpose of the OKL4 programming environment is the execution of *programs* organized as one or more *tasks*, each of which runs in a separate *address space* and consists of one or more *threads*.

Address spaces implement the OKL4 memory abstraction. This facilitates the management of both physical memory and memory-mapped system resources and provides a virtualization layer that simplifies programming and enforces the fault isolation policy. Address spaces are further described in Chapter A-3.

OKL4 threads can be viewed as a sequence of machine instructions operating on a *virtual-state* composed of a set of *virtual registers* defined in Chapter A-11. Each thread has a separate copy of all virtual registers which may be accessed and modified both synchronously by the sequence of machine instructions in the thread and asynchronously by other threads using the OKL4 system calls, within the set of constraints designed to enforce the OKL4 fault isolation policy described in Chapter A-12. OKL4 threads are further described in Chapter A-2.

OKL4 facilitates communication between threads via the use of the *inter-process communication* (IPC) mechanism described in Chapter A-6. This mechanism is implemented by the Ipc system call described in Section A-12.5. The OKL4 IPC facility allows the exchange of short messages comprising of zero or more *words* between two threads. The message exchange is always unbuffered and synchronous and both the sender and receiver must explicitly indicate their willingness to participate, using an appropriate invocation of the Ipc system call. This makes the OKL4 IPC mechanism extremely light-weight despite its flexibility, which is the key to the high performance of the OKL4 microkernel. This design of IPC also allows the Ipc system call to be used as a powerful synchronization mechanism. The OKL4 IPC mechanism is also used to abstract over many cases of asynchronous communication including *page faults* described in Section A-3.2 and *hardware exceptions* described in Section A-2.3.

In addition, threads in *overlapping address spaces* may communicate through memory objects mapped into the shared portion of the address space. Two address spaces are said to *overlap* if they are the same address space, or if a portion of each address space has been mapped to the same physical memory using the MapControl system call described in Section A-12.6.

Effects of time-related misbehaviour within one thread are confined to the execution of that thread using OKL4's pre-emptive scheduler described in Chapter A-5.

A-1.2 OKL4 API Elements

The OKL4 API consists of a number of *interface elements* that allow user programs to access features of the OKL4 microkernel. These elements may be grouped into the following categories:

- *Address spaces* and *threads*, described in Chapters A-3 and A-2, respectively. The scheduling algorithm designed to enforce secure management of processing time described in Chapter A-5 and the inter-process communication (IPC) mechanism described in Chapter A-6.
- *Data types*, *data constructors* and *data fields* are abstract API elements used to simplify the presentation of the OKL4 programming interface in this manual, and are further described in Chapter A-9.
- *System parameters* described in Chapter A-10 capture several cases of implementation-defined behaviour within the API.
- *Virtual registers* represent individual components of the state of the automata formed by a thread and are covered in Chapter A-11.
- *System calls* provide the mechanisms for all inter-process communication in OKL4, including specialized instances of communication such as scheduling and address space configuration. System calls are further described in Chapter A-12.
- *Communication protocols* are established by OKL4 for cases of IPC where the microkernel plays an active role by acting as, or on behalf of one of the participating threads. OKL4 communication protocols are described in Chapter A-13.

The way in which API elements are accessed by a user program is dependent on the architecture, OKL4 implementation and programming language and is described separately in Parts C and D of this manual.

A-1.3 Compatibility between OKL4 Versions

An OKL4 program can be made upwards-compatible with future versions of the OKL4 microkernel by observing the guidelines described in Section A-1.5, provided that the future version has been defined to be backwards-compatible with the current version. A version can be made backwards-compatible by ensuring that its implementation of the architecture and language bindings to the OKL4 API is a superset of the older version. The OKL4 API described in this manual specifies many undefined elements which may be used to extend the OKL4 API in a backwards-compatible manner.

An OKL4 program can be made backwards-compatible with a different OKL4 version on the same architecture by providing an alternative implementation for each supported value of the system parameters described in Chapter A-10. Since many operating system personalities are intended for specialized and highly-configurable embedded environments, it is often appropriate to abort the program on detecting an unsupported OKL4 implementation.

A-1.4 Variants of the OKL4 API

While the OKL4 API is highly portable and largely abstracts over details of a particular architecture, some portability issues must be recognized in the architecture-neutral parts of this manual. This is a result of the *minimality with respect to security* principle as abstracting over these architectural differences would increase the complexity of the microkernel without improving fault isolation.

Some architectures do not provide the hardware facilities required for an efficient implementation of the OKL4 API in its full generality. For example, many architectures do not support execute-only page protection attributes. These issues are discussed in the description of the relevant OKL4 features. In addition each language binding inadvertently enforces its own view of the OKL4 API, often introducing convenience interfaces as described in Part D of this manual.

For each architecture, an OKL4 implementation must map the OKL4 microkernel abstractions onto the hardware-specific features in conformance with the architectural bindings described in Part C. Architectural bindings may be used directly in situations where the high-level language bindings are not appropriate, thus exposing architectural variations of the OKL4 API. In addition some hardware features can be utilized in the OKL4 environment using various implementation-defined extensions, further described in Part C of this manual.

The OKL4 microkernel may introduce extensions to the current API. The impact of future extensions on programs that use the current version of the API is further discussed in Section A-1.5.

A-1.5 Microkernel Extensions and Compatibility

Future versions of the OKL4 microkernel may introduce extensions and modifications to the current API that may affect software designed using the generic version of the microkernel. To minimize incompatibility issues, each program running in the OKL4 environment may choose to observe the following guidelines:

- When examining the content of a virtual register or an output parameter returned from an OKL4 system call, it should not rely on the contents of any fields marked as *reserved* in this manual. These parameters may hold information relevant to implementation-defined extensions and may be ignored by programs that do not use the functionality of these extensions.

- When writing to a virtual register or setting an input parameter on an OKL4 system call, a program should set any fields marked as *constant* to the value specified in this manual. Setting it to any other value may result in undefined operation or enable an implementation-defined extension.

- Further restrictions on the portable use of the OKL4 API are imposed by the generic OKL4 interface in the relevant sections of this manual and should be followed even when it is relaxed in a particular OKL4 implementation.

- Programs using high-level language bindings should observe the language-specific restrictions described in Part D of this manual.

- Lastly, each program should observe the architecture-specific restrictions described in Part C of this manual.

A-2 Threads

OKL4 provides *threads* which represent a context of execution within a program. Threads can be used together with scheduling to manage the sharing of processing time. An OKL4 system consists of a fixed set of thread identifiers, each of which may be used or available. A unique thread identifier is allocated to each thread, and each thread is associated with an address space as described in Chapter A-3. Threads are created and deleted using the ThreadControl system call described in Section A-12.14. Thread creation and deletion may also be referred to as "thread activation" and "thread deactivation" respectively. In OKL4, threads serve three main purposes:

- Threads represent the active parties during communication between different address spaces, or between user programs and the microkernel.
- Threads endeavour to provide time protection and an abstraction of an uninterrupted flow of time on a processing unit.

In OKL4, threads are the API entities responsible for issuing system calls to the microkernel. Threads are also the primary senders and recipients of messages such as synchronous IPC described in Chapter A-6.

OKL4 threads behave in a similar manner to threads in most multi-tasking environments. OKL4 threads do not normally detect the occurrence of preemptions and the fact that other threads may be sharing time on a processing unit. Threads are continued from the next instruction that would have been executed prior to the preemption, with unchanged register state. It should be noted that memory contents may have changed during the execution of other threads while the thread was preempted. Threads may explicitly preempt or *block* themselves only through either: explicit IPC *send* and *receive* operations described in Chapter A-6, page faults and exceptions as described in Sections A-3.2 and A-2.3, or other asynchronous changes to the control flow via the use of the ExchangeRegisters system call described in Section A-12.3.

OKL4 achieves the illusion of an uninterrupted flow of time using an integrated pre-emptive scheduler described in Chapter A-5. This protects threads from time-related misbehaviour within other components in the system. For the benefit of real-time systems that require control over thread scheduling, the microkernel defines a mechanism for delivering pre-emption notification events to threads described in Section A-5.7.

Access to system calls that affect system resources are restricted to threads that have access to the appropriate capability. Capabilities are further described in Chapter A-4. An attempt by any other thread to execute restricted system calls is detected by a microkernel as an error. Other system calls may be executed by an arbitrary thread, although certain parameter combinations may be restricted to threads possessing the requisite capabilities as described in Chapter A-12 of this manual.

A-2.1 Thread States

At a given point in time, the state of a thread may be captured by one of ten distinct *thread states*. The current state of any thread in the system may be obtained using the Schedule system call described in Section A-12.11.

1. A thread is in an *inactive* state when it has been temporarily suspended using the ExchangeRegisters system call described in Section A-12.3, or it encounters an unhandled exception. An inactive thread is unable to be scheduled for execution until it has been resumed using an appropriate ExchangeRegisters system call.
2. A thread is in an *running* state when it has been properly configured using the ThreadControl system call described in Section A-12.14, has not been suspended using the ExchangeRegisters system call described

in Section A-12.3 and is not currently participating in an IPC message exchange described in Chapter A-6. Running threads are free to be scheduled by OKL4. Note that running threads may not be executing on a processor at a given point in time.

3. A thread is in a *polling* state after it has issued a *blocking* IPC *send* operation, but the receiver of the message has yet to issue a corresponding IPC *receive* operation.
4. A thread is in a *sending* state during the transfer of data that occurs after it has issued an IPC *send* operation and the receiver of the message has issued a corresponding IPC *receive* operation.
5. A thread is in a *waiting to receive* state after it has issued a *blocking* IPC *receive* operation to a thread that has not yet issued a corresponding IPC *send* operation, except when it is *waiting for notification* as described below.
6. A thread is in a *waiting for notification* state, when it has issued a *blocking* IPC *receive* operation requesting the reception of an asynchronous notification message as described in Sections A-6.4 and A-12.5.
7. A thread is in a *waiting on mutex* state after it has issue a *blocking* Mutex *lock* operation on a mutex is currently locked.
8. A thread is in a *receiving* state during the transfer of data that occurs after it has issued an IPC *receive* operation and the receiver of the message has issued a corresponding IPC *send* operation.

IPC operations and the definition of the Ipc system call are further discussed in Chapter A-6 and Section A-12.5, respectively.

A-2.2 Thread Identifiers

Each thread in the system, regardless of its state, is associated with a *thread number* which allows each thread in the system to be uniquely identified. As the ability to discern its own thread number is viewed as a capability, the thread number of each thread is a part of its *thread identifier* constructor which belongs to the type, **CAPID**. Thread identifiers and capabilities are further decribed in Section A-9.8.2 and Chapter A-4, respectively.

A-2.3 Exceptions

Most architectures implement *hardware exceptions* which are typically used to detect abnormal programming conditions such as division by zero, or an attempt by user code to execute an undefined machine instruction. The list of conditions that result in an exception is defined by the architecture. Most architectures also use exceptions for delivering of hardware interrupts and page faults. It should be noted that this document does not consider either of these two conditions to be exceptions. They are considered separately in Chapter A-8 and Section A-3.2, respectively.

Typically, the detection of an exceptional condition results in the control flow of the current thread being interrupted by the hardware and transferred to the operating system. Many operating systems including POSIX utilize this to perform automatic error recovery, often by destroying the offending thread. The OKL4 microkernel enables the implementation of such operating system personalities by associating each thread in the system with an *exception handler* entrusted with performing any required error recovery procedure on behalf of the thread. The microkernel converts each hardware exception into an IPC message delivered on behalf of the thread to its exception handler. The exception handler may implement POSIX signal semantics by adjusting the execution of the offending thread using the ExchangeRegisters system call described in Section A-12.3, or simply kill the thread using the ThreadControl system call described in Section A-12.14. The general format of the exception messages is outlined in Section A-13.1, although details may vary between architectures as described in Part C of this manual.

A-2.4 Virtual Registers

Each thread is a finite state machine whose state is contained in its *registers*. In addition to the registers provided by the underlying architecture, OKL4 defines a number of *virtual registers* described in Chapter A-11 that are implemented as thread-local variables stored in the UTCB entry of the thread. Virtual registers come in four variants identified by their *access type*: *none*, *read-only*, *write-only* and *read/write*. Registers with an access type of *none* are said to be *inaccessible*. Each OKL4 implementation provides a documented method of obtaining the value of all virtual registers with an access type of *read-only* or *read/write* and a documented method of modifying the value of all registers with an access type of *write-only* and *read/write*. Details of the access methods are implementation-defined and documented in Part C of this manual. Any attempt by the thread to modify the value of an inaccessible or read-only register or to obtain the content of an inaccessible or write-only register will render the state of the corresponding address space undefined.

The OKL4 API defines three groups of *transient registers*: *parameter*, *result* and *message* registers. For performance reasons, transient registers may be mapped to the general-purpose registers of the underlying architecture during certain system calls. These registers are referred to as transient because their values may be implicitly modified in an unpredictable manner by any machine instruction or system call. Therefore, programs should not assume that the values of these registers are preserved except when used in the manner described in this manual. The remaining virtual registers contain various thread control parameters. Their values are always preserved except when explicitly modified by the user or a system call.

A-3 Address Spaces

OKL4 uses address spaces to manage sharing of memory-mapped resources, such as physical memory, device registers and I/O ports. OKL4 considers system resources that are directly accessible via the memory management unit of a processor to be memory-mapped.

An address space can be viewed as a mathematical function, containing a set of mappings of *virtual addresses* to memory-mapped resources together with a class of permitted memory access operations and a *caching policy* for each mapping. Access permissions and caching policies are further discussed in Sections A-3.2 and A-3.3, respectively. Note that the term "address space" is used to refer to the entire address-mapping function. Each address space covers the entire range of virtually addressable memory supported by the underlying architecture. Similarly, each address space mapping can reference any memory-mapped resource in the system and is limited only by hardware constraints.

The address space mappings are constructed using the MapControl system call described in Section A-12.6. OKL4 divides both the virtual addresses of the system into discrete regions called *flexible pages*. A flexible page has a size of 2^k where $k \geq 10$ and aligned to its own page size, that is they are not restricted to the set of pages supported by the hardware architecture. Flexible pages map on to *pages* provided by the hardware architecture where each page covers a unique region of $2^{MinPageWidth}$ consecutive virtual addresses. Some systems may additionally support larger pages sizes. All pages have a base address which is aligned to the size of the page. Pages of virtual memory map to consecutive regions of physical memory of the same size with a base address that is page aligned. Where hardware cannot support the particular flexible page size, mapping of flexible pages is implemented using multiple pages of a smaller size.

It should be noted that the MapControl system call only allows the mapping of flexible pages greater than or equal to smallest architecture supported page size.

Address spaces are required by OKL4 threads in order to execute code and access memory. Thus, all threads are associated with an address space. Each address space may have zero or more threads associated with it at given point in time. It should be noted that an address space cannot be deleted while it has threads associated with it.

Address spaces serve two main purposes. Firstly, they implement the memory protection facilities that permit operating system personalities to manage secure distribution and sharing of memory-mapped system resources. This is achieved by controlling the visibility of these resources to particular system threads. Secondly, address spaces provide user programs with an *address translation* facility. Parts of the address space may be left unused by the user, allowing for sparse data arrays, demand loading, dynamic memory allocation and virtual memory paging. To facilitate these applications, OKL4 implements *page fault handlers* which provide an efficient mechanism for the incremental construction of an address space function throughout the execution of a program within that address space. Page fault handling is further described in Section A-3.2.

A-3.1 Address Space Identifiers

An OKL4 system consists of a fixed set of *space identifiers*, each of which may be used or available at a given time. A unique space identifier is allocated to each address space and is used to reference a particular address space when using OKL4 system calls. The space identifier of an address space is determined by the user at the time of its creation and cannot be subsequently modified. OKL4 allows the space identifier of an address space to be recycled once the space has been deleted as described in Section A-12.12.

A-3.2 Access Permissions and Page Faults

Each address mapping is associated with zero or more *access permissions* from the set $\{read, write, execute\}$. The individual permissions in the set refer to the use of the architecture-defined *load*, *store* and *instruction fetch* operations, respectively. A mapping with an empty set of access permissions (denoted by $\emptyset$) is effectively excluded from corresponding address space function (deleted). Since no operations are permitted on such a mapping, the corresponding physical address and caching policy is effectively ignored by the microkernel.

The access permissions for a particular virtual address page are specified by the user when establishing the mapping using the MapControl function described in Section A-12.6. Note that access permissions are associated with virtual pages rather than physical pages, so a single physical page may be simultaneously mapped at different locations within an address space using a different set of access permissions.

To facilitate restrictions imposed by individual architectures, the set of permissions specified by the user is adjusted using the PMask parameter described in Section A-10.6. This adjustment does not remove permissions from the set requested by the user, but may add additional permissions to the set as dictated by the requirements of the underlying architecture. In addition, the permission adjustment does not add permissions to a set specified as empty ($\emptyset$) by the user. The adjusted set of access permission is called the set of the *effective access permissions* for the virtual address.

An attempt by a user thread to read, write or execute from memory without the effective *read*, *write* or *execute* permissions is detected by the microkernel as a *page fault*. The microkernel generates a *page fault message* on behalf of the user thread and delivers it to the thread's *page fault handler* as described in Section A-13.2. After performing any recovery actions, the page fault handler replies to the message using the ordinary IPC mechanism described in Chapter A-6, which is also handled by the microkernel on behalf of the faulting thread. A page fault handler thread is associated with an active user thread using the ThreadControl system call described in Section A-12.14.

A page fault event is not necessarily an indication of a programming error. Page fault messages may be used to facilitate dynamic and incremental construction of the address space function by selectively mapping resources to the virtual addresses referenced by the faulting memory access instruction. Page faults may also be used to implement operating system features such as demand-loading, copy-on-write and virtual memory paging.

A-3.3 Caching Policies

The OKL4 microkernel recognizes the importance of caching on memory-mapped resources and annotates each page of virtual memory in the system with a *caching policy*. These annotations dictate the behaviour of memory system for all memory accesses referencing the corresponding virtual address.

The choice of caching policy may affect the semantics of access to physical memory resources. In particular, accesses to memory mapped devices often require uncached mappings. This is to ensure that writes propagate to the device immediately and reads always get the current state of the device. Also of importance is preserving the strict ordering of operations on the device.

In addition, on systems with multiple processing units that do not share a portion of the cache hierarchy, the hardware must ensure that results of a memory operation performed on one processing unit are visible on the remaining units in the system. This is known as *cache coherency*. A particular architecture may choose to provide caching policies without cache coherency in order to improve system performance. On such systems, the software must insert explicit *memory barriers* to ensure that modifications to memory performed on one processing units are visible to the rest of the system. The specific mechanism by which memory barriers are inserted on a particular architecture is defined by the corresponding instruction set. However, in all cases OKL4 provides the CacheControl system call that may be used to maintain cache coherency without use of architecture-specific memory barrier instructions. The CacheControl system call is further described in Section A-12.1. In summary, the design space for caching policies can be described by considering caching of memory *read* and *write* operations separately as described in Section A-3.3.1.

Irrespective of the capabilities of a particular architecture, the OKL4 microkernel recognizes a set of caching policies represented by the **CachingPolicy** type defined in Section A-9.6. The policy desired for a particular region of the address space is configured using the MapControl system call defined in Section A-12.6. Few architectures support the full design space of caching policies described in this chapter. Therefore, OKL4 implementations adjust the caching policy requested by the user to any policy that provides equal or higher predictability. Any request for a particular caching policy may be "satisfied" by an OKL4 implementation executing on an architecture without a user-controlled caching hardware by disabling caching of the corresponding memory resources, although it is expected that such drastic means of implementation will not be necessary in practice. Caching policies form a weak total ordering defined by the list in Section A-3.3.1 and any requests for a particular policy are interpreted by the microkernel as a "ceiling" of the actual policy to be employed for the corresponding virtual memory page. Caching policy adjustments and additional architecture defined caching policies are described in Part C of this manual.

A-3.3.1 Caching Operations

An explanation of all possible caching operations is beyond the scope of this document. As caching of operations significantly affect system performance, choosing the correct policy for your system may require analysis and testing. There are a large number of caching policies implemented by a variety of different hardware architectures. A few common points and issues are discussed below.

- *cacheable* memory generally refers to memory for which the processor may consult the caches first to find a cached copy, before accessing physical memory. In addition, the processor may update the cached copy when performing a write operation.

- *line-allocation* refers to the policy the processor uses to insert entries into the cache when a cache-miss occurs. Typically, either read operations or write operations are chosen, sometimes both. In caches where a cache-miss occurs and the line-allocation policy is not satisfied, the operation is performed on physical memory without accessing the cache.

- *write-back* refers to a policy where updates to the cache mark an entry as "dirty" for later write-back to physical memory (typically an explicit flush or cache-line eviction). If write-back is disabled, the processor may update the cache as well as physical memory.

- *write-buffers* are small caches which are used to collect data from multiple consecutive write operations to adjacent memory locations into a single write operation which may subsequently be presented to the caches or physical memory.

- *cache-coherency* refers to hardware-supported mechanisms for keeping two or more separate caches coherent. Typically this allows other processing units to see changes to memory without software support.

A particular type of cache-coherency issue faced by OKL4 implementations is the consistency between instruction and data caches. Unless stated otherwise in the corresponding chapter in Part C, programs should always assume a lack of hardware-enforced coherency between instruction and data caches. Programs should therefore flush all instruction caches after each modification to an executable region in memory. Note that this is rarely a cause of portability problems, as manipulation of executable memory is architecture-specific by nature.

A-3.4 Address Space Regions

The address space function is partitioned into *address space regions*. The OKL4 API defines the following region types:

- *User regions* are accessible to all threads executing in the corresponding address space. This region may be configured using the MapControl system call described in Section A-12.6.

- *System regions* of an address space are reserved for internal use by the OKL4 microkernel. An attempt by a user thread to access an memory within such a region is detected by the microkernel as a page fault.

- *Unmapped regions* of an address space are reserved by the hardware architecture. An attempt by a user thread to access an object within such a region is detected by the microkernel as an address error exception.

- The *UTCB region* described in Section A-3.5 is used by OKL4 to store the *thread control blocks* of the threads associated with the corresponding address space.

- *Global regions* are automatically mapped into each address space and shared between all address spaces. The generic OKL4 API does not require any such regions to be provided by an implementation. However, these regions may be present on some OKL4 implementations as described in Part C of this manual.
- Lastly, some architectures define one or more *special regions*. The semantics of such regions are defined for each architecture in the corresponding chapter in Part C. Depending on the architecture, some of these regions may be available for mapping using MapControl which assigns special semantics to memory accessed through virtual addresses within those regions.

The UTCB and system regions are known as *reserved regions*. An OKL4 application should not attempt to access these regions except by using the facilities provided by the OKL4 API. However, such an access does not violate the integrity of other address spaces in the system. Page fault handlers should always detect attempts to access reserved regions and treat them as unrecoverable programming errors.

A-3.5 The UTCB Region

Each address space contains a *User Thread Control Block (UTCB) region*, used by OKL4 to store some of the virtual registers for threads executing in the address space.

The current OKL4 API does not provide a facility for a user thread to obtain the location of the UTCB region within its address space. The operating system personality must establish its own protocol for user threads to determine the location of the UTCB region.

The size and location of the UTCB region is fixed in some OKL4 implementations. This allows the operating system personality to treat it as another system region. Such implementations are distinguished by the UtcbRegionWidth parameter which is set to a zero value, and place no restrictions on the number of threads associated with an address space at any given time. The UtcbRegionWidth parameter is further described in Section A-10.8.

In OKL4 implementations with a UtcbRegionWidth parameter set to a non-zero value, the size and location of the UTCB region must be configured by the user, using the SpaceControl system call described in Section A-12.12. Note that the address spaces must be in the new state, the requested size of the UTCB region must be greater than $2^{\mathsf{UtcbRegionWidth}}$, and the selected location must reside within what would otherwise constitute a part of a user region of the address space. Under such implementations the number of threads *n* associated with an address space at any given time is restricted by the following equation:

$$0 \leq n \leq \left\lfloor \frac{2^{RegionSize}}{UtcbSize} \right\rfloor$$

where $UtcbSize$ is defined as $2^{\mathsf{UtcbAlignment}} \times \mathsf{UtcbMultiplier}$ and $RegionSize$ is the size of the particular UTCB region configured using the SpaceControl system call described in Section A-12.12.

The effective access permissions for the UTCB region are implementation-defined and cannot be changed by the user. The content of this region is also implementation-defined and described in Part C of this manual. User threads should use the facilities provided by the OKL4 API to access the content of the UTCB region and not attempt to access it directly. On implementations with a user-configurable UTCB region location, OKL4 relies on the operating system personality to manage allocation of *user thread control block entries* (UTCB entries) within the UTCB region of an address space. On such implementations, the UTCB region is considered to be an array of UTCB entries, each of which occupies $UtcbSize$ bytes of the UTCB region. The content of individual UTCB entries is implementation-defined. The operating system personality is responsible for ensuring that no two threads within an address space share the same UTCB entry. The location of the UTCB entry of a new thread is configured using the ThreadControl system call described in Section A-12.14.

A-4 Capabilities

Cpability systems are used to control access to objects, by restricting access to objects to users who have the requisite tokens or *capabilities*. Each *token* or *capability* also specifies the rights the user has with regards to the object, such as reading, modifying, or copying the capability.

A *capability space* is a collection of tokens that the user has access to. The capability space may be shared with other users, that is all sharing parties have the same access rights to the objects represented by the capabilities.

OKL4 uses capabilities to manage access control to system resources. In the current version of the API, only thread objects are protected using capabilities. A *thread identifier* as described in Chapter A-2 is a capability to a thread.

Capabilities can be used to construct secure systems in which communication between threads can be completely controlled using capabilities and capability spaces.

A-4.1 Capability Lists

In OKL4, each address space has a single associated capability space. All threads within the address space can access the capabilities within the capability space. A capability space may be shared by zero, one or many address spaces. A capability space in the current version of OKL4 is implemented as a *capability list* or *clist* object. A clist is a first class kernel object and can be operated on using the CapControl system call described in Section A-12.2.

A clist is a container for capabilities and has a number of slots numbered from 0 to $n-1$ where n is the total number of slots in the clist. Each clist may have a different number of slots and can be created and deleted using the CapControl system call. In each slot of the clist, one capability can be created. Empty slots contain invalid capabilities on which no operations can be performed.

A-4.2 Capability Types

OKL4 provides several types of capabilities, each of which is decribed below.

- ***Access Capabilities***

  Access capabilities are general capabilities which index into either the callers or specified capability list.

- ***Physical Memory Segment Resource Capabilities***

  Physical Memory Segment Resource Capabilities are used to provide access to physical memory segment resources owned by the address space of the caller.

- ***Reply Capabilities***

  Reply Capabilities are used to allow a thread to reply to a particular IPC message.

- ***Special Identifiers***

  Special Identifiers are used for IPC and other system calls to indicate *nilthread*, *anythread*, *myself-id* and *waitnotify*.

A-4.3 Thread identifier mapping

A thread identifier as specified in Chapter A-2 is a capability to a thread and may be a thread cap or ipc cap. A thread identifier is used by the kernel to lookup the thread object from the clist associated with the thread's address space. The *thread number* part of the thread identifier is used by OKL4 as the index into the clist. A thread number that falls out side the valid slot range of the clist will reference an invalid capability.

A-4.4 Reply Capability

A special type of transient capability also exists. This capability is the *reply cap* and is part of the capability space but not within a clist. This capability is generated by the kernel during IPC operations and can only be used to reply to threads using IPC. See Chapter A-6 and Section A-12.5.

A-5 Scheduling Threads

The OKL4 microkernel provides a pre-emptive scheduler responsible for allocating processing time to user threads. The scheduler implements a pre-emptive rigid-priority round-robin scheduling algorithm. It is used by the OKL4 microkernel to enforce its fault isolation policy with regards to time-related misbehaviour. Pre-emptive scheduling allows the microkernel to forcibly revoke the processing time of a thread. This ensures that misbehaving threads do not affect the integrity of the remainder of the system. Priorities can be used by the operating system to implement an array of scheduling policies including policies suitable for real-time applications.

OKL4 allows the use of schedule inheritance to avoid unbounded priority inversion which occurs when a high-priority thread cannot run for an indefinite period of time because it is waiting on an object held by a low-priority thread prevented from executing by a medium priority thread. OKL4 allows the user to enable or disable scheduling inheritance at compile time as required. Unless schedule inheritance is enabled, the rigid nature of OKL4 priorities means that the microkernel never implicitly adjusts the priorities of individual threads, giving the operating system personality complete control over management of thread priority levels. Note that a rigid priority system allows for the possibility of thread *starvation*. If this is unacceptable, the operating system personality should dynamically adjust thread priorities to prevent the starvation of lower-priority threads. Schedule inheritance is further described in Section A-5.8, below.

Each user thread in the system is associated with the following *scheduling parameters: priority level*, *allowed execution units*, *time slice* and *remaining time slice*. The first three parameters are configured using the Schedule system call described in Section A-12.11, and the remaining time slice parameter is maintained internally by the microkernel. Each parameter is described in the following sections.

A-5.1 Priority Levels

Each thread is associated with a *priority level* (or *priority*) which is used by the microkernel during the allocation of processing time to user threads. The priority level is a positive integer in the range 0–255. A thread is said to have a *higher priority* than another thread if its priority level is numerically greater than that of the other thread. Upon initialization, the priority of the newly created thread is set to 0. It may be subsequently modified using the Schedule system call described in Section A-12.11.

A-5.2 Allowed Execution Units

The set of execution units on which a thread may be allocated time is specified by its *allowed execution units* as specified by its *hardware thread mask*. The hardware thread is defined as an execution unit and is associated with a single domain which is a collection of one or more hardware threads.

A-5.3 Time Slices

The time slice of a thread specifies the maximum period of time a thread may execute on a processor before being pre-empted. Specifically, it defines the initial value of the *remaining time slice*. Upon initialization of a new thread, its time slice is set to a default value. This value may be subsequently modified using the Schedule system call as described in Section A-12.11. The *time slice* parameter can be set to an infinite value, effectively disabling pre-emption of the corresponding user thread.

A-5.4 The Remaining Time Slice

Each user thread with a finite time slice parameter is associated with a *remaining time slice* parameter that is maintained internally by the microkernel. Upon initialization, the remaining time slice of a thread is set to a default value. The remaining time slice is reset to the time slice of the thread when the time slice parameter of the thread is modified.

The microkernel decrements the remaining time slice parameter one to one for every microsecond of processing time consumed by the thread unless the parameter has been initialized to an infinite value. When the remaining time slice reaches 0, the thread is pre-empted as described in Section A-5.7. The remaining time slice is reset back to its initial value as configured using the Schedule system call. The current value of a thread's remaining time slice parameter may be obtained using the Schedule system call as described in Section A-12.11.

A-5.5 Scheduling Queues

The microkernel maintains one or more *system scheduling queues* which collectively contain all active threads in the system. Each queue maintains threads running on one or more processing units. Each processing unit can only be associated with one system scheduling queue. These queues are operated on by the thread selection algorithm described in Section A-5.6. In addition, it maintains an additional *IPC queue* for each initialized OKL4 thread in the system. These queues contain threads currently performing IPC *send* operations on the corresponding thread as described in Section A-6.3.

Each queue is represented by a first in first out (FIFO) list of thread identifiers, with the first inserted thread located at the start of the queue. Threads are removed from a queue (dequeued) using the thread selection algorithm described in Section A-5.6. When a new thread is added to a queue (enqueued), it is appended to the end of the queue. IPC queues are not priority sorted.

A-5.6 Thread Execution and Selection Algorithm

Each processing unit in the system is associated with a *current thread*, which is the thread that is allocated processing time on that unit at the given point in time. The current thread can obtain the number of the processing unit from its ProcessingUnit register described in Section A-11.17. The current thread releases its processing unit for use by another thread in the following situations:

- When its remaining time slice parameter reaches 0 as described in Section A-5.4, the thread is *pre-empted*. Thread pre-emption is described in Section A-5.7.
- When a thread invokes the ThreadSwitch system call, it forfeits any remaining processing time allocation specified by the remaining time slice parameter as described in Section A-12.15 and is immediately enqueued in the system scheduling queue.
- When a thread performs an IPC *send* operation described in Chapter A-6, or when the microkernel performs an IPC *send* operation on behalf of the thread as described in Chapter A-13, the thread immediately releases its remaining allocation of processing time and is added to the *sender queue* of the target thread.
- When a current thread is suspended using the ExchangeRegisters system call described in Section A-12.3, its allocation of processing time is released until it is subsequently resumed using the ExchangeRegisters system call.
- Lastly, when a current thread is deleted using the ThreadControl system call described in Section A-12.14, its remaining allocation of processing time is released before entering the *new* state as described in Section A-2.1.

In all situations, the microkernel selects the new current thread by searching the system scheduling queue for the first thread with the correct domain and corresponding hardware thread included in its allowed processing units. The selected thread is then removed from the system scheduling queue and allocated the amount of processing time specified by its time slice parameter. The same algorithm is used to select a thread from the thread's receiver and sender queues during the IPC operations described in Chapter A-6.

A-5.7 Thread Pre-emption

When a current thread is pre-empted by the microkernel, it immediately releases the corresponding processing unit for use by other threads. If the *Signalled* flag in the thread's PreemptionFlags register is set, the thread's PreemptedIp register is set to the current value of its InstructionPointer register, and the InstructionPointer is set to the value of the thread's PreemptCallbackIp register. Irrespective of the value of the *Signalled* flag, the thread is then added to the system scheduling queue as described in Section A-5.5. This permits real-time threads to detect pre-emption events in order to maintain integrity of its data structure, timing invariants or real-time deadlines.

A-5.8 Schedule Inheritance

OKL4 allows the user to enable schedule inheritance at compile time to prevent *priority inversion*. Priority inversion occurs when a high-priority thread is forced to wait for a low-priority thread to finish an operation. In certain cases, a high-priority thread may be prevented from running for an indefinite period of time because a medium-priority thread prevents a low-priority thread from executing. This is known as an *unbounded priority inversion*.

A priority inheritance protocol, where the scheduler automatically donates the priority of a thread blocked on a resource to the thread holding that resource, may be used to avoid unbounded priority inversion.OKL4 uses *schedule inheritance*, an implementation of priority inheritance to avoid unbounded priority inversion.

OKL4 implements schedule inheritance in the following manner. Ordinarily, when a thread becomes dependent on another thread, it is removed from the scheduling queue. However, where a thread *P*, becomes dependent on another identifiable thread *Q*, the kernel notes the object that the thread is waiting on, but leaves thread *P* in the scheduling queue. If thread *P* is selected by the scheduler, this information is then used to determine the thread to which thread *P*'s schedule is donated to.

For example, if thread *P* that is currently dependent on thread *Q* is selected, the scheduler will donate thread *P*'s schedule to thread *Q*. Thread *Q* will then run until either thread *P*'s schedule is pre-empted or until thread *Q* changes state in such a way that thread *P* no longer depends on it.

If thread *Q* is in turn dependent on another identifiable thread, the scheduler will continue the process of following this dependency chain until either it finds a thread to which it can donate thread *P*'s schedule, or until it detects a deadlock. Deadlock detection is further described in Section A-5.9, below.

A-5.9 Deadlock Detection

Where schedule inheritance is enabled, it is possible for a chain of dependencies to form a loop. As a result, deadlock detection is required by the scheduler to ensure that it is able to determine when it should stop following a chain of dependencies.

The OKL4 kernel performs deadlock detection to ensure that the kernel does not deadlock during scheduling operations. Once detected, deadlocks are removed by halting all threads in the dependency chain.

A-6 Inter-Process Communication

Inter-process communication (IPC) is the central component of the OKL4 system design. OKL4 facilitates both communication between threads residing in different address spaces and threads within the same address space. In OKL4, all communication is synchronous and unbuffered with the exception of the restricted form of asynchronous communication described in Section A-6.4.

IPC consists of the exchange of short messages between two threads in the system. Each message consists of a *message tag* and an optional list of *message data* described in Sections A-6.1 and A-6.2. The messages are exchanged directly through the MessageData registers described in Section A-11.7. This exchange is *unbuffered*, the microkernel copies the content of the MessageData registers of the sender directly to the corresponding registers of the receiver. This is the key to the high performance of the OKL4 microkernel. To ensure that the integrity of the receiving thread is not compromised, messages are exchanged *synchronously*. That is, the microkernel does not deliver the message until the recipient is ready to receive the supplied data, often suspending execution of the sender until the message is transmitted.

OKL4 defines two fundamental IPC operations.

- A *send* operation delivers messages from the calling thread to a destination thread. This operation is further described in Section A-6.3.
- A *receive* operation requests a message from another thread. This operation is further described in Section A-6.5.

These two operations are associated with user requested policy or whether to perform the operation *blocking* or *non-blocking*. If an IPC operation is blocking, it will cause the IPC, and thus the thread, to block until a corresponding IPC operation has been performed by the target thread. A non-blocking IPC operation fails immediately if the target thread is not ready to participate in the message exchange at the time of request. Additionally, a special encoding of the send operation known as a *notify* operation may be used to send asynchronous notification events.

OKL4 supports the asynchronous delivery of a restricted form of messages of a set of **FLAG** objects described in Section A-6.4.

Additionally, a single invocation of the Ipc system call may be used to perform a combination of IPC operations as described in Section A-12.5. Such composite operations are executed atomically by the microkernel. The invoking thread remains blocked until all operations requested by the system call are completed or aborted. Specifically, a single invocation of the Ipc system call may be used to perform either a single IPC operation or a pair of IPC operations. In the later case, the first operation must be a *send*, and the second must a *receive* operation.

The recipient of an IPC message is supplied with a reply cap to the thread that has issued the *send* or *reply* operation. This capability allows the recipient to reply to the message as long as the sender has also specified a *receive* operation to the same recipient thread in the IPC operation.

A-6.1 Message Tags

Each IPC message begins with a *message tag* stored in the $\mathsf{MessageData}_0$ register. Message tags are represented by the **MESSAGETAG** type described in Section A-9.26. Each tag consists of a *message label*, a *Untyped* field specifying the number of data words in the corresponding message, and a set of four *message flags* *S*, *R*, *N* and *M*. These flags are used to identify the requested IPC operations as described in Section A-12.5 and are not delivered to the target thread. Upon the delivery of the message, the microkernel replaces the these flags of the message tag with an error indicator and a performance-related *remote IPC* flag described in Section A-12.5. All other fields of the message tag are always delivered unchanged to the recipient of the message.

A-6.2 Message Data

Each message contains at least 16 bits (or 48 bits on 64-bit architectures) of user-interpreted data represented by the message label stored in the *Label* field of its message tag. The message label is not interpreted or modified by the microkernel and is delivered directly to the recipient. The user is free is use this field for any purpose, however it is commonly used for specifying the message type.

The size of the message label means it is not generally useful for sending information. Therefore, OKL4 implementations allow the user to attach up to 63 user-interpreted words of information to a single IPC message. Note that individual OKL4 implementations may reduce the maximum message size by setting the MaxMessageData system parameter to a value less than 63 as specified in Section A-10.2. The number of data words attached to a particular message is specified in the *Untyped* field of the message tag. As with the message label, the microkernel does not interpret or modify the data words. The sender of a message supplies the values of any required data words in its $\mathsf{MessageData}_1$ to $\mathsf{MessageData}_k$ registers, where k is the value of the *Untyped* field in the corresponding message tag. Upon successful delivery of a message, these registers are copied to the corresponding registers of the target thread.

A-6.3 Sending IPC Messages

When a thread requests an IPC *send* operation, the microkernel looks up the destination thread, by checking the corresponding capability in the sender's capability space. If the capability is not found, the operation is immediately aborted with an appropriate error message without performing any *receive* operations requested. Otherwise, the behaviour depends on the state of the target thread:

- If the target thread is waiting to receive a message from the sender, the message is delivered immediately to the recipient and the microkernel proceeds with any IPC *receive* operations requested by the sender.
- Alternatively, the microkernel checks whether the requested send operation was requested as blocking or non-blocking.
  - If a blocking send operation is requested, the microkernel marks the sender as *polling* and enqueues it in the IPC queue of the recipient as described in Section A-5.5. The sender will remain blocked until the recipient issues an appropriate *receive* operation, or until the IPC operation is cancelled using the ExchangeRegisters system call as described in Section A-12.3.
  - If a non-blocking send operation is requested, the operation is immediately aborted with an appropriate error message without performing any IPC *receive* operations requested by the sender.

A common term for a non-blocking *send* is an IPC *reply* operation.

Thread states and thread scheduling, are further discussed in Section A-2.1 and Chapter A-5 of this manual.

A-6.4 Sending Asynchronous Notification Messages

The IPC *notify* operation provides a restricted form of asynchronous communication between threads and consists of the delivery of a set of *notify flags* to the recipient without the need for the recipient to invoke an explicit IPC *receive* The *notify* operation is always non-blocking and delivers only a single word of data representing the desired notify flags to the recipient irrespective of the number of data words specified by the sender in the corresponding message tag.

When a thread requests an IPC *notify* operation, the microkernel looks up the destination thread, by checking the corresponding capability in the sender's capability space. If the capability is not found, the operation is immediately aborted with an appropriate error message. The operation is also aborted if the *Notify* flag is cleared in the Acceptor register of the recipient. Otherwise, the notify flags are immediately delivered to the recipient by updating its NotifyFlags. Notify flags can only set flags in the destination, a cleared flag in the message has no effect on the current value of the corresponding flag in the NotifyFlags register of the recipient.

The microkernel then checks whether the recipient is waiting for an IPC from the sender. If so, it compares the *updated* value of the recipient's NotifyFlags with the value of its NotifyMask register. If the intersection of these two registers is non-empty, the microkernel delivers an asynchronous notification message to the recipient.

When an asynchronous notification message is delivered, the microkernel delivers the set of flags from the intersection of the recipient's NotifyFlags ands NotifyMask registers in the message. It subsequently clears these bits from the recipient's NotifyFlags register. These two operations are done atomically. Furthermore, the microkernel clears the *Untyped*, *Label* and *P* fields in the message tag as described in Section A-12.5.2 and delivers the supplied message to the recipient as if it was performing a normal *send* operation. The microkernel indicates the origin of message as coming from NILTHREAD. The microkernel then proceeds with any IPC *receive* operations requested by the sender.

A-6.5 Receiving IPC Messages

The IPC *receive* operation is used by a thread to request a message from another thread in the system. Besides blocking and non-blocking variants, the OKL4 microkernel defines two variants of the *receive* operation, *closed* and *open*. In a closed receive operation, a thread (*recipient*) requests a message from a specific OKL4 thread. The caller must have a capability to the recipient in its capability space.

In an open receive operation, the recipient requests a message from any member of a particular class of threads. The list of recognized classes of threads is specified in the description of the Ipc system call in Section A-12.5. The behaviour of the microkernel upon reception of the IPC *receive* operation depends on the content of its IPC queue described in Section A-5.5:

- If the IPC queue contains the specific thread, or a thread from the class of threads specified by the recipient, the first such thread is removed from the queue and the message is delivered immediately to the recipient. The sender is added back to the system scheduling queue. The microkernel will complete any pending *receive* operations requested by the sender once it is allocated processing time by the thread selection algorithm described in Section A-5.6.
- Alternatively, the microkernel checks whether the requested receive operation was requested as blocking or non-blocking.
  - If the recipient requests a blocking receive operation, the microkernel marks it as *waiting to receive* and will remain blocked until the sender issues an appropriate IPC *send* operation, or until the IPC operation is cancelled using the ExchangeRegisters system call described in Section A-12.3.
  - If the recipient requests a non-blocking receive operation, the *receive* operation is aborted immediately with an appropriate error message.

On successful reception of a message the microkernel provides the recipient with a reply cap to the sender. If the sender specified a *receive* operation specifying the recipient, the recipient may use this capability to reply to the thread.

A common term for a blocking *receive* operation is an IPC *wait* operation which may also be used with the *open* or *closed* prefix.

Thread states and thread scheduling are further described in Section A-2.1 and Chapter A-5 of this manual.

A-7 Mutex

OKL4 uses mutexes to manage access to shared resources which should only be accessed by a single thread at a time. A mutex is a synchronization primitive used to ensure that only a single thread has access to a shared resource at a given point in time. A thread acquires a mutex by performing a *lock* call on the mutex and once it has finished with the resource, the thread releases the mutex by performing an *unlock* call. As only a single thread can hold a particular mutex at any one time, if a thread tries to acquire a mutex that is already held by another thread, it will block until the holder releases the mutex.

OKL4 provides two types of mutexs, kernel mutexes and hybrid mutexes, each of which is described in Section A-7.1. Mutex creation and deletion is described in Sections A-7.2 and A-7.3. Mutex usage is described in Section A-7.4.

A-7.1 Mutex Types

OKL4 provides two types of mutexes, kernel mutexes and hybrid mutexes.

A-7.2 Mutex Creation

A mutex may be created by calling the MutexControl system call. The caller must specify the appropriate capability to create a mutex.

The MutexControl system call is further described in Section A-12.9.

A-7.3 Mutex Deletion

An existing mutex may be deleted by calling the MutexControl system call with the appropriate values.

The MutexControl system call is further described in Section A-12.9.

A-7.4 Using a Mutex

Prior to its use a mutex must be created as described in Section A-7.2. A thread possessing the appropriate ¡¿ capability for an existing mutex may attempt to obtain that mutex using the ¡¿ system call.

The ¡¿ system call is further described in Chapter A-12.

- mutex must exist - cannot be held by another thread - need to hold the appropriate capability.

A-8 Interrupts

OKL4 supports the management of hardware interrupts in two aspects by providing an abstraction of hardware operations and providing access control.

OKL4 defines a common interface to handle common interrupt-related platform-specific operations that must be performed on hardware. The interface includes operations such as enabling and disabling interrupts, the clearing and acknowledgement of interrupts, as well as defining the interrupt vector entry point. This interface is a part of the OKL4 Board Support Package (BSP).

OKL4 uses the asynchronous inter-process communication (IPC) mechanism to deliver hardware interrupts to threads.

Each handler thread can register with the kernel for one or more asynchronous notification bits to be used to indicate pending interrupts. The OKL4 Board Support Package (BSP) defines an *interrupt descriptor* data-structure. One or more interrupt descriptors reside in the UTCB of a handler thread. This descriptor is used by the BSP to indicate to the handler which interrupt(s) are currently pending.

In this design, the BSP is responsible for maintaining the mapping of interrupt sources to interrupt descriptors, and informing the kernel that a notification needs to be delivered to the handler. The BSP is required to determine the handler thread and which notify bits need to be updated when delivering an interrupt. The platform code updates the handlers interrupt descriptors and calls the generic kernel code to deliver the notify bits to the thread. The BSP may call this interface multiple times to deliver interrupts to multiple handlers. When all pending interrupts have been handled, the BSP calls the kernel scheduler to determine the next thread to run.

Delivering interrupts using asynchronous notification provides the handler the option of polling pending notify bits and interrupt descriptor(s). Note that this is supported at the discretion of the BSP. The handler thread may still be required to acknowledge these interrupts with the kernel to guarantee future delivery of these interrupts.

A-8.1 Interrupt Acknowledgement

A new dedicated system call, InterruptControl, will be added to perform acknowledgement of interrupts as well as registration of interrupts as described below. Acknowledgement of interrupts is necessary in order to signal the hardware that the interrupt has been handled and can be unmasked.

Acknowledgement of interrupts optionally allows a thread to wait for subsequent interrupts (equivalent to WaitNotify) within the same system call, allowing efficient interrupt handling.

Waiting for interrupts may also be performed using a `L4_WaitNotify` operation, though this does not acknowledge the hardware. This is only used in order to wait for an initial interrupt, or if interrupts do not require explicit acknowledgement. On some platforms, acknowledgement of interrupts may not be required (e.g.. edge triggered). The BSP has the discretion of determining whether acknowledgement of interrupts is required. Note that if pending interrupts are maintained by the BSP, acknowledgement of interrupts is likely to be required.

When performing an acknowledge operation using the InterruptControl system call, the caller is required to specify the interrupts to acknowledge. These are encoded in a set of one or more *reply interrupt descriptors* provided in the message registers. The kernel then calls the BSP code and provides these arguments for decoding. The BSP will acknowledge these interrupts and may inspect the hardware (or saved pending interrupts) for further pending interrupts at this time and deliver them to the handler. If an optional wait for pending interrupts is specified, the kernel will check if any of the calling thread's registered notify are set and returns immediately if any are set. Otherwise, the calling thread blocks until it receives new notify bits.

Security checks must efficiently determine whether the calling thread may acknowledge the reply interrupt descriptor(s). These checks must ensure that the caller's address space has access to the specified interrupts to acknowledge. Access control to interrupts is described below.

Note that care must be taken when acknowledging interrupts from a thread other than the registered interrupt handler. If the optional wait for pending interrupts is requested, the calling thread will not be delivered pending interrupts for the handler thread.

A-8.2 Interrupt Access Control

All threads must be granted permission prior to using the InterruptControl system call to wait and acknowledge interrupts. The *Elfweaver* tool is used to grant address spaces access to interrupts at build time. Threads within the address space can then register and unregister as handlers to any interrupt for which the address space has access to.

It is up to the BSP to encode and store the interrupt to address space mappings. It also enforces the address space access rights. The BSP is required to perform these checks when registering and unregistering interrupts to handlers. The BSP is also required to perform checks during acknowledgement of interrupts to ensure that only authorized threads can acknowledge interrupts.

A-8.3 Interrupt Registration

Threads may register and unregister as interrupt handlers using the InterruptControl system call. Only one thread can be the handler for a particular interrupt at any one time. Note that the handler thread for a particular interrupt must be unregistered prior to registering a new interrupt handler thread for the same interrupt. Registration of a new handler will return an error if a handler already registered for that interrupt.

A-9 Data Types

In this manual the term *type* is used to characterise the objects stored in memory and/or virtual registers. Types are used to define the encoding of objects in terms of a sequence of bits and occasionally to impose additional restrictions such as a range of numeric values allowed in a particular field. In the generic version of the OKL4 API, any specified restrictions should be interpreted as *conservative*. Specific OKL4 versions are permitted to relax these restrictions provided that they are documented. As the binary encoding of data types is covered in Parts B and C of this manual, it is not covered in this chapter.

An OKL4 type represents one or more alternative pieces of information called *data constructors* (or *constructors*). At any time, an object of a given type may *contain* precisely one of the constructors allowed for that type. Each constructor consists of a globally-unique name and a list of zero or more *data fields* (or *fields*). Each field contains a single object of a predetermined type. Note that, in many situations the values of a given type may be restricted beyond the requirements imposed by the types of their fields. For example, values of an individual component of the **DATE** type are restricted to refer to valid calendar dates. Such restrictions are introduced as they become relevant. C programmers may find it convenient to think of OKL4 data types as `union` types containing one member for each constructor, and individual constructors as `struct` types with one member for each data field.

In this manual, the values of constructors with no data fields will be represented by the name of the constructor (for example, NILPAGE), and the values of constructors with one or more fields by their name followed by a comma-separated list of field values in parentheses, listed in the order of their specification in this chapter. Any exceptions to this notation is addressed in the description of the corresponding type. The mechanisms used to identify the constructor contained in an object are defined in Parts B, C and D of this manual.

Some types serve a specialised purpose and are only used by a single virtual register or system call parameter. The description of these types has been replaced with a reference to the section of the manual which describes the relevant feature.

A-9.1 Acceptor

Constructor:	Description:
Acceptor	The Acceptor Constructor

Data Field:	Type:	Description:
Notify	**Flag**	Accept asynchronous notifications

The Acceptor type is used to specify the type of messages accepted by a thread. Specifically, the *Notify* flag in the Acceptor virtual register defined in Section A-11.1 indicates whether the thread is willing to accept asynchronous notification messages described in Section A-6.4.

A-9.2 Address

The **Address** type is a refinement of the **Word** type used to represent absolute virtual memory addresses. It is restricted to values which are valid within an address space. Since **Address** refines the **Word** type, it is subject to the notational conventions and restrictions of the **Word** type described in Section A-9.45.

A-9.3 CacheControl

Constructor:	Description:
CacheControl	The CacheControl Constructor

Data Field:	Type:	Description:
Count	**Word**$_6$	Number of cache regions
Op	**CacheOp**	The cache operation
L1	**Flag**	Select L1 cache
L2	**Flag**	Select L2 cache
L3	**Flag**	Select L3 cache
L4	**Flag**	Select L4 cache
L5	**Flag**	Select L5 cache
L6	**Flag**	Select L6 cache

The **CacheControl** type is used to define the content of the *Control*$_{\text{IN}}$ input parameter to the CacheControl system call. CacheControl is further described in Section A-12.1.

A-9.4 CacheOp

Constructor:	Description:
FlushCache	Flush specific cache regions
LockCache	Lock a region in the cache
UnlockCache	Unlock a region in the cache
FlushICache	Flush the entire instruction cache
FlushDCache	Flush the entire data cache
FlushAllCaches	Flush all instruction and data caches

The **CacheOp** type specifies the cache operation to be performed by the CacheControl system call. The CacheControl system call is further described in Section A-12.1.

The encodings for the **CacheOp** constructors are defined in B-2.2.1.

A-9.5 CACHEREGIONOP

Constructor:	Description:	
CACHEREGIONOP	The CACHEREGIONOP Constructor	
Data Field:	**Type:**	**Description:**
Count	**WORD**$_{28}$	Number of consecutive 16-byte sized lines
IC	**FLAG**	Include instruction cache
DC	**FLAG**	Include data cache
C	**FLAG**	Clean cache lines
I	**FLAG**	Invalidate cache lines

The **CACHEREGIONOP** type is used to selectively control cache lines corresponding to a specific range of virtual addresses using the CacheControl system call described in Section A-12.1. Note that the **FPAGE** type cannot be used for this purpose as caches are typically controlled with a finer granularity. The **CACHEREGIONOP** type specifies the size of the region (encoded as $16 * \mathit{Count}$), two flags indicating which of the two cache types (instruction and/or data) should be modified, and two flags indicating whether any cache lines referring to the specified region should be invalidated and/or cleaned. The location of the region is supplied to CacheControl in a separate parameter. **CACHEREGIONOP** fields are further described in Section A-12.1.

A-9.6 CACHINGPOLICY

Constructor:	Description:
DEFAULTPOLICY	Architecture-specific default policy
CACHEDPOLICY	Architecture-specific
UNCACHEDPOLICY	Caching disabled
WRITEBACKPOLICY	Write-back caching policy
WRITETHROUGHPOLICY	Write-through caching policy
COHERENTPOLICY	A coherent caching policy
DEVICEPOLICY	Architecture-specific policy for device memory
WRITECOMBININGPOLICY	Uncached write-combining policy

The **CACHINGPOLICY** type defines the caching policy desired for a particular memory-mapped resource. The caching policy for a resource may be configured using the MapControl system call described in Section A-12.6. Individual caching policies are further described in Section A-3.3.

The encodings for the **CACHINGPOLICY** constructors are defined in B-2.2.2.

A-9.7 CAPCONTROL

Constructor:	Description:	
CAPCONTROL	The CAPCONTROL Constructor	
Data Field:	**Type:**	**Description:**
Op	**CAPOP**	The capability operation

The **CAPCONTROL** type is used to define the content of the $\mathit{Control}_{\mathrm{IN}}$ input parameter to the CapControl system call. The CapControl system call is further described in Section A-12.2.

A-9.8 CapId

Constructor:	Description:
CapId	The CapId Constructor (see: Section A-9.8.1)
ThreadId	The ThreadId Constructor (see: Section A-9.8.2)
NilThread	The NilThread Constructor (see: Section A-9.8.3)
MySelf	The MySelf Constructor (see: Section A-9.8.4)
AnyThread	The AnyThread Constructor (see: Section A-9.8.5)
WaitNotify	The WaitNotify Constructor (see: Section A-9.8.6)

The **CapId** type describes a value that uniquely identifies a capability within the capability space of the caller. A capability may address a thread or kernel object with implicit restrictions based on the access controls of the capability. Capabilities are further described in Chapter A-4.

A-9.8.1 The CapId Constructor

Field:	Type:	Description:
Type	**CapIdType**	Capability identifier type
Index	**Word**	Capability identifier index

The CapId constructor is used to identify a kernel capability and consists of a *type* and an *index*. The type allows the user to specify the class of caabilities to which this particular capability belongs and the *index* field is used to specify the capability number within the specified class.

A-9.8.2 The ThreadId Constructor

Field:	Type:	Description:
Type	**CapIdType**	This value is set to 0
Index	**Word**	Thread identifier

The ThreadId constructor identifies a particular user thread and consists of a *type* and an *index*. The type is set to 0 and the index is used to specify the thread identifier of the particcular thread. Threads are further described in Chapter A-2.

Thread identifiers are managed internally by the microkernel. Thread identifiers define a fixed set of threads available for use in the system. The maximum thread number is dependent on the size of the capability list associated with the address space to which the thread belongs.

A-9.8.3 The NilThread Constructor

The NilThread constructor is used to represent a non-existent *nil thread*. This constructor is used for special purposes including specifiying system call arguments and disabling certain features.

A-9.8.4 The MySelf Constructor

The MySelf constructor is used to represent the current thread. This constructor is used for special purposes including specifying system call argument whe the thread is unaware of its thread identifier.

A-9.8.5 The AnyThread Constructor

The AnyThread constructor is used in the Ipc system call to match any active thread in the system as described in Section A-12.5. Supplying it to any other system call or OKL4 virtual register results in an error.

A-9.8.6 The WAITNOTIFY Constructor

The WAITNOTIFY constructor is a special encoding used in the Ipc system call to indicate a *blocking receive* operation that only accepts asynchronous notification messages, as described in Section A-12.5. Supplying it to any other system call or OKL4 virtual register results in an error.

A-9.9 CAPIDTYPE

Constructor:	Description:
ACCESS	Generic Capability List Index Class
TCB	TCB resource class
MUTEX	Mutex resource class
SPACE	Address space resource class
CAPABILITYLIST	Capability list resource class
PHYSSEGMENT	Physical memory segment resource class
REPLYCAPABILITY	IPC reply capability class
SPECIAL	Special identifiers class

A-9.10 CAPOP

Constructor:	Description:
CREATECLIST	Create a capability list
DELETECLIST	Delete an empty capability list
CREATEIPCCAP	Create an IPC capability
DELETECAP	Delete a capability

The **CAPOP** type specifies the operation to be performed by the CapControl system call. The CapControl system call is further described in Section A-12.2.

The encodings for the **CAPOP** constructors are defined in B-2.2.4.

A-9.11 CODE

The **CODE** type describes a sequence of machine-executable instructions. A **CODE** object is not a procedure in the sense of the C/C++ nomenclature. It is a sequence of instructions that take no parameters and never returns. The object makes no assumptions about the state of the input stack. The correct usage of this type depends on context, and is described separately for each appearance of the **CODE** type in this manual. Since **CODE** objects are architecture-specific, they are not constructed in the abstract OKL4 API. In Part C of this manual, such objects are presented using the native assembly language syntax of the underlying architecture. This type is typically implemented as the address of the first instruction in the sequence.

A-9.12 COPFLAGS

Constructor:	Description:	
COPFLAGS	The COPFLAGS Constructor	
Data Field:	**Type:**	**Description:**
C0	**FLAG**	Coprocessor 0
C1	**FLAG**	Coprocessor 1
C2	**FLAG**	Coprocessor 2
C3	**FLAG**	Coprocessor 3
C4	**FLAG**	Coprocessor 4
C5	**FLAG**	Coprocessor 5
C6	**FLAG**	Coprocessor 6
C7	**FLAG**	Coprocessor 7

The **COPFLAGS** type is used to represent the content of the CopFlags register defined in Section A-11.2. The *Ci* flags represent subsets of the functionality provided by individual processing units in the system which may be selectively disabled by threads to improve overall system performance. The CopFlags register is further described in Section A-11.2. OKL4 bindings for individual architectures are covered in Part C of this manual.

A-9.13 CLISTID

Constructor:	Description:	
CLISTID	The CLISTID Constructor	
Data Field:	**Type:**	**Description:**
ClistNo	**WORD**	Clist number

The **CLISTID** type is a refinement of the **WORD** type used to represent capability list identifiers. This type is restricted to integer values within the range *0 to MaxClists - 1*, inclusive.

A-9.14 DATE

Constructor:	Description:	
DATE	The DATE Constructor	
Data Field:	**Type:**	**Description:**
Year	**WORD**$_7$	Year component
Month	**WORD**$_4$	Month component
Day	**WORD**$_5$	Day component

The **DATE** type is used to represent calendar dates in the OKL4 API. It is capable of representing all dates from 2000-01-01 to 2127-12-31, inclusive. For example, DATE(01,2,29) corresponds to 2001-02-29 and is illegal. In this manual date objects are presented in the ISO YYYY-MM-DD notation.

A-9.15 ERRORCODE

Constructor:	Description:
INVALIDTHREAD	Invalid thread specified
INVALIDSPACE	Invalid address space specified
INVALIDPARAMETER	Invalid system call parameter value
INVALIDUTCB	Invalid UTCB / UTCB area location specified
OUTOFMEMORY	Insufficient system resources to complete the operation
SPACENOTEMPTY	Attempting to delete a space with threads associated with it
CLISTNOTEMPTY	Attempting to delete a capability list that is not empty
INVALIDMUTEX	Invalid mutex specified
MUTEXBUSY	Specified mutex is currently locked
INVALIDCLIST	Invalid capability list specified
INVALIDCAP	Invalid capability specified
DOMAINCONFLICT	Attempting to add a window to a memory region that is not empty
NOTIMPLEMENTED	System call does not exist
IPCERROR	An IPC operation failed (see: Section A-9.15.1)

The **ERRORCODE** type is used to describe the cause of failure by OKL4 system calls. This is provided to user programs via the the ErrorCode register described in Section A-11.3. Error codes are further described in Chapter A-12.

A-9.15.1 The IPCERROR Constructor

Field:	Type:	Description:
Cause	**IPCERRORCODE**	Error code
P	**FLAG**	Phase flag

The IPCERROR constructor is used to describe the cause of failure during an IPC operation. The phase flag is set if the error occurred during the *receive* phase of an IPC operation and is cleared for the *send* phase. IPC operations are further described in Chapter A-6 and Section A-12.5.

The encodings for the *Cause* field are defined in B-2.2.8.

A-9.16 FLAG

Constructor:	Description:
TRUE	TRUE constructor
FALSE	FALSE constructor

The **FLAG** type is used to represents "on/off" flags, where "on" and "off" states are represented by the TRUE and FALSE constructors, respectively. A **FLAG** object with a TRUE value is said to be "set", and an object with a FALSE value is said to be "cleared."

The encodings for the **FLAG** type are defined in B-2.2.7.

A-9.17 FPAGE

Constructor:	Description:
FPAGE	A proper subset of the address space (see: Section A-9.17.1)
WHOLESPACE	The complete address space (see: Section A-9.17.2)
NILPAGE	An empty subset of the address space (see: Section A-9.17.3)

OKL4 manages regions of an address space using objects known as *flexible pages* (or *fpages*). An fpage describes both the region of an address space and the permissions required for that region.

OKL4 fpages are not objects in their own right, but rather a pair associating a set of permissions with a collection of consecutive, suitably aligned page-sized objects. It is used by OKL4 system calls to manipulate address spaces as described in Section A-12.6. The use of the permission set associated with a region by an fpage is dependant on the context and is described for each use in this manual.

As a design implication, a single program may refer to a particular region of memory through a variety of fpages of different sizes throughout its lifetime. At any point, the effect is as if all bytes of the user areas covered by the fpage were supplied individually to the OKL4 system call or virtual register. In particular, use of overlapping fpages is permitted without restrictions, and later uses of the intersecting regions take precedence over earlier uses of the same region. Address spaces are further described in Chapter A-3.

A-9.17.1 The FPAGE Constructor

Field:	Type:	Description:
Base	**PAGENO**	The page number (b)
Width	**WORD$_6$**	Width of the page (w)
Meta	**FLAG**	Extended rights attribution
Access	**PERMISSIONS**	The required permissions (P)

The FPAGE constructors associate permissions P with all bytes of memory in the region from b to $b + 2^w - 1$, inclusive, excluding any bytes on pages within that region belonging to the reserved areas of the address space. The OKL4 API restricts the parameters b and w as follows:

$$\mathsf{MinPageWidth} \leq w < k \quad \text{and} \quad b \,\mathbf{mod}\, 2^w = 0$$

where k has the value of 32 and 64 on 32-bit and 64-bit OKL4 architectures, respectively. The second equation restricts fpage objects to be aligned on 2^w boundaries in memory. Both restrictions are imposed to enable the efficient implementation of the OKL4 API on a variety of architectures.

A-9.17.2 The WHOLESPACE Constructor

Field:	Type:	Description:
Access	**PERMISSIONS**	The required permissions (P)

The WHOLESPACE constructor associates the complete address space with the permission set P, excluding any reserved regions and architecture-defined regions as described in Part C of this manual.

A-9.17.3 The NILPAGE Constructor

The NILPAGE constructor associates an empty region of memory with the empty set of permissions $\varnothing$. It is used to indicate an absence of an object or a feature. The semantics of this constructor are defined for each use of NILPAGE in this manual.

A-9.18 HWTHREADMASK

The HwThreadMask type contains the set of *hardware threads* scheduling parameter which is used to specify the set of processing units within a domain that an OKL4 thread may execute on. Each bit represents a processing unit (or hardware thread) which when set, enables execution on that processing unit.

The **HWTHREADMASK** type is used to configure part of the *allowed processing units* scheduling parameter using the Schedule system call described in Section A-12.11.

A-9.19 HYBRIDMUTEX

Constructor:	**Description:**
HYBRIDMUTEX	The HYBRIDMUTEX Constructor

Data Field:	**Type:**	**Description:**
C	**FLAG**	Contention status.
Holder	**THREADHANDLE**	Thread handle of current mutex holder.

The **HYBRIDMUTEX** type is used to define the hybrid mutex state shared between the kernel and user. Thus it must be updated using atomic operations. The **HYBRIDMUTEX** type consists of the status *C* flag, and the *Holder* thread handle which indicate the state of the mutex. The *C* flag may only be manipulated by the kernel and indicates that the mutex's current contention stat. When the *C* flag is set the mutex is contended and has threads blocked on it. The *Holder* field may be manipulated by both the user and kernel, and indicates both the availability of the lock and the holding thread if the lock is held.

A-9.20 IA32SEGMENTDESCRIPTOR

Constructor:	**Description:**
IA32SEGMENTDESCRIPTOR	The IA32SEGMENTDESCRIPTOR Constructor

Data Field:	**Type:**	**Description:**
limitlow	**WORD**$_{16}$	Low part of limit
baselow	**WORD**$_{24}$	Low part of base
type	**WORD**$_{4}$	Type of descriptor
s	**WORD**$_{1}$	System segment bit
dpl	**WORD**$_{2}$	Priviledge bit
p	**WORD**$_{1}$	Present bit
limithigh	**WORD**$_{4}$	High part of limit
avl	**WORD**$_{2}$	Low part of limit
g	**WORD**$_{1}$	Byte or page size limit
d	**WORD**$_{1}$	16 or 32-bit segment
basehigh	**WORD**$_{8}$	High part of the base

The **IA32SEGMENTDESCRIPTOR** type is used to provide a raw description of the segment descriptor, as specified by the architecture manual for the IA32 processor, with the exception of the *s* bit and the *dpl* bit. As such this type is only relevant to the IA32 processor. It should be noted that the only valid value of the *s* bit is 1 and the only valid value of the *dpl* bit is 3.

A-9.21 INTCONTROL

Constructor:	Description:	
INTCONTROL	The INTCONTROL Constructor	
Data Field:	**Type:**	**Description:**
Count	**WORD**$_6$	Highest message register
Operation	**WORD**$_2$	Operation type
Request	**WORD**$_{18}$	Platform specific request parameter
NotifyBit	**WORD**$_5$	Asynchronous notify bit requested

The **INTCONTROL** type is used to define the content of the *Control*$_{\text{IN}}$ input parameter to the InterruptControl system call. The operation types are further described in Section B-2.2.9. The InterruptControl system call is further described in Section A-12.4

A-9.22 IPCERRORCODE

Constructor:	Description:
NOPARTNER	No asynchronous notification partner
INVALIDPARTNER	Target thread does not exist
MESSAGEOVERFLOW	Insufficient number of MessageData registers
IPCREJECTED	Asynchronous notification rejected by the recipient
IPCCANCELLED	IPC operation has been cancelled
IPCABORTED	IPC operation has been aborted

The **IPCERRORCODE** is used to define the content of the *Cause* field in the IPCERROR constructor defined in Section A-9.15.1. It describes the cause of failure during an Ipc system call as described in Section A-12.5. The OKL4 IPC mechanism is further discussed in Chapter A-6.

The encodings for the **IPCERRORCODE** type are defined in Section B-2.2.5.

A-9.23 MAPCONTROL

Constructor:	Description:	
MAPCONTROL	The MAPCONTROL Constructor	
Data Field:	**Type:**	**Description:**
Count	**WORD**$_6$	Number of map items - 1
Win	**FLAG**	Perform window mapping
Q	**FLAG**	Query current mapping data
M	**FLAG**	Modify mapping data

The **MAPCONTROL** type is used to define the content of the *Control*$_{\text{IN}}$ input parameter to the MapControl system call. It should be noted that when **WIN** is set, the flags Q and M are ignored and should be cleared. The MapControl system call is further described in Section A-12.6.

A-9.24 MAPITEM

Constructor:	Description:	
MAPITEM	The MAPITEM Constructor	
Data Field:	**Type:**	**Description:**
Virt	**VIRTUALDESCRIPTOR**	Virtual descriptor
Size	**SIZEDESCRIPTOR**	Size descriptor
Seg	**SEGMENTDESCRIPTOR**	Segment descriptor

The **MAPITEM** type is used to provide input for address space create, remove and modify operations issued by the MapControl system call described in Section A-12.6.

The **VIRTUALDESCRIPTOR**, **SIZEDESCRIPTOR** and **SEGMENTDESCRIPTOR** types are described in Sections A-9.44, A-9.38, A-9.37, respectively.

A-9.25 MEMORYCOPYDESCRIPTOR

Constructor:	Description:	
MEMORYCOPYDESCRIPTOR	The MEMORYCOPYDESCRIPTOR Constructor	
Data Field:	**Type:**	**Description:**
Address	**WORD**	Address of remote buffer
Size	**WORD**	Size of remote buffer
Direction	**WORD**$_2$	MemoryCopy directions

The **MEMORYCOPYDESCRIPTOR** type is used to specify the location and size of the buffer of the remote thread as well the permitted memory copy directions.

A-9.26 MESSAGETAG

Constructor:	Description:
MSGTAGIN	Sending Message Tag (see: Section A-9.26.1)
MSGTAGOUT	Received Message Tag (see: Section A-9.26.2)

The **MESSAGETAG** is used to describe the content of the MessageData$_0$ register supplied to and returned by the Ipc system call. Since the **MESSAGETAG** is used for both IPC input and output, constructors are defined for these two cases. The Ipc system call is further described in Section A-12.5 and Chapter A-6.

A-9.26.1 The MSGTAGIN Constructor

Field:	Type:	Description:
Untyped	**WORD**$_6$	Number of data words
Label	**WORD**$_{16}$	Message label
S	**FLAG**	The *send* blocking flag
R	**FLAG**	The *receive* blocking flag
N	**FLAG**	The asynchronous notification flag
M	**FLAG**	The memory copy flag

A-9.26.2 The MSGTAGOUT Constructor

Field:	Type:	Description:
Untyped	**WORD**$_6$	Number of data words
Label	**WORD**$_{16}$	Message label
E	**FLAG**	The error flag
X	**FLAG**	The remote-IPC flag

A-9.27 MUTEXCONTROL

Constructor:	Description:	
MUTEXCONTROL	The MUTEXCONTROL Constructor	
Data Field:	**Type:**	**Description:**
Create	**FLAG**	Create new mutex
Delete	**FLAG**	Delete existing mutex

The **MUTEXCONTROL** type is used to describe the architecture defined control word that is supplied as an argument to the MutexControl system call. The **MUTEXCONTROL** type coonsits of the flags *create* and *delete* which are used to indicate the desired operation to the MutexControl system call.

A-9.28 MUTEXFLAGS

Constructor:	Description:	
MUTEXFLAGS	The MUTEXFLAGS Constructor	
Data Field:	**Type:**	**Description:**
T	**FLAG**	Operation type, aquire or release
B	**FLAG**	The *aquire* blocking flag
H	**FLAG**	Mutex type

The **MUTEXFLAGS** type is used to describe the value of an architecture defined control word that is supplied as an argument to the Mutex system call. The **MUTEXFLAGS** type consists of the flags *T*, *B* and *H*. The The flags *T* and *B* are used to indicate the type of the operation, (set for *acquire*), and whether the call is blocking (set for blocking), respectively. The *H* flag is used to indicate whether the mutex is a hybrid (kernel and user) or kernel only mutex.

A-9.29 MUTEXID

Constructor:	Description:	
MUTEXID	The MUTEXID Constructor	
Data Field:	**Type:**	**Description:**
MutexNo	**WORD**	Mutex number

The **MUTEXID** type is a refinement of the **WORD** type used to represent mutex identifiers. This type is restricted to integer values within the range *0 to MaxMutexes - 1*, inclusive.

A-9.30 PAGENO

The **PAGENO** type is a refinement of the **WORD** type used to represent 10-bit (2^{10} bytes) sized *page numbers*. It is equivalent to dividing the address of the page by 2^{10} bytes, or *address* >> 10. It is restricted to integer values that are an integral multiple of the corresponding *page size*, that is, a page number represents the location of the first byte of the page. The determination of the page size is dependent on the context in which **PAGENO** is used and is described separately for each use of this type. Since **PAGENO** refines the **WORD** type, it is subject to the notational conventions and restrictions of the **WORD** type described in Section A-9.45.

A-9.31 PERMISSIONS

Constructor:	**Description:**	
PERMISSIONS	The PERMISSIONS Constructor	
Data Field:	**Type:**	**Description:**
R	**FLAG**	*read* permission
W	**FLAG**	*write* permission
X	**FLAG**	*execute* permission

The **PERMISSIONS** type describes a set of memory access operations that are supported by a particular page object as discussed in Sections A-3.2. In this manual, permission objects are presented as a set of fields that have been given TRUE values in the constructor. For example, the commonly-used constructor PERMISSIONS(TRUE, FALSE, TRUE) represents read/execute permissions and is presented as $\{read, execute\}$ and PERMISSIONS(FALSE, FALSE, FALSE) is presented as $\emptyset$.

A-9.32 PLATCONTROL

Constructor:	**Description:**	
PLATCONTROL	The PLATCONTROL Constructor	
Data Field:	**Type:**	**Description:**
Count	**WORD**$_6$	Number of input parameters
Request	**WORD**$_{26}$	Implementation specific

The **PLATCONTROL** type is used to define the content of the *Control*$_{\text{IN}}$ input parameter to the PlatformControl system call. The PlatformControl system call is further described in Section A-12.10.

A-9.33 PREEMPTIONFLAGS

Constructor:	**Description:**	
PREEMPTIONFLAGS	The PREEMPTIONFLAGS Constructor	
Data Field:	**Type:**	**Description:**
Signalled	**FLAG**	The pre-emption notification flag

The **PREEMPTIONFLAGS** type is used to define the content of the PreemptionFlags register described in Section A-11.14. Specifically, it contains the *Signalled* flag used to request the delivery of pre-emption signals described in Section A-5.7.

A-9.34 QueryDescriptor

Constructor:	Description:	
QueryDescriptor	The QueryDescriptor Constructor	
Data Field:	**Type:**	**Description:**
Access	**Permissions**	Requested permissions for the mapping
PageSize	**Word**$_6$	Size of pages to be used in mapping (power of 2)
PageRef	**Permissions**	Current referenced bits

A-9.35 QueryItem

Constructor:	Description:	
QueryItem	The QueryItem Constructor	
Data Field:	**Type:**	**Description:**
Virt	**VirtualDescriptor**	Virtual descriptor
Query	**QueryDescriptor**	Query descriptor
Seg	**SegmentDescriptor**	Segment descriptor

A-9.36 RegControl

Constructor:	Description:	
RegControl	The RegControl Constructor	
Data Field:	**Type:**	**Description:**
HE	**Flag**	Enable halt/resume operation
AS	**Flag**	Abort send operation
AR	**Flag**	Abort receive operation
SP	**Flag**	Set StackPointer
IP	**Flag**	Set InstructionPointer
FL	**Flag**	Set Flags
UH	**Flag**	Set UserHandle
CPY	**Flag**	Copy thread's registers
Rget	**Flag**	Deliver thread's registers
Rset	**Flag**	Modify thread's registers
D	**Flag**	Deliver old register values
H	**Flag**	Halt/Resume
Tls	**Flag**	Set pointer to TLS
Source	**Word**	Source thread number

The **RegControl** type is used to represent the *Control*$_{\text{IN}}$ input parameter to the ExchangeRegisters system call. It contains a separate flag for each operation supported by the system call, allowing the user to describe the set of operations required on the target thread. **RegControl** is also used to describe the state of the target thread in the *Control*$_{\text{OUT}}$ output parameters of the ExchangeRegisters system call. The function of each **RegControl** field is described in Section A-12.3.

A-9.37 SEGMENTDESCRIPTOR

Constructor:	Description:
SEGMENTDESCRIPTOR	The SEGMENTDESCRIPTOR Constructor

Data Field:	Type:	Description:
Segment	**WORD**$_{10}$	Memory to be mapped
Offset	**WORD**$_{22}$	Offset from the start of physical memory to be mapped

A-9.38 SIZEDESCRIPTOR

Constructor:	Description:
SIZEDESCRIPTOR	The SIZEDESCRIPTOR Constructor

Data Field:	Type:	Description:
Access	**PERMISSIONS**	Requested permissions for the mapping
Valid	**FLAG**	Map (1) or unmap (0) operation
PageSize	**WORD**$_{6}$	Size of pages to be used in mapping (power of 2)
NumPages	**WORD**$_{22}$	Number of pages to be mapped

A-9.39 SPACECONTROL

Constructor:	Description:
SPACECONTROL	The SPACECONTROL Constructor

Data Field:	Type:	Description:
New	**FLAG**	Create new address space
Delete	**FLAG**	Delete address space
Resources	**FLAG**	Modify and retrieve old space resources
Resourced	**FLAG**	Give new space access to resources
MaxPriority	**WORD**$_{8}$	Maximum priority threads in space can assign

The **SPACECONTROL** type describes the value of an architecture-defined control word that is supplied to the SpaceControl system call described in Section A-12.12. The **SPACECONTROL** type consists of the flags, *New, Delete, Resources Resourced* and *MaxPriority*. The flags *New, Delete, Resources* and *Resourced* are used to indicate the desired operation to the SpaceControl system call.

The value of *MaxPriority* is used to indicate the maximum priority threads residing in a newly created address space can assign to other threads. The maximum priority of an address space cannot be altered or set on an existing address space and must be less than or equal to the maximum priority of the calling address space. Note that the newly created address space may subsequently contain threads that have a priority greater than its maximum priority. This may be as a result of a thread residing in another address with a higher maximum priority assigning a similar priority to thread in question.

A-9.40 SPACEID

Constructor:	Description:
NILSPACE	The NILSPACE Constructor (see: Section A-9.40.1)

The **SPACEID** type is a refinement of the **WORD** type used to represent address space identifiers (space IDs). It is restricted to values which are valid within the range 0 ... MaxSpaces − 1.

A-9.40.1 The NILSPACE Constructor

The NILSPACE constructor is used to represent a non-existent address space identifier. This constructor is used for special purposes includig specifying system call arguments for thread deletion and disabling certain features.

A-9.41 SPACERESOURCES

Constructor:	Description:
DEFAULTSPACERESOURCES	Default space resources word

The **SPACERESOURCES** type describes the value of an architecture-defined control parameter that is supplied to the SpaceControl system call. The DEFAULTSPACERESOURCES constructor specifies the default space resources semantics supported by all OKL4 architectures. Additional constructors for **SPACERESOURCES** are defined separately for each architecture.

A-9.42 THRDRESOURCES

Constructor:	Description:
DEFAULTTHREADRESOURCES	Default thread resources word

The **THRDRESOURCES** type describes the value of an architecture-defined control parameter that is supplied to the ThreadControl system call described in Section A-12.14. The DEFAULTTHREADRESOURCES constructor specifies the default thread resources semantics supported by all OKL4 architectures. Additional constructors for **THRDRESOURCES** are defined separately for each architecture in Part C of this manual.

A-9.43 THREADSTATE

Constructor:	Description:
UNKNOWNSTATE	Unknown thread status
DEAD	Deleted / Not configured
INACTIVE	Temporarily suspended
RUNNING	Thread is runnable
POLLING	Pending IPC *send* operation
SENDING	IPC *send* operation in progress
WAITING	Pending IPC *receive* operation
RECEIVING	IPC *receive* operation in progress
WAITINGNOTIFY	Pending IPC *receive* of asynchronous notification operation
WAITINGMUTEX	Pending Mutex *lock* operation

The **THREADSTATE** type is used to represent states of OKL4 threads as described in Section A-2.1. The current state of any user thread may be obtained using the Schedule system call described in Section A-12.11.

A-9.44 VIRTUALDESCRIPTOR

Constructor:	Description:	
VIRTUALDESCRIPTOR	The VIRTUALDESCRIPTOR Constructor	
Data Field:	**Type:**	**Description:**
Attr	**WORD**$_8$	Caching attributes of the mapping
Replace	**FLAG**	Replace an existent mapping
Sparse	**FLAG**	Sparse unmapping
VirtualBase	**WORD**$_{22}$	Base of the virtual address

A-9.45 WORD

Constructor:	Description:
MAXWORD	Largest number representable as a **WORD** object

The **WORD**$_k$ type describes a non-negative number (natural number) with a value between 0 and $2^k - 1$, inclusive. When the subscript k is omitted, it is has a value of 32 and 64 on 32-bit and 64-bit implementations, respectively. Note that the **WORD** values that are valid in a particular context are often further restricted to a subset of the above range. In this manual, objects of type **WORD** are presented directly as binary, decimal or hexadecimal numerals. MAXWORD$_k$ is used to represent the largest natural number representable as a **WORD**$_k$ object, that is $2^k - 1$. MAXWORD is equal to $2^{32} - 1$ and $2^{64} - 1$ on 32-bit and 64-bit OKL4 implementations, respectively.

A-10 System Parameters

In order to simplify the implementation of OKL4 on specific architectures, the generic OKL4 API microkernel API allows a degree of flexibility in meeting specification requirements. This is accomplished using a set of *system parameters* or *OKL4 API parameters*. OKL4 API parameters are constants whose values are chosen from a specified set by each specific OKL4 architecture. The value of each OKL4 API parameter must be documented and made available to programs at run-time through the corresponding language bindings described in Part D.

Note that a majority of system parameters have been included in the OKL4 API for information purposes only. They may be valuable when designing operating system software that is capable of executing on a number of hardware architectures. As these constants do not increase the complexity of the microkernel, they have been included in the the API for programming convenience.

A-10.1 MaxClists

Type: WORD

The MAXCLISTS parameter defines the maximum number of capability lists supported by OKL4. The MAXCLISTS parameter is used to determine the range of valid Clist IDs. Clist IDs are described in Chapter A-4.

A-10.2 MaxMessageData

Type: WORD$_6$

The MaxMessageData parameter provides the number of MessageData virtual registers by specifying the highest MessageData register number supported by the implementation. The MessageData registers are numbered from 0 to MaxMessageData inclusive. The OKL4 sets the MaxMessageData parameter to an integer between 7 and 63 inclusive, which is defined in the Part C part of the manual.

A-10.3 MaxMutexes

Type: WORD

The MAXMUTEXES parameter defines the maximum number of mutexes supported by OKL4. The MAXMUTEXES parameter is used to determine the range of valid Mutex IDs. Mutex IDs are described in Chapter A-7.

A-10.4 MaxSpaces

Type: WORD

The MAXSPACES parameter defines the maximum number of address spaces supported by OKL4. The MAXSPACES parameter is used to determine the range of valid space IDs. Space IDs are described in Section A-3.1.

A-10.5 PageSizeMask

Type: PAGENO

The PageSizeMask parameter describes the values of the FPAGE.*Width* field that are supported directly by the memory management hardware of the architecture. The OKL4 microkernel is required to handle all values of the FPAGE.*Width* field within the range described in Section A-9.17. Users may assume that the values listed in PageSizeMask will be handled more efficiently than other values.

The value of PageSizeMask is formed by a bit-wise inclusive OR of all page size values (assumed to be an integral power of 2) supported by the memory management hardware. For example, on a hardware that supports only 4KB and 4MB pages directly, the PageSizeMask parameter would have the value of 4KB | 4MB, or `100·1000`$_{16}$ in hexadecimal.

A-10.6 PMask

Type:	PERMISSIONS

The PMask parameter is used to compute the *effective permissions* of a given **FPAGE** object as described in Section A-9.17. This is due to the fact that most architectures do not support all eight combinations of the three memory access permissions (*read*, *write* and *execute*) recognized by the OKL4 microkernel.

The PMask parameter contains the set of permissions that are not added implicitly to any permission set that explicitly specifies at least one other permission. That is, if set P is the set of explicitly specified permissions, and Q is the set of permissions specified by the PMask parameter, the set P' of effective permissions is computed as follows:

$$P' = \begin{cases} \varnothing, & \text{if } P = \varnothing; \\ P \cup \overline{Q}, & \text{otherwise.} \end{cases}$$

where $\overline{Q}$ represents the complement of the set Q, i.e. $\{read, write, execute\} \setminus Q$.

A-10.7 UtcbAlignment

Type:	WORD$_6$

The parameters UtcbAlignment and UtcbMultiplier specify the size of the UTCB entries described in Section A-3.5. If the UtcbMultiplier has the value m and UtcbAlignment has the value k, the total size of each UTCB entry, including padding, is $m \times 2^k$ bytes.

A-10.8 UtcbRegionWidth

Type:	WORD$_6$

The UtcbRegionWidth parameter specifies the smallest valid value of the FPAGE.*Width* field for an fpage intended for the UTCB area of an address space, as described in Section A-3.5. If the UtcbRegionWidth contains a value of 0, the UTCB area resides outside of the mappable user region of the address space and is managed automatically by the microkernel.

A-10.9 UtcbMultiplier

Type:	WORD$_{10}$

The two parameters UtcbAlignment and UtcbMultiplier specify the size of a single UTCB entry in the UTCB area of an address space as described in Section A-10.7.

A-11 Virtual Registers

Each thread is associated with a number of *virtual registers*. This chapter provides a list of all virtual registers defined by the OKL4 microkernel. The type of data values stored and the access type are also specified for each register.

This list includes the stack and instruction pointer registers of the thread. These registers are not part of the microkernel interface and may be viewed as “placeholders” for certain special-purpose hardware registers and features. They have only been included for the purpose of clarity as the microkernel provides a semi-portable means of controlling them on all architectures. The implementation of *placeholder registers* is further described in Part C of this manual.

This list groups together registers with uniform properties. The individual registers in the group are denoted by the name of the group followed by a numeric subscript, for example MessageData_7. These registers are often classed as being of the type **WORD**. It should be noted that the type of objects stored in the register may be of a different type depending on the use of the register.

A-11.1 Acceptor

Type:	ACCEPTOR
Access:	read/write

The Acceptor register is used to request the delivery of special message types. In the current API, this only concerns delivery of asynchronous notification flags to the thread, both asynchronously via the NotifyFlags and synchronously as IPC messages. This is further described in Sections A-11.9 and A-6.4.

A-11.2 CopFlags

Type:	WORD$_8$
Access:	read/write

The CopFlags register may be used to selectively disable parts of the processing unit for the duration of the execution of the current thread. The meaning of this register is architecture-defined and described in Part C of this manual. In general, individual flags in this register should be updated using an atomic read-and-update instruction as some fields may affect the integrity of the thread for the duration of the update.

A-11.3 ErrorCode

Type:	ERRORCODE
Access:	read-only

The ErrorCode register contains the error code returned by the most recent system call. The value of this register is undefined after a successful system call. Typically the value remains unchanged until an error occurs during a system call issued by the corresponding thread.

A-11.4 ExceptionHandler

Type:	THREADID
Access:	not-directly-accessible

The ExceptionHandler register contains the global thread ID of the thread that receives exception messages from the current thread as described in Section A-2.3.

A-11.5 Flags

Type:	WORD
Access:	read/write

The Flags register is a *placeholder* for an architecture-defined hardware register. It typically represents control flags containing the result of certain arithmetic instructions such as those that may cause numeric overflow. This register is included in the generic OKL4 API as it is modified by the generic ExchangeRegisters system call. This is further described in Part C of this manual.

A-11.6 InstructionPointer

Type:	CODE
Access:	read/write

The InstructionPointer register is a *placeholder* for an architecture-defined hardware register. It contains the address of the next instruction to be executed by the current thread.

A-11.7 MessageData

Type:	WORD
Access:	read/write

$\mathsf{MessageData}_0$ to $\mathsf{MessageData}_{\mathsf{MaxMessageData}}$ are a group of transient registers. They are used by the CacheControl, Ipc and MapControl system calls described in Sections A-12.1, A-12.5 and A-12.6. There is an architecture-defined number of message registers, however all OKL4 kernels must provide at least 8 message registers and a mechanism for accessing and modifying the values stored in them. As they are transient, any machine instruction may alter the values stored in these registers, with the following exceptions:

- The documented method for accessing and modifying the value of a particular message register will not modify the value of any other message register.
- The Ipc and MapControl system calls modify the values of the message registers in a predictable manner as described in Sections A-12.5 and A-12.6.

Note that an attempt to access the value of a particular message register may modify the its value. Therefore the value of a message register may only be accessed once before it is destroyed.

A-11.8 MyThreadHandle

Type:	THREADID
Access:	read-only

The MyThreadHandle register provides access to the global thread handle of the currently executing thread. This register is only used by OKL4 system libraries and should be considered to be deprecated and not used in user code.

A-11.9 NotifyFlags

Type:	WORD
Access:	read/write

The NotifyFlags register contains the accumulated set of *asynchronous notification flags* received by the thread. The set is defined as an inclusive bit-wise OR of the values of the form 2^k, for all values of k corresponding to the asynchronous notification bits that have been received by the thread. This is further described in Section A-6.4.

A-11.10 NotifyMask

Type:	WORD
Access:	read/write

The NotifyMask register contains the set of *asynchronous notification flags* accepted by an asynchronous IPC to the thread as described in Section A-6.4. This set is defined as an inclusive bit-wise OR of the values of the form 2^k, for all values of k corresponding to the asynchronous notification flags that the thread is willing to receive.

A-11.11 Pager

Type:	THREADID
Access:	read-only

The Pager register contains the global thread ID of the thread that receives page fault messages from the current address space as described in Sections A-3.2 and A-13.2.

A-11.12 Parameter

Type:	WORD
Access:	read/write

Parameter$_0$ to Parameter$_7$ are a group of transient registers, used to represent an architecture-defined method of supplying input parameters to system calls as described in Chapter A-12. Except as part of the system call sequence, language bindings to the OKL4 API are not required to provide a general-purpose method of modifying the values of these registers.

A-11.13 PreemptedIp

Type:	CODE
Access:	read-only

The PreemptedIp register contains the address of the thread's instruction pointer at the point at which the thread has been pre-empted by the microkernel. This is further described in Section A-5.1.

A-11.14 PreemptionFlags

Type:	PREEMPTIONFLAGS
Access:	read/write

The PreemptionFlags register contains the *pre-emption flags* described in Section A-9.33. It is used during thread pre-emption described in Section A-5.7.

A-11.15 PreemptCallbackIp

Type:	CODE
Access:	read/write

The PreemptCallbackIp register refers to the start address of code that is to be executed by a thread following a pre-emption when a pre-emption event notification has been enabled in the PreemptionFlags register as described in Section A-5.7.

A-11.16 Priority

Type:	WORD
Access:	not-directly-accessible

The Priority register contains a thread's current priority which was assigned using the scheduling API. This is further described in Section A-5.7.

A-11.17 ProcessingUnit

Type:	WORD
Access:	read-only

The ProcessingUnit register provides access to the processing unit that is executing the current thread. This register always contains a valid value while it is accessible by the thread. Note that the value may change asynchronously in an unpredictable manner due to the ability of the OKL4 microkernel to dynamically migrate threads between processing units.

A-11.18 Result

Type:	WORD
Access:	read-only

Result_0 to Result_7 are a group of transient registers. They are used to represent an architecture-defined method of supplying the output parameters of a system call as described in Chapter A-12. As with parameter registers, the OKL4 API language bindings are not required to provide a general-purpose method of modifying the values of these registers except as part of the system call sequence.

A-11.19 SenderSpace

Type:	SPACEID
Access:	read-only

The SenderSpace register contains the space thread ID of the thread that sent the IPC message. This is useful for pager threads.

A-11.20 StackPointer

Type:	WORD
Access:	read/write

The StackPointer is a placeholder register for an architecture-specific hardware register. It contains the address of the top of the control stack required for the execution of nested procedure calls.

A-11.21 ThreadWord

Type:	WORD
Access:	read/write

ThreadWord$_0$ to ThreadWord$_{\text{MaxThreadWords}-1}$ are a group of registers that may be used by applications to store a arbitrary words of data.

OKL4 does not guarantee providing any user data registers. All OKL4 implementations do however provide a single UserHandle register. See below. Most architecture implementations provide one or more user data registers as described in Part C of this manual. These registers cannot be accessed outside of the corresponding thread.

A-11.22 UserHandle

Type:	WORD
Access:	read/write

The UserHandle register may be used by applications to store an arbitrary word of data. UserHandle can additionally be set or inspected using the ExchangeRegisters system call described in Section A-12.3.

A-11.23 Utcb

Type:	WORD
Access:	read-only

The Utcb register contains the location of the UTCB entry of the currently-executing thread. Note that on some architectures, this may not point directly at the start of the UTCB location.

A-12 System Calls

This chapter defines the set of OKL4 *system calls*. Each system call is presented as a function taking zero or more *input parameters* and providing zero or more *output parameters*. Any side-effects of a system call are specified in its description.

The microkernel obtains the input parameters from and stores the output parameters in the virtual registers specified in the system call description. Inputs are typically retrieved from the Parameter registers described in Section A-11.12 and outputs are typically stored in the Result registers described in Section A-11.18. Some OKL4 system calls accept and return a variable number of parameters. These parameters are always supplied and returned in the MessageData registers described in Section A-11.7. The number of registers used in a particular invocation of a system call is supplied in one of the "fixed" parameter registers.

Parameter names are annotated with the subscript *i* which iterates over the indices of the parameters used in a particular invocation of a system call. For example, if a particular system call invocation accepts three $\mathit{Data}_{\mathsf{IN}}$ parameters, these parameters are presented as $\mathit{Data}_{0_{\mathsf{IN}}}$, $\mathit{Data}_{1_{\mathsf{IN}}}$ and $\mathit{Data}_{2_{\mathsf{IN}}}$, and described collectively as $\mathit{Data}_{i_{\mathsf{IN}}}$. The subscript *i* is then used to derive their location in the MessageData register file. For example, if the above $\mathit{Data}_{\mathsf{IN}}$ parameters are stored in $\mathsf{MessageData}_1$, $\mathsf{MessageData}_3$ and $\mathsf{MessageData}_5$, respectively. The location of $\mathit{Data}_{i_{\mathsf{IN}}}$ will be presented as $\mathsf{MessageData}_{2i+1}$.

The implementation of these registers and the sequence of instructions used to invoke these system calls are described in Part C of this manual. Unless otherwise specified, all architecture and virtual registers not mentioned explicitly in the system call description will be preserved by the system call.

Some system calls are restricted to callers that have access to appropriate capabilities. An attempt to invoke this functionality by a caller that does not have access to appropriate capabilities is detected by the microkernel as an error. Capabilities are further described in Chapter A-4. System calls which do not have such restrictions may be executed by an arbitrary thread, though some features may be restricted to threads that have access to particular capabilities.

A-12.1 CacheControl

INPUT PARAMETERS

Parameter:	Description:	Type:	Register:
$\mathit{Target}_{\mathrm{IN}}$	Target address space	SPACEID	Parameter$_0$
$\mathit{Control}_{\mathrm{IN}}$	Control word	CACHECONTROL	Parameter$_1$

INPUT REGISTERS: MessageData$_0$ TO MessageData$_{2(\mathit{Control}_{\mathrm{IN}}.\mathit{Count})-1}$

$\mathit{RegionAddress}_{i_{\mathrm{IN}}}$	Location of an address range	WORD	MessageData$_{2i}$
$\mathit{RegionOp}_{i_{\mathrm{IN}}}$	Address range operation	CACHEREGIONOP	MessageData$_{2i+1}$

OUTPUT PARAMETERS

Parameter:	Description:	Type:	Register:
$\mathit{Result}_{\mathrm{OUT}}$	The result flag	FLAG	Result$_0$
$\mathit{ErrorCode}_{\mathrm{OUT}}$	The error code	ERRORCODE	ErrorCode

The CacheControl system call provides OKL4 threads with a means of controlling the data and instruction caches. It supports *clean* and *invalidate* cache operations. Each operation may be applied selectively to a subset of an address space by specifying a list of memory locations in the $\mathit{RegionAddress}_{i_{\mathrm{IN}}}$ parameters, and their sizes in the corresponding $\mathit{RegionOp}_{i_{\mathrm{IN}}}.\mathit{Size}$ parameters. The number of specified regions is contained in the $\mathit{Control}_{\mathrm{IN}}.\mathit{Count}$ parameter field.

The address space that contains these regions of memory is specified using its space ID in the $\mathit{Target}_{\mathrm{IN}}$ parameter. A *resourced* thread may perform any cache operation on an active address space to which it has access to the space id of the address space. A *non-resourced* thread may only perform *clean* and *clean+invalidate* operations on its own address space.

CacheControl permits up to six levels of the hardware cache hierarchy to be independently manipulated. The desired level can be selected using the $\mathit{Control}_{\mathrm{IN}}.\mathit{Li}$ flags (*L1* to *L6*). The instruction and data caches may be selected independently for each memory region using the $\mathit{RegionOp}_{i_{\mathrm{IN}}}.\mathit{IC}$ and $\mathit{RegionOp}_{i_{\mathrm{IN}}}.\mathit{DC}$ flags. The cache operations that may be specified in the $\mathit{Control}_{\mathrm{IN}}.\mathit{Op}$ field are:

FLUSHCACHE Perform any *clean* then *invalidate* operations as indicated by the corresponding $\mathit{RegionOp}_{i_{\mathrm{IN}}}.\mathit{C}$ and $\mathit{RegionOp}_{i_{\mathrm{IN}}}.\mathit{I}$ flags on the specified address space regions. These operations are further discussed in Sections A-12.1.1 and A-12.1.2.

FLUSHICACHE Perform the *clean* then *invalidate* operations on the entire instruction cache.

FLUSHDCACHE Perform the *clean* then *invalidate* operations on the entire data cache.

FLUSHALLCACHES Perform the *clean* then *invalidate* operations on all instruction and data caches.

Not all architectures support the entire range of cache operations described above. Therefore, individual machine architectures are permitted to substitute a *conservative* alternative for each caching operation supported by the CacheControl system call. These alternatives preserve the effects of cache operations that are visible to user programs. The alternatives for each operation are presented below.

A-12.1.1 Cleaning Cache Lines

The *clean* cache operation writes any data marked as "modified" to a higher level of the cache hierarchy. Data stored at the highest level of the cache is written to system memory. This may be done to ensure coherence of memory accesses across virtual aliases as well as different processing units in absence of a hardware-enforced cache coherency protocol, or as a performance optimization directing memory traffic to less "busy" periods during program execution. Cache coherency is further discussed in Section A-3.3.

The alternatives permitted for this operation include to clean, (or clean then invalidate) a larger region of the address space, or clean (or clean then invalidate) more levels of caching than requested by the user. OKL4 is also permitted to clean the cache at any time without the request of the user. Furthermore, OKL4 is also permitted to leave caches unmodified for any physical memory resources accessed under a hardware-enforced cache coherency protocol. Resource types and caching protocols are further discussed in Sections **??** and A-3.3.

A-12.1.2 Invalidating Cache Lines

The *invalidate* cache operation marks any cache lines that contain data of the requested regions of memory as "unused". This frees them for future use, bypassing the usual cache replacement policies used by the particular architecture. This operation is usually accompanied by a preceding *clean* operation on the same regions of the address space. When used together, *clean+invalidate* does not alter the semantics of user programs. Therefore this operation may be performed by any thread on its own address space as a performance optimization or to ensure cache coherency in the absence of a hardware-enforced cache coherency protocol. In this case the alternatives for the *invalidate* operation are the same as that of the *clean* operation described above.

When used in isolation the *invalidate* operation renders the values of any memory objects that have been modified since the most recent *clean* operation undefined. Therefore this operation may only be performed by resourced threads. The alternative for architectures that are unable to perform this exact operation is to leave the content of the corresponding caches unchanged.

A-12.1.3 Errors

Successful completion of the requested CacheControl operation sets the *Result*$_{\text{OUT}}$ parameter to TRUE and the ErrorCode register to an undefined value. The CacheControl system call may fail for any of the following reasons:

INVALIDSPACE The *Target*$_{\text{IN}}$ parameter does not specify a valid space ID; non-resourced thread attempting to perform an operation other than *clean* or *clean+invalidate* operation on its own address space; a non-resourced thread attempting to perform an operation on an address space which is not its own.

INVALIDPARAMETER The *Control*$_{\text{IN}}$*.Op* field specifies an invalid operation or one or more of the selected memory regions overlaps with a reserved region of an address space as described in Section A-3.4. Alternatively the *Control*$_{\text{IN}}$*.Count* field is set to a non-zero value for the FLUSHICACHE, FLUSHDCACHE or FLUSHALLCACHES operation, or is set to a number greater than MaxMessageData/2.

In all cases, the CacheControl system call will set the *Result*$_{\text{OUT}}$ parameter to FALSE and the ErrorCode register of the calling thread to the specified value. If more than one of the above conditions hold, the ErrorCode register will be set to the first applicable condition from the above list. The cache content will not be modified.

A-12.2 CapControl

INPUT PARAMETERS

Parameter:	Description:	Type:	Register:
$Clist_{IN}$	Target capability list	**ClistId**	Parameter$_0$
$Control_{IN}$	Control word	**CapControl**	Parameter$_1$

INPUT REGISTERS: MessageData$_0$ TO MessageData$_{(Control_{IN}.Count)-1}$

Parameter:	Description:	Type:	Register:
$Parameter_{i\,IN}$	Argument$_i$	**CapParameter**	MessageData$_i$

OUTPUT PARAMETERS

Parameter:	Description:	Type:	Register:
$Result_{OUT}$	The result flag	**Flag**	Result$_0$
$ErrorCode_{OUT}$	The error code	**ErrorCode**	ErrorCode

The CapControl system call is used to create and delete capability lists as well as to create an IPC capability and delete an individual capability.

Each of these operations are further described in Sections A-12.2.1 to A-12.2.4, below. The **CapControl** data type is further described in Section A-9.7.

A-12.2.1 Creating a Capability List

The CapControl system call can be used to create a new empty capability list by setting the $Control_{IN}$ input parameter to CreateClist. The size of the capability list to be created is specified using $Parameter_{0\,IN}$ of the MessageData registers. The size of the capability list must be specified in terms of the number of entries. The ClistId specfied by the input parameter, $Clist_{IN}$, must be within the range of Clist Ids accessible to the address space of the calling thread. In addition, it should be noted that the address space of the calling thread must have access to a kernel heap.

A-12.2.2 Deleting a Capability List

The CapControl system call can be used to delete an existing empty capability list by setting the $Control_{IN}$ parameter to DeleteClist. The capability list to be deleted is specified using the $Clist_{IN}$ input parameter. Where the specified capability list is not empty, OKL4 will return the ErrorCode, ClistNotEmpty. Note that the ClistId specified by the input parameter, $Clist_{IN}$, must be within the range of Clist Ids accessible to the address space of the calling thread in order for this operation to succeed.

A-12.2.3 Creating an IPC Capability

The CapControl system call can be used to create an IPC capability by setting the $Control_{IN}$ parameter to CreateIpcCap. The capability list in which the newly created IPC capability is to be stored, must be specified using the $Clist_{IN}$ input parameter.

The capability list containing the thread capability from which the new IPC capability is to be generated is specified using the $Parameter_{1\,IN}$ input parameter of the MessageData registers. The source capability identifier, that is the entry number of this thread capability within the capability list specified by $Parameter_{1\,IN}$, must be specified using $Parameter_{0\,IN}$. The capability identifier which specifies the entry number at which the newly created IPC capability is to be stored within the capability list specified by $Clist_{IN}$ is specified using $Parameter_{2\,IN}$. Note that the Clist Ids specified by the input parameters $Parameter_{1\,IN}$ and $Clist_{IN}$ must be within the range of Clist Ids accesible to the source and destination address spaces, respectively.

A-12.2.4 Deleting a Capability

The CapControl system call can be used to delete an existing capability by setting the *Control*$_{\text{IN}}$ parameter to DeleteCap. The capability list containing the capability to be deleted is specified using the *Clist*$_{\text{IN}}$ parameter.

The capability identifier which specifies the entry number of the capability to be deleted within the capability list specified by the *Clist*$_{\text{IN}}$ parameter must be specified using *Parameter*$_{0_{\text{IN}}}$. Note that the ClistId specified by the input parameter, *Clist*$_{\text{IN}}$, must be within the range of Clist Ids accessible to the address space of the calling thread in order for this operation to succeed.

A-12.2.5 Errors

On successful completion the CapControl system call sets the *Result*$_{\text{OUT}}$ parameter to True and the ErrorCode register to an undefined value. The CapControl system call may fail for any of the following reasons:

InvalidClist The capability list specified using the *Clist*$_{\text{IN}}$ input parameter or one of the input registers is not a valid capability list.

OutOfMemory The kernel was unable to complete the operation due to a lack of available system resources.

ClistNotEmpty The caller attempted to delete a clist that was not empty.

InvalidCap The capability specified using an Input Register is not a valid capability.

InvalidParameter The operation specified using the *Control*$_{\text{IN}}$ parameter is not a valid operation.

In all cases, the CapControl system call will set the *Result*$_{\text{OUT}}$ parameter to False and the ErrorCode register of the calling thread to the specified value. If more than one of the above conditions hold, the ErrorCode register will be set to the first applicable condition from the above list.

A-12.3 ExchangeRegisters

INPUT PARAMETERS

Parameter:	Description:	Type:	Register:
Target$_{\text{IN}}$	Target thread	CAPID	Parameter$_0$
Control$_{\text{IN}}$	Control word	REGCONTROL	Parameter$_1$
StackPointer$_{\text{IN}}$	New value of StackPointer	WORD	Parameter$_2$
InstructionPointer$_{\text{IN}}$	New value of InstructionPointer	CODE	Parameter$_3$
Flags$_{\text{IN}}$	New value of Flags	WORD	Parameter$_4$
UserHandle$_{\text{IN}}$	New value of UserHandle	WORD	Parameter$_5$

OUTPUT PARAMETERS

Parameter:	Description:	Type:	Register:
Result$_{\text{OUT}}$	Result flag	FLAG	Result$_0$
Control$_{\text{OUT}}$	Control word	REGCONTROL	Result$_1$
StackPointer$_{\text{OUT}}$	Old value of StackPointer	WORD	Result$_2$
InstructionPointer$_{\text{OUT}}$	Old value of InstructionPointer	CODE	Result$_3$
Flags$_{\text{OUT}}$	Old value of Flags	WORD	Result$_4$
UserHandle$_{\text{OUT}}$	Old value of UserHandle	WORD	Result$_5$
ErrorCode$_{\text{OUT}}$	The error code	ERRORCODE	ErrorCode

OKL4 provides the ExchangeRegisters system call to enable user-level implementation of exception handlers and similar operating system features. The ExchangeRegisters system call supports temporary suspension of thread execution and aborting of IPC operations, allowing operating systems a means of error recovery. Note that the type, CapId is interchangeable with the type, ThreadId for compatibility reasons.

The ExchangeRegisters system call provides the following functionality:

- Obtaining the current value of the StackPointer, InstructionPointer, Flags and UserHandle registers.
- Modifying the values of the above registers.
- Inspecting the user state of a thread.
- Copying the user state between threads.
- Aborting pending IPC operations.
- Temporarily suspending and subsequently resuming thread execution.

Each function is controlled and performed independently of the others and are described in Sections A-12.3.1 to A-12.3.7. With the exception of aborting IPC operations, all system calls executed by the target thread before the ExchangeRegisters system call are completed prior to the operations requested by the ExchangeRegisters system call.

The ExchangeRegisters system call is only permitted if the thread specified by the *Target*$_{\text{IN}}$ parameter is active and meets at least one of the following conditions:

- The target thread is a thread capability.
- The target thread is an IPC capability and both the caller and the target threads reside within the same address space.

A-12.3.1 Obtaining Values of Virtual Registers

If the *D* flag is set in the *Control*$_{\text{IN}}$ parameter, the ExchangeRegisters system call places the the current values of the StackPointer, InstructionPointer, Flags and UserHandle registers of the target thread in the *StackPointer*$_{\text{OUT}}$, *InstructionPointer*$_{\text{OUT}}$, *Flags*$_{\text{OUT}}$ and *UserHandle*$_{\text{OUT}}$ parameters, respectively. If the system call is also used to modify the values of these registers, the values prior to modification are delivered.

A-12.3.2 Setting Values of Virtual Registers

The *SP*, *IP*, *FL* and *UH* flags in the *Control*$_{\text{IN}}$ parameter are used to enable modifications to the StackPointer, InstructionPointer and UserHandle registers of the target thread, respectively. Where one or more flags are set in *Control*$_{\text{IN}}$, the corresponding registers of the target thread are set to the value of the associated input parameter.

A-12.3.3 Inspecting Values of General-Purpose Registers

Parts of the thread state not covered in Section A-12.3.1 can be inspected using the ExchangeRegisters system call by copying the values of all architecture-defined general-purpose hardware registers of the target thread into the caller's MessageData registers. This operation is requested by setting the the *Rget* flag in the *Control*$_{\text{IN}}$ parameter to the ExchangeRegisters system call. The actual sets of general-purpose registers and its mapping onto the OKL4 MessageData registers is architecture-defined and described in Part C of this manual.

A-12.3.4 Changing Values of General-Purpose Registers

Parts of the thread state not covered in Section A-12.3.2 can be modified using the ExchangeRegisters system call by copying the values the caller's MessageData registers into the target thread's architecture-defined general-purpose hardware registers. This operation is requested by setting the the *Rset* flag in the *Control*$_{\text{IN}}$ parameter to the ExchangeRegisters system call. The actual sets of general-purpose registers and its mapping onto the OKL4 MessageData registers is architecture-defined and described in Part C of this manual. The use of the *Rset* flag cannot be combined with the *Rget* flag and will result in an undefined operation and will leave the target thread's registers in an undefined state.

A-12.3.5 Copying Values of General-Purpose Registers

Parts of the thread state not covered in Section A-12.3.2 can be set by copying values of all architecture-defined general-purpose hardware registers from another thread. The actual set of general-purpose registers is architecture-defined and described in Part C of this manual.

This ExchangeRegisters operation is specified by setting the *CPY* flag in the *Control*$_{\text{IN}}$ parameter and specifying the thread number of the source thread in the *Source* field of the *Control*$_{\text{IN}}$ parameter. If the *Source* field refers to an inactive thread, the state of the target thread will be undefined after the system call. This function is useful for implementing POSIX fork semantics. Note that the thread specified by the *Source* field is subject to the same conditions described above, as the thread specified by the ExchangeRegistersTarget parameter.

A-12.3.6 Aborting IPC Operations

The ExchangeRegisters system call can be used to abort an IPC exchange involving the target thread. This operation is requested by setting the *AS* and *AR* flags in the *Control*$_{\text{IN}}$ parameter. When *AS* is set and the target thread is currently executing an IPC *send* operation, the operation will be aborted. Similarly, when *AR* is set and the target thread is currently executing an IPC *receive* operation, the operation will be aborted. The effects of aborting an IPC operation visible to the target thread is further discussed in Section A-12.5.

A-12.3.7 Suspending and Resuming Thread Execution

The ExchangeRegisters system call can be used to temporarily suspend execution of a thread. This is specified by setting the *H* flag in the *Control*$_{\text{IN}}$ parameter. If both the *H* flag and the *HE* flag is set, execution of the target thread is suspended until it is resumed by calling ExchangeRegisters with the *H* flag set and the *HE* flag cleared.

If the ExchangeRegisters system call attempts to suspend an already-suspended thread, or resume execution of a thread that has not been suspended, the status of the thread is not affected.

A-12.3.8 Setting and clearing the LDT on IA-32

The ExchangeRegisters system call can be used to set and clear the LDT on IA-32. This is achieved using the `Set_Tls` bit, bit 7, of the *Control*$_{\text{IN}}$ parameter. It manipulates the TLS in an architecture dependent fashion. On the IA-32, this allows the LDT to be modified. The actual segment descriptors are provided in the message registers. This operation uses three message registers. Message register 3n is used to supply the function word, with message registers 3n + 1 and 3n + 2 supplying the the low word and the high word of the segment descriptor, respectively. As use of the `MRToRegs` bit also uses message registers to supply inputs, the `SetTls` bit cannot be set in conjunction with the `MRToRegs` bit.

A-12.3.9 Outputs

On successful completion of the requested operation, the ExchangeRegisters system call will set the *Result*$_{\text{OUT}}$ parameter to 1 and the ErrorCode register to an undefined value. It will clear all fields of the *Control*$_{\text{OUT}}$ parameter with the exception of the *AS*, *AR* and *HE* flags. Irrespective of the value of the corresponding flag in the *Control*$_{\text{IN}}$ parameter, these flags will be set as follows:

- The *AS* flag is set if the target thread is executing an IPC *send* operation at the time of the ExchangeRegisters system call.
- The *AR* flag is set if the target thread is executing an IPC *receive* operation at the time of the ExchangeRegisters system call.
- The *HE* flag is set if the target thread has been suspended prior to the invocation of the ExchangeRegisters system call.

In addition, if the *REGS* flag is set, all architecture-defined general-purpose hardware registers of the target thread are copied into the caller's MessageData registers.

A-12.3.10 Errors

In the event of an error the ExchangeRegisters system call will set the *Result*$_{\text{OUT}}$ to 0. The ExchangeRegisters system call may fail for any of the following reasons:

INVALIDTHREAD The *Target*$_{\text{IN}}$ input parameter does not specify an existing user thread or does not specify the requisite capability for the requested operation.

If one of the above conditions hold, the ErrorCode register will be set to INVALIDTHREAD. The target thread will not be modified and remaining output parameters will be set to undefined values.

A-12.4 InterruptControl

INPUT PARAMETERS

Parameter:	Description:	Type:	Register:
$Target_{IN}$	Target thread	**THREADID**	Parameter$_0$
$Control_{IN}$	Control word	**INTCONTROL**	Parameter$_1$
	INPUT REGISTERS: MessageData$_0$ TO MessageData$_{Control_{IN}.Count-1}$		
$Parameter_{IN}$	IrqParameter Word$_i$	**IRQPARAMETER**	MessageData$_i$

OUTPUT PARAMETERS

Parameter:	Description:	Type:	Register:
$Result_{OUT}$	The result flag	**FLAG**	Result$_0$
$ErrorCode_{OUT}$	The error code	**ERRORCODE**	ErrorCode

The InterruptControl system call is used to control interrupt registration and acknowledgement in the OKL4 kernel. The InterruptControl system call requires the addressed interrupts to have been granted to the address space of the caller using the *Elfweaver* tool at build time. The $Target_{IN}$ parameter must be a thread that resides in the same address space as the calling thread.

The target thread may register and unregister for an interrupt by setting the *Request* field of the $Control_{IN}$ parameter to *Register* and *Unregister*, respectively. Similarly, the target thread may acknowledge or optionally wait for further interrupts by setting the *Request* field of the $Control_{IN}$ parameter to *Acknowledge* and *AcknowledgeWait*, respectively. The *Request* field may be used to supply other optional data to the BSP as required.

The interrupts to be acknowledged and registered are encoded in interrupt descriptors which are stored in the $Parameter_{i\,IN}$ message registers. The interrupt descriptor encodings for registration and acknowledgement is defined by the BSP. The *Count* field of the $Control_{IN}$ parameter is used specify the highest message register containing a $Parameter_{i\,IN}$ word.

The *NotifyBit* field of the $Control_{IN}$ parameter is used during registration to specify the asynchronous notification bit to use when delivering the requested interrupts. The BSP has discretion in allowing the use of separate notification bits for different interrupt descriptors.

A-12.4.1 Errors

Upon successful completion of the requested operation, the InterruptControl system call sets the $Result_{OUT}$ parameter to TRUE and the ErrorCode register to an undefined value. The InterruptControl system call may fail for any of the following reasons:

INVALIDPARAMETER	The encoding is BSP specific. Typically, this is the result of any of the following: invalid control word; handler already registered for specified interrupt; invalid notify bit requested; incorrect handler specified for unregistering; the address space of the caller does not have access to the specified interrupt(s).
OUTOFMEMORY	The kernel was unable to complete the operation due to a lack of available system resources.
BSP DEFINED	A BSP defined error.

In each case, the InterruptControl system call will set the $Result_{OUT}$ parameter to FALSE and the ErrorCode register of the calling thread to the specified value. If more than one of the above conditions hold, the ErrorCode register will be set to the first applicable condition from the above list. Note that where multiple interrupts are specified, some operations may have succeeded prior to the failure case.

A-12.5 Ipc

INPUT PARAMETERS

Parameter:	Description:	Type:	Register:
Target$_{\text{IN}}$	Target thread	**ThreadId**	Parameter$_0$
Source$_{\text{IN}}$	Source thread	**ThreadId**	Parameter$_1$
Acceptor$_{\text{IN}}$	Message acceptor	**Acceptor**	Acceptor
Tag$_{\text{IN}}$	Message tag	**MessageTag**	MessageData$_0$

INPUT REGISTERS: MessageData$_1$ TO MessageData$_{Tag_{\text{IN}}.Untyped}$

Data$_{i\,\text{IN}}$	Message data to send	**Word**	MessageData$_i$

OUTPUT PARAMETERS

Parameter:	Description:	Type:	Register:
ReplyCap$_{\text{OUT}}$	Reply Capability	**ThreadId**	Result$_0$
SenderSpace$_{\text{OUT}}$	The sender's address space	**SpaceId**	SenderSpace
ErrorCode$_{\text{OUT}}$	The error code	**ErrorCode**	ErrorCode
Tag$_{\text{OUT}}$	Message tag	**MessageTag**	MessageData$_0$

OUTPUT REGISTERS: MessageData$_1$ TO MessageData$_{Tag_{\text{IN}}.Untyped}$

Data$_{i\,\text{OUT}}$	Received message data	**Word**	MessageData$_i$

The Ipc system call provides access to the OKL4 inter-process communication facilities described in Chapter A-6. The Ipc system call may be used to perform IPC *send*, *receive* or *notify* operations. Each of these operations are further discussed in sections A-12.5.1 to A-12.5.3.

The IPC *send* or *notify* operations may be performed by setting the *Target*$_{\text{IN}}$ parameter to the ThreadId of the desired target thread. The *Target*$_{\text{IN}}$ parameter may be set to NilThread to indicate that no *send* or *notify* operation is to be performed. In this case, all output parameters will have undefined values upon completion of the system call, with the exception of the *E* flag in the *Tag*$_{\text{OUT}}$ parameter which is used to report IPC errors as described in Section A-12.5.7 and the *Untyped* field in the *Tag*$_{\text{OUT}}$ parameter, which will be set to 0.

The IPC *receive* operations may be performed by setting the *Source*$_{\text{IN}}$ parameter to a value specified in the table in Section A-12.5.4. The *Source*$_{\text{IN}}$ parameter may be set to NilThread to indicate that no *receive* operation is to be performed.

If both the *Target*$_{\text{IN}}$ and *Source*$_{\text{IN}}$ parameters are set to NilThread, Ipc will behave as a *null system call* and return immediately without affecting the system state. If one or both of these parameters are non set to NilThread, then the corresponding operations are performed. If two operations are to be performed, the *send* or *notify* operation is always performed before the *receive* operation. If the *send* or *notify* operation fails, the system call is immediately aborted and any *receive* operations specified are not performed.

It should also be noted that for the *send* operations, the microkernel is permitted to expose the values of all MessageData registers to the recipient of the message. Therefore, the sender should ensure that MessageData registers do not contain any sensitive information prior to invoking the Ipc system call.

A-12.5.1 Sending IPC Messages

To perform an IPC *send* operation, the Ipc system call should be invoked with the $Target_{IN}$ parameter set to a valid ThreadId. The *S* in the Tag_{IN} parameter defines a *blocking send* when set and a *non-blocking send* when cleared. The *N* flag of the message tag must be cleared.

The actual message delivered to the target thread is supplied in the Tag_{IN} and $Data_{i_{IN}}$ parameters, with the exception of the *S*, *R* and *N* flags which will be modified by the microkernel as described in Sections A-12.5.6 and A-12.5.7. The IPC *send* operation is further discussed in Section A-6.3.

A-12.5.2 Sending Asynchronous Notification Messages

To perform an IPC *notify* operation, the Ipc system call should be invoked with the $Target_{IN}$ parameter set to a valid ThreadId with the *N* flag set in the Tag_{IN} parameter. The values supplied in the *Untyped*, *Label*, *S* and *P* fields will be ignored by the microkernel, and will be implicitly set to 1, 0, False and False respectively prior to processing the message. The required signal flags should be supplied in the $MessageData1_{IN}$ parameter. The IPC *notify* operation is further discussed in Section A-6.4.

A-12.5.3 Receiving IPC Messages

To perform an IPC *receive* operation, the sender should invoke the Ipc system call with the $Source_{IN}$ and $Acceptor_{IN}$ parameters configured to indicate the desired class of sender threads as specified Section A-12.5.4. The *R* in the Tag_{IN} parameter defines a *blocking receive* when set and a *non-blocking receive* when cleared.

Upon successful completion of the operation, the received message will be available in the Tag_{OUT} and $Data_{i_{OUT}}$ parameters. Tag_{OUT} will contain the *Untyped* and *Label* fields supplied by the sender. The *E* and *X* output flags will be set appropriately. See Sections A-12.5.6 and A-12.5.7 below.

The reply capability to the sender of the message will be available in the $ReplyCap_{OUT}$ parameter as described in Section A-12.5.5. The IPC *receive* operation is further discussed in Section A-6.5.

A-12.5.4 Thread Classes

$Source_{IN}$	$Acceptor_{IN}.Notify$	**Thread Class**
WaitNotify	True	Messages will be accepted from any thread that performs a *notify* operation to the thread invoking the Ipc system call. Setting the $Acceptor_{IN}.Notify$ parameter to False with the $Source_{IN}$ parameter set to WaitNotify will render the state of the address space associated with the thread invoking the Ipc system call undefined.
AnyThread	False	Messages will be accepted from any OKL4 thread which performs an IPC *send* operation to the thread invoking the Ipc system call.
AnyThread	True	Messages will be accepted from any thread which performs an IPC *send* or *notify* operation to the thread invoking the Ipc system call.
ThreadId	N/A	Messages will be accepted only from the specified OKL4 user thread. Note that IPC *notify* operations will not be accepted.

OKL4 allows the target thread to specify the class of threads from which a message should be accepted when performing IPC *receive* operations. This is achieved via the use of the InputIpcSource and $Acceptor_{IN}$ parameters as specified in the above table.

A-12.5.5 Obtaining the Message Origin

Upon completion of an IPC *receive* operation, *ReplyCap*$_{\text{OUT}}$ will be set to the thread reply capability of the sender with the exception of a messages sent via an IPC *notify* operation. If the message was sent using the IPC *notify* operation, the *ReplyCap*$_{\text{OUT}}$ specifier will be set to NilThread.

A-12.5.6 Remote Operations

Upon completion of an IPC *receive* operation, the *X* flag will be set if the sender of the message was executing on a different processing unit to the recipient when it invoked the IPC *send* operation. The *X* flag is not set by the IPC *notify* operation. The *X* flag is provided by the microkernel as a hint only which an operating system may use to optimize IPC performance.

A-12.5.7 Errors

Upon the successful completion of all requested IPC operations, the Ipc system call sets the *E* flag in the *Tag*$_{\text{OUT}}$ parameter to False and the ErrorCode register to an undefined value.

In the event of an error, the Ipc system call will set the *E* flag in the *Tag*$_{\text{OUT}}$ parameter to True and the ErrorCode register of the calling thread to IpcError. The *P* flag in the ErrorCode register will be set if the error occurred during a *receive* operation and cleared in all other cases. The *Cause* flag will be set to one of the following values:

NoPartner — A *reply* operation was issued to a target thread that is not waiting to receive, or a *receive* operation was issued to a target thread that is ready to send. The target thread is not notified.

InvalidPartner — An invalid **ThreadId** was specified for the target thread or the sender does not have permission to communicate with the specified recipient due to ipc-control restrictions.

MessageOverflow — *Untyped* field of the message tag specifies a value greater than the MaxMessageData system parameter. The error is delivered to both the sender and the recipient.

IpcRejected — A *notify* operation was issued to a thread that has cleared the *Notify* flag in its Acceptor register. The recipient of the message is not notified.

IpcCancelled — The IPC operation was cancelled by another thread using the ExchangeRegisters system call described in Section A-12.3 prior to commencing data transfer between the threads. The target thread is not notified.

IpcAborted — The IPC operation has been aborted by another thread using the ExchangeRegisters system call described in Section A-12.3 after data transfer has commenced between the threads. Both the sender and receiver of the message are notified.

If more than one of the above conditions hold, the *Cause* field will be set to the first applicable condition from the above list.

A-12.6 MapControl

INPUT PARAMETERS

Parameter:	Description:	Type:	Register:
Target$_{\mathrm{IN}}$	Target address space	**SpaceId**	Parameter$_0$
Control$_{\mathrm{IN}}$	Control word	**MapControl**	Parameter$_1$

INPUT REGISTERS: MessageData$_0$ TO MessageData$_{3(\mathit{Control}_{\mathrm{IN}}.\mathit{Count})-2}$

MapItem$_{i\,\mathrm{IN}}$	Input map data	**MapItem**	MessageData$_{3i}$

OUTPUT PARAMETERS

Parameter:	Description:	Type:	Register:
Result$_{\mathrm{OUT}}$	The result flag	**Flag**	Result$_0$
ErrorCode$_{\mathrm{OUT}}$	The error code	**ErrorCode**	ErrorCode

OUTPUT REGISTERS: MessageData$_0$ TO MessageData$_{2(\mathit{Control}_{\mathrm{IN}}.\mathit{Count})-1}$

QueryItem$_{i\,\mathrm{OUT}}$	Output query data	**QueryItem**	MessageData$_{3i+2}$

The MapControl system call is used to configure a virtual address space. Adddress spaces are described in Chapter A-3. It is used to create and remove mappings of virtual pages to physical pages, as well as being used to obtain information about existing mappings in the address space. It is usually called to resolve address errors which are signalled by a page fault message to a pager thread as described in Sections A-3.2 and A-13.2. Both mapping and querying operations may be performed independently during a single invocation of the MapControl system call and are further described in Sections A-12.6.1 and A-12.6.3.

The target address space is specified by setting the *Target*$_{\mathrm{IN}}$ parameter to its space ID. As a single invocation of MapControl may be used to perform multiple address space operations (n), multiple groups of three MessageData registers may be provided. Each group of the three MessageData registers are encoded as a **MapItem**. These range from 0 to $3i+2$ where $i = n-1$. The number of operations must be specified in the *Control*$_{\mathrm{IN}}$.*Count* field as $n-1$ or i. Where multiple operations are performed by a single invocation of MapControl, they are carried out in sequence with each query operation returning the state of the address space *prior* to the corresponding modify operation. That is, later modifications override those performed earlier and any query operations requested after a previous address space modification will return the state of the modified address space. This allows MapControl to be used to atomically modify a set of mappings and return the previous state of the modified mappings.

A-12.6.1 Creating Address Space Mappings

The MapControl system call may be used to create a mapping in the target address space by setting the *M* flag in the *Control*$_{\mathrm{IN}}$ parameter. The mappings to be inserted are specified using the MessageData registers. The region of the address space selected for modification, is specified using the *VirtualBase* field of the **VirtualDescriptor** and the *PageSize* field of the **SizeDescriptor** belonging the **MapItem**. Any existing mappings in the region are replaced with the new mapping. Note that the *Valid* flag of the **SizeDescriptor** of the **MapItem** must be set to indicate the creation of a mapping.

Currently the *NumPages* field of the **SizeDescriptor** has to be set to 1. In addition the *Replace* and *Sparse* fields must be set to 0.

The **VirtualDescriptor**, **SizeDescriptor** and **MapItem** types are further described in Sections A-9.44, A-9.38 and A-9.24, respectively.

The subsequent mapping consists of three components and is specified using the following fields:

- The access permissions for the mapping are specified using the *Access* field of the **SIZEDESCRIPTOR** corresponding to the relavent *MapItem*$_{i_{\text{IN}}}$ parameter. These permissions are used in combination with the value of the PMask parameter to compute the effective access permissions for the corresponding memory object.
- The caching policy required for mapping is specified using the *Attr* field of the **VIRTUALDESCRIPTOR** belonging to the corresponding *MapItem*$_{i_{\text{IN}}}$ parameter.

It is recommended that operating system personalities map memory objects using the largest possible page size supported by the hardware in preference to a sequence of mappings to smaller consecutive regions as many OKL4 architecture implementations are designed to optimize utilization of system resources for such usage.

A-12.6.2 Removing Address Space Mappings

The MapControl system call may be used to remove a mapping in the target address space by setting the *M* flag in the *Control*$_{\text{IN}}$ parameter and specifying the mappings to be deleted using the MessageData registers. The region of the address space selected for modification, is specified using the *VirtualBase* field of the **VIRTUALDESCRIPTOR** and the *PageSize* field of the **SIZEDESCRIPTOR** belonging the **MAPITEM**. Note that the *Valid* flag of the **SIZEDESCRIPTOR** of the **MAPITEM** must be cleared to indicate the removal of a mapping.

Currently the *NumPages* field of the **SIZEDESCRIPTOR** has to be set to 1. In addition the *Replace* and *Sparse* fields must be set to 0.

The **VIRTUALDESCRIPTOR**, **SIZEDESCRIPTOR** and **MAPITEM** types are further described in Sections A-9.44, A-9.38 and A-9.24, respectively.

A-12.6.3 Querying The Current Address Space Mapping

Querying the address space mapping is deprecated for the OKL4 2.2 release. Note that specifics relating to this operation may have changed from previous releases.

A-12.6.4 Errors

Upon successful completion of the requested operation, the MapControl system call sets the *Result*$_{\text{OUT}}$ parameter to TRUE and the ErrorCode register to an undefined value. The MapControl system call may fail for any of the following reasons:

INVALIDSPACE The *Space*$_{\text{IN}}$ parameter does not specify a valid space Id or the requesting address space does not have access to the specified space Id.

INVALIDPARAMETER The *Control*$_{\text{IN}}$.*Count* parameter specifies a value greater than MaxMessageData/2 or and invalid encoding of *MapItem*$_{i_{\text{IN}}}$ is encountered.

OUTOFMEMORY The kernel was unable to complete the operation due to a lack of available system resources.

In each case, the MapControl system call will set the *Result*$_{\text{OUT}}$ parameter to FALSE and the ErrorCode register of the calling thread to the appropriate value. If more than one of the above conditions hold, the ErrorCode register will be set to the first applicable condition from the above list. The target address space may have been modified if the error occurred when multiple operations where specified and the error did not occur in the first operation.

A-12.7 MemoryCopy

INPUT PARAMETERS

Parameter:	Description:	Type:	Register:
Remote$_{\text{IN}}$	Reply Capability	**CAPID**	Parameter$_0$
Local$_{\text{IN}}$	Local memory address	**WORD**	Parameter$_1$
Size$_{\text{IN}}$	Number of bytes to copy	**WORD**	Parameter$_2$
Direction$_{\text{IN}}$	The direction of the copy	**WORD**	Parameter$_3$

OUTPUT PARAMETERS

Parameter:	Description:	Type:	Register:
Result$_{\text{OUT}}$	The result flag	**FLAG**	Result$_0$
ErrorCode$_{\text{OUT}}$	The error code	**ERRORCODE**	ErrorCode

The MemoryCopy system call may be used to copy memory between two address spaces. The *Remote*$_{\text{IN}}$ parameter is used to specify an IPC reply capability contained in IPC message from the remote thread with which the memory copy is to take place. This IPC message is also used to specify the allowed directions of memory copy, that is whether the local calling thread of MemoryCopy system call is permitted to copy from, copy to or is free to choose the direction of the memory copy. The MemoryCopyFrom option only allows the local thread to copy memory from the buffer of the remote thread, the MemoryCopyTo option allows the local thread to copy memory to the buffer of the remote thread and the MemoryCopyBoth gives the local thread the choice of whether to copy to or from the buffer of the remote thread. The chosen direction is then specified using the *Direction*$_{\text{IN}}$ parameter. If the direction specified using the *Direction*$_{\text{IN}}$ parameter is not permitted by the remote thread, the kernel will raise an appropriate error.

The *Location*$_{\text{IN}}$ and *Size*$_{\text{IN}}$ parameters are used to specify the start of the buffer of the local thread and the number of bytes to be copied, respectively. If the buffer size of the local thread to be copied exeeds the buffer size of the remote thread, the kernel will raise an error.

A-12.7.1 Errors

Upon successful completion of the requested operation, the MemoryCopy system call sets the *Result*$_{\text{OUT}}$ parameter to TRUE and the *ErrorCode*$_{\text{OUT}}$ register to an undefined value.

The MemoryCopy system call may fail for any of the following reasons:

INVALIDPARAMETER One or more of the input parameters contain invalid data. This may mean that the provided IPC capability is invalid, or a direction may not match what the caller of the Ipc expects, or the memory address provided may not fall within native word boundary or that the caller does not have access to the destination memory.

NOMEM The source or the target contains one or more unmapped areas.

In all cases, the MemoryCopy system call will set the *Result*$_{\text{OUT}}$ parameter to FALSE and the *ErrorCode*$_{\text{OUT}}$ register of the calling thread to the appropriate value. If more than one of the above conditions hold, the *ErrorCode*$_{\text{OUT}}$ register will be set to the first applicable condition in the above list.

A-12.8 Mutex

INPUT PARAMETERS

Parameter:	Description:	Type:	Register:
Target$_{\text{IN}}$	Target mutex	**MutexId**	Parameter$_0$
Control$_{\text{IN}}$	Control word	**MutexControl**	Parameter$_1$
VA$_{\text{IN}}$	Virtual address of hybrid mutex	**Address**	Parameter$_2$

OUTPUT PARAMETERS

Parameter:	Description:	Type:	Register:
Result$_{\text{OUT}}$	The result flag	**Flag**	Result$_0$
ErrorCode$_{\text{OUT}}$	The error code	**ErrorCode**	ErrorCode

The Mutex system call is used to allow threads to perform a *lock* (acquire blocking), *trylock* (acquire non-blocking) and *unlock* (release operations) on kernel only or hybrid kernel/user mutex objects. The *VA*$_{\text{IN}}$ field is used only by hybrid mutex operations and points to a memory mapped location containing a word of type **HybridMutex**. It should be noted that all Mutex operations require that the calling address space to have acess to the MutexId specified by the *Target*$_{\text{IN}}$ parameter.

A-12.8.1 Errors

On successful completion, the Mutex system call will set the *Result*$_{\text{OUT}}$ parameter to TRUE and the ErrorCode register of the calling thread to an undefined value. The Mutex system call may fail for any of the following reasons:

INVALIDMUTEX	The specified mutex does not exist or is not in the range *0 to MaxMutexes - 1*.
MUTEXBUSY	The mutex specified to be deleted is currently locked.
INVALIDPARAM	The specified virtual address is not mapped. This error can only occur in hybrid mutex operations.

In each case, the Mutex system call will set the *Result*$_{\text{OUT}}$ parameter to FALSE and the ErrorCode register of the calling thread to the specified value. If more than one of the above conditions hold, the ErrorCode register will be set to the first applicable condition from the above list.

A-12.9 MutexControl

INPUT PARAMETERS

Parameter:	Description:	Type:	Register:
Target$_{\text{IN}}$	Target mutex	**MUTEXID**	Parameter$_0$
Control$_{\text{IN}}$	Control word	**MUTEXCONTROL**	Parameter$_1$

OUTPUT PARAMETERS

Parameter:	Description:	Type:	Register:
Result$_{\text{OUT}}$	The result flag	**FLAG**	Result$_0$
ErrorCode$_{\text{OUT}}$	The error code	**ERRORCODE**	ErrorCode

The MutexControl system call is used to control the creation and deletion of kernel level mutex objects. It should be noted that the calling address space must have access to the MutexId specified by the *Target*$_{\text{IN}}$ parameter and access to a mutex heap in order for operations provided by the MutexControl system call to succeed.

A-12.9.1 Errors

On successful completion, the MutexControl system call will set the *Result*$_{\text{OUT}}$ parameter to TRUE and the ErrorCode register of the calling thread to an undefined value. The MutexControl system call may fail for any of the following reasons:

INVALIDMUTEX — The specified mutex does not exist, is not in the range *0 to MaxMutexes - 1*, or is not accessible to the calling address space. This error can only occur on mutex deletion.

MUTEXBUSY — The mutex specified to be deleted is currently locked.

OUTOFMEMORY — The kernel was unable to complete the operation due to a lack of available system resources. This error can only occur on mutex creation.

In each case, the MutexControl system call will set the *Result*$_{\text{OUT}}$ parameter to FALSE and the ErrorCode register of the calling thread to the specified value. If more than one of the above conditions hold, the ErrorCode register will be set to the first applicable condition from the above list.

A-12.10 PlatformControl

INPUT PARAMETERS

Parameter:	Description:	Type:	Register:
Control$_{\text{IN}}$	Control word	PLATCONTROL	Parameter$_0$
Param1$_{\text{IN}}$	General purpose word	WORD	Parameter$_1$
Param2$_{\text{IN}}$	General purpose word	WORD	Parameter$_2$
Param3$_{\text{IN}}$	General purpose word	WORD	Parameter$_3$

INPUT REGISTERS: MessageData$_0$ TO MessageData$_{(\mathit{Control}_{\text{IN}}.\mathit{Count})}$

Argument$_{i\,\text{IN}}$	General purpose input argument$_i$	WORD	MessageData$_i$

OUTPUT PARAMETERS

Parameter:	Description:	Type:	Register:
Result$_{\text{OUT}}$	The result word	WORD	Result$_0$
ErrorCode$_{\text{OUT}}$	The error code	ERRORCODE	ErrorCode

OUTPUT REGISTERS: MessageData$_0$ TO MessageData$_{(\mathit{Control}_{\text{IN}}.\mathit{Count})}$

Argument$_{i\,\text{OUT}}$	General purpose output argument$_i$	WORD	MessageData$_i$

The PlatformControl system call is available for platform specific functionality to be added to the microkernel. This system call is intended to provide access only to special features provided by the platform that are not possible to implement from user thread. For example, PlatformControl may be used to provide mechanisms for entering low-power sleep modes.

PlatformControl provides for up to four input parameters and up to MaxMessageData of extra arguments in the Message Registers. There are also up to MaxMessageData output arguments in message registers in addition to a result word and error code. *Result*$_{\text{OUT}}$ returns 0 on error and non-zero for success.

PlatformControl only defines the format of the *Control*$_{\text{IN}}$ parameter, which contains the number of input parameters in the *Count* field and a platform specific *Request* field.

Access to the PlatformControl system call can be controlled on a per address space granularity using the *Elfweaver* tool at build time.

A-12.10.1 Errors

Upon successful completion of the requested operation, the PlatformControl system call sets *Result*$_{\text{OUT}}$ to TRUE and the ErrorCode register to an undefined value. The PlatformControl system call may fail for any of the following reasons:

INVALIDSPACE The address space of the target thread does not have access to the PlatformControl system call.

OUTOFMEMORY The kernel was unable to complete the operation due to a lack of available system resources.

IMPLEMENTATION DEFINED The platform may define additional error return values if required.

In each case, the PlatformControl system call will set the *Result*$_{\text{OUT}}$ parameter to FALSE and the ErrorCode register of the calling thread to the specified value. If more than one of the above conditions hold, the ErrorCode register will be set to the to the first applicable condition from the above list. The target thread is not modified.

A-12.11 Schedule

INPUT PARAMETERS

Parameter:	**Description:**	**Type:**	**Register:**
Target$_{\text{IN}}$	Target thread	**CapId**	Parameter$_0$
Slice$_{\text{IN}}$	Time slice length in milliseconds	**Word**	Parameter$_1$
HwThreadMask$_{\text{IN}}$	Set of valid hardware threads	**HwThreadMask**	Parameter$_2$
Domain$_{\text{IN}}$	Reserved parameter	**Word**$_{16}$	Parameter$_3$
Priority$_{\text{IN}}$	The priority	**Word**$_8$	Parameter$_4$
Reserved$_{\text{IN}}$	Reserved parameter	**Word**	Parameter$_5$

OUTPUT PARAMETERS

Parameter:	**Description:**	**Type:**	**Register:**
Result$_{\text{OUT}}$	Current state of the target thread	**ThreadState**	Result$_0$
Slice$_{\text{OUT}}$	Remaining time slice	**Word**	Result$_1$
ErrorCode$_{\text{OUT}}$	The error code	**ErrorCode**	ErrorCode

The Schedule system call is used to configure the thread scheduling parameters used by the OKL4 pre-emptive scheduler described in Chapter A-5.

The *Target*$_{\text{IN}}$ parameter is used to specify the thread to be configured. It shoud be noted that the type, CapId is interchangeable with the type, ThreadId for compatibility reasons. A single invocation of Schedule may be used to modify multiple scheduling parameters of the target thread. Therefore, the scheduling parameters that the user wishes to remain unmodified must be indicated by setting the corresponding input parameters to MaxWord. Note that the *Domain*$_{\text{IN}}$ and the *Reserved*$_{\text{IN}}$ parameters are not currently used and should be set to MaxWord and 0, respectively.

The *Slice*$_{\text{OUT}}$ parameter is set to the remaining time slice length as described in Section A-12.11.4. All time quantities supplied and returned to the Schedule system call are represented in microseconds.

The Schedule system call is only permitted if the thread specified by the *Target*$_{\text{IN}}$ parameter is active and meets at least one of the following conditions:

- The target thread is a thread capability.
- The target thread is an IPC capability and both the caller and the target threads reside within the same address space.

A-12.11.1 Setting The Time Slice Length

The Schedule system call may be used to configure the time slice length associated with each user thread in the system. A thread's time slice is initially set to a default value. This value may be subsequently modified by setting the *Slice*$_{\text{IN}}$ parameter of the Schedule system call to a positive integer. Setting *Slice*$_{\text{IN}}$ to 0 results in the time slice of the thread being set to an infinite value, effectively disabling the implicit pre-emption of the target thread by the microkernel. Note that the smallest time slice length for the target thread can be selected by setting *Slice*$_{\text{IN}}$ to 1.

In the event that time slice is modified by the Schedule system call, the remaining time slice is also reset by the microkernel as described in Section A-12.11.4. If the target thread is executing on different processing unit at the time a Schedule system call sets the length of the time slice, the target thread will execute for the length of time specified by the new time slice prior to its next pre-emption. The impact of the time slice parameter on the OKL4 scheduling algorithm is further discussed in Chapter A-5.

A-12.11.2 Setting The Priority Level

The priority level of a new thread is initially set to `0`. This value may be subsequently modified using the *Priority*$_{\text{IN}}$ parameter of the Schedule system call. The specified priority level must be less than or equal to the maximum priority of the address space of the calling thread. The impact of thread priorities on the OKL4 scheduling algorithm is further discussed in Chapter A-5.

A-12.11.3 Setting The Allowed Processing Units

The Schedule system call may be used to specify the *allowed execution units* parameter which describes the set of execution units on which the thread may be allocated time. A thread is by default free to execute on any execution unit belonging to the default domain. Specifying the allowed execution units parameter is achieved by specifying a hardware thread mask using the *HwThreadMask*$_{\text{IN}}$ parameter. This restricts the thread to executing on a set of specified processing units. Specifying a *HwThreadMask*$_{\text{IN}}$ with no units enabled will render the thread unable to execute. Scheduling is further discussed in Chapter A-5.

A-12.11.4 The Remaining Time Slice

Each user thread is associated with a *remaining time slice*. The remaining time slice is initialized on thread creation to a default value. It is reset to the value of the time slice on each invocation of the Schedule system call that modifies the time slice parameter of a thread. Unless it is initialized to an infinite value, the microkernel decrements the remaining time slice by one microsecond for each microsecond of processing time consumed by the thread. When the remaining time slice reaches 0, the thread is pre-empted as described in Chapter A-5 and the remaining time slice is reset back to its initial value as configured using the Schedule system call. The current value of the remaining time slice is returned in the *Slice*$_{\text{OUT}}$ parameter on every invocation of the Schedule system call. Note that the remaining time slice is never 0; the Schedule system call returns 0 in the *Slice*$_{\text{OUT}}$ for target threads with an infinite value of the time slice parameter.

A-12.11.5 Errors

Upon successful completion of the requested operation, the Schedule system call sets the *Result*$_{\text{OUT}}$ parameter to the current state of the target thread and the ErrorCode register to an undefined value. The Schedule system call may fail for any of the following reasons:

INVALIDTHREAD	The *Target*$_{\text{IN}}$ parameter does not specify an existing user thread.
INVALIDPARAMETER	The *Priority*$_{\text{IN}}$ parameter has a value higher than the priority of the calling thread and is not set to MAXWORD, or the *Domain*$_{\text{IN}}$ parameter is not set to MAXWORD.

In each case, the Schedule system call will set the *Result*$_{\text{OUT}}$ parameter to UNKNOWNSTATE and the ErrorCode register of the calling thread to the specified value. If more than one of the above conditions hold, the ErrorCode register will be set to the first applicable condition from the above list. The target thread will not be modified.

A-12.12 SpaceControl

INPUT PARAMETERS

Parameter:	Description:	Type:	Register:
Target$_{\text{IN}}$	Target address space	**SPACEID**	Parameter$_0$
Control$_{\text{IN}}$	Control word	**SPACECONTROL**	Parameter$_1$
Clist$_{\text{IN}}$	Capability list	**CLISTID**	Parameter$_2$
UtcbRegion$_{\text{IN}}$	Location of the UTCB region	**FPAGE**	Parameter$_3$
SpaceResources$_{\text{IN}}$	New space resources	**SPACERESOURCES**	Parameter$_4$

OUTPUT PARAMETERS

Parameter:	Description:	Type:	Register:
Result$_{\text{OUT}}$	The result flag	**FLAG**	Result$_0$
SpaceResources$_{\text{OUT}}$	Old value of the space resources	**SPACERESOURCES**	Result$_1$
ErrorCode$_{\text{OUT}}$	The error code	**ERRORCODE**	ErrorCode

The SpaceControl system call may be used to create and delete address spaces. On creation, it also configures the layout of a new address space, that is specify the size and location of the UTCB region as described in Section A-12.12.4 as well as determining whether or not the newly created address space is a *resourced* address space. Where the newly created address space is a resourced address space, it has unlimited access to create and delete objects from the resources accessible to the calling address space. Note that resources cannot be given to existing address spaces as it can only be determined on creation.

On certain architectures, the SpaceControl system call may also be used to modify certain address space parameters using the *Control*$_{\text{IN}}$ parameter as described in Part C of this manual. Programs intended to be portable across multiple architectures should set the *Control*$_{\text{IN}}$ to DEFAULTSPACECONTROL.

The target address space is selected by setting *Target*$_{\text{IN}}$ parameter to its space ID. It should be noted that the UTCB region cannot be reconfigured after the address space has been created.

A-12.12.1 Address Space Creation

A new address space may be created by calling SpaceControl with *Target*$_{\text{IN}}$ set to a valid space ID and the *new* flag set in the *Control*$_{\text{IN}}$ parameter. Where the *resourced* flag is also set in the *Control*$_{\text{IN}}$ parameter, a new resourced address space is created. The newly created resourced address space has unlimited access to create and delete objects from the resources accessible to the calling address space. Note that an address space cannot be given resources after creation.

The *Clist*$_{\text{IN}}$ parameter must specify an existing capability list and the *UtcbRegion*$_{\text{IN}}$ parameter must be set to a valid value. Where the newly created address space is granted access to certain resouces, the *resources* flag must be set in the *Control*$_{\text{IN}}$ parameter, and resources specified using the *SpaceResources*$_{\text{IN}}$ parameter.

The space ID provided using the *Target*$_{\text{IN}}$ parameter is chosen by the user and must be a value between 0 and MAXSPACES − 1. Where the specified space ID already exists or is outside the range of valid space IDs, an INVALIDSPACE error is reported and no new address space is created.

Note that setting both the *new* and *delete* flags in the *Control*$_{\text{IN}}$ parameter will result in undefined behaviour.

A-12.12.2 Address Space Deletion

An address space may be deleted by calling SpaceControl with *Target*$_{\text{IN}}$ set to the space ID of the target address space and the *delete* flag set in the *Control*$_{\text{IN}}$ parameter. An address space can only be deleted once it is no longer associated with any threads. Attempting to delete an address space that is associated with at least one thread will result in a SPACENOTEMPTY error and the address space will remain unmodified.

A-12.12.3 Modifying Address Space Resources

Address space resources may be modified using the SpaceControl system call by setting only the *resources* flag in the *Control*$_{\text{IN}}$ parameter and providing a valid value in the *SpaceResources*$_{\text{IN}}$ parameter. Setting the *Target*$_{\text{IN}}$ to an invalid space ID will result in an INVALIDSPACE error.

A-12.12.4 Configuring the UTCB Region

Under OKL4 architectures that set UtcbRegionWidth to a non-zero value, the SpaceControl system call may be used to configure the size and location of the UTCB region within the target address space. The UTCB region is specified as an **FPAGE** object in the *UtcbRegion*$_{\text{IN}}$ parameter and must describe a region of the address space that is entirely within the user region. The *Width* field in the *UtcbRegion*$_{\text{IN}}$ parameter must be greater than or equal to the UtcbRegionWidth. On architecture implementations which set the UtcbRegionWidth parameter to zero, the location of the UTCB region cannot be modified by the user and *UtcbRegion*$_{\text{IN}}$ parameter must be set to NILPAGE. The UTCB region is further described in Section A-3.5.

A-12.12.5 Errors

Upon successful completion of the requested operation, the SpaceControl system call sets the *Result*$_{\text{OUT}}$ parameter to TRUE and the ErrorCode register to an undefined value. The current value of the address space control parameter is returned by SpaceControl in the *Control*$_{\text{OUT}}$ parameter. The SpaceControl system call may fail for any of the following reasons:

INVALIDSPACE — The *Space*$_{\text{IN}}$ parameter does not specify a valid space ID.

INVALIDUTCB — If UtcbRegionWidth is non-zero and the region of memory specified by *UtcbRegion*$_{\text{IN}}$ is too small or overlaps a non-user region of the address space. Otherwise if UtcbRegionWidth is zero, and *UtcbRegion*$_{\text{IN}}$ is not equal to NILPAGE.

SPACENOTEMPTY — The address space that has been specified for deletion has associated threads. An address space can only be deleted once it no longer has any associated threads.

INVALIDPARAM — There was an attempt to give an existing address space resources. Resources can only be given to a new address space at the time of its creation.

In each case, the SpaceControl system call will set the *Result*$_{\text{OUT}}$ parameter to FALSE and the ErrorCode register of the calling thread to the specified value. If more than one of the above conditions hold, the ErrorCode register will be set to the first applicable condition from the above list. The target address space will not be modified.

A-12.13 SpaceSwitch

INPUT PARAMETERS

Parameter:	Description:	Type:	Register:
Target$_{\text{IN}}$	Target thread	THREADID	Parameter$_0$
Space$_{\text{IN}}$	Destination address space of the thread	SPACEID	Parameter$_1$
Utcb$_{\text{IN}}$	Utcb address in destination address space	WORD	Parameter$_2$

OUTPUT PARAMETERS

Parameter:	Description:	Type:	Register:
Result$_{\text{OUT}}$	The result flag	FLAG	Result$_0$
ErrorCode$_{\text{OUT}}$	The error code	ERRORCODE	ErrorCode

The SpaceSwitch system call is used to move a thread residing in a particular address space to the *Space*$_{\text{IN}}$ address space. All other parameters of the thread, with the exception of its UTCB and its associated address space, remain unchanged. The scheduling context of the thread is also not affected by this system call. The thread count of the *Space*$_{\text{IN}}$ address space is incremented and the thread count its old address space is decremented as a result of the relocation of the thread.

All address spaces involved in the SpaceSwitch system call must be associated with a kernel resource object. That is, the address space of the thread that makes the system call, the address space that the target thread is switched from and switched to should all be associated with a kernel resource object.

Once the thread has been relocated it receives its new UTCB address as specified in the *Utcb*$_{\text{IN}}$ parameter unless the OKL4 architecture provides kernel allocated UTCB addresses, in which case the microkernel will allocate a new UTCB address for the thread. The contents of the old UTCB of the thread is copied to the new UTCB with the exception of message registers and reserved regions of the UTCB. Where a thread is relocated, it continues executing in the new address space immediately using the remainder of its current time slice. The register context of the thread, including the *IP* and *SP* registers, is unaffected.

The SpaceSwitch system call can only be used to specify the UTCB address on architectures that set the UtcbRegionWidth parameter to a non-zero value.

It should be noted that using the SpaceSwitch system call to have more than one thread sharing the same UTCB address, will result in undefined operations.

A-12.13.1 Errors

Upon successful completion of the requested operation, the SpaceSwitch system call sets *Result*$_{\text{OUT}}$ to TRUE and the ErrorCode register to an undefined value. The SpaceSwitch system call may fail for any of the following reasons:

INVALIDTHREAD — The target thread is a non-existing thread.

INVALIDSPACE — The *Space*$_{\text{IN}}$ parameter does not specify a valid space identifier and/or does not have a kernel heap.

INVALIDUTCB — The *Utcb*$_{\text{IN}}$ parameter specifies an address of a memory object residing outside of the UTCB area of the thread's address space.

OUTOFMEMORY — The kernel was unable to complete the operation due to a lack of available system resources.

In each case, the SpaceSwitch system call will set the *Result*$_{\text{OUT}}$ parameter to FALSE and the ErrorCode register of the calling thread to the specified value. If more than one of the above conditions hold, the ErrorCode register will be set to the to the first applicable condition from the above list. The target thread is not modified.

A-12.14 ThreadControl

INPUT PARAMETERS

Parameter:	Description:	Type:	Register:
Target$_{\text{IN}}$	Target thread	ThreadId	Parameter$_0$
Space$_{\text{IN}}$	New address space of the thread	SpaceId	Parameter$_1$
Dummy$_{\text{IN}}$	Argument ignored	ThreadId	Parameter$_2$
Pager$_{\text{IN}}$	New pager for the thread	ThreadId	Parameter$_3$
ExceptionHandler$_{\text{IN}}$	New exception handler for the thread	ThreadId	Parameter$_4$
ThreadResources$_{\text{IN}}$	Thread resources specifier	ThrdResources	Parameter$_5$
Utcb$_{\text{IN}}$	UTCB entry of the new thread	Word	Parameter$_6$

OUTPUT PARAMETERS

Parameter:	Description:	Type:	Register:
Result$_{\text{OUT}}$	The result flag	Flag	Result$_0$
ErrorCode$_{\text{OUT}}$	The error code	ErrorCode	ErrorCode
ThreadHandle$_{\text{OUT}}$	New thread's handle	ThreadId	MessageData$_0$

The ThreadControl system call is used to create and delete threads, as well as to modify the Pager, ExceptionHandler and Utcb registers of any user thread in the system. Note that these operations are restricted to threads that have the appropriate resources and or capabilities. The requirements to perform each operation is described below. Capabilities are further described in Chapter A-4.

The parameters *Pager*$_{\text{IN}}$, *ExceptionHandler*$_{\text{IN}}$ and *ThreadResources*$_{\text{IN}}$ are ignored by OKL4 when set to NilThread, they are said to be unspecified, leaving the associated thread parameters unmodified. If the values of these parameters are not specified on thread creation, they will be set to NilThread. Where the ThreadControl system call is used for thread modification, the *Space*$_{\text{IN}}$ and *Utcb*$_{\text{IN}}$ parameters must be set to NilSpace and MaxWord, respectively.

On certain architectures, the ThreadControl system call may also be used to modify certain thread specific resources using the *ThreadResources*$_{\text{IN}}$ parameter as described in Part C of this manual. Programs intended to be portable across multiple architectures should set the *ThreadResources*$_{\text{IN}}$ to DefaultThreadResources. Any combination of these operations may be performed during a single invocation of the ThreadControl system call. These operations are described below.

A-12.14.1 Creating Threads

A thread is created by calling ThreadControl with *Target*$_{\text{IN}}$ set to the thread ID of a new thread and *Space*$_{\text{IN}}$ set to space ID of a valid address space. Space IDs are further described in Section A-3.1. The *Utcb*$_{\text{IN}}$ parameter must be set to the location of the thread's Utcb. The priority of the newly created thread is set to 0, the lowest priority in the system. This priority may be subsequently modified using the Schedule system call.

Upon initialization of a new thread, the thread waits for an start-up message according to the thread start protocol. The microkernel issues a blocking IPC *receive* operation on behalf of the target thread to obtain the values of its InstructionPointer and StackPointer registers as described in Section A-13.3.

On successful creation of a thread, the new thread's handle is returned in the *ThreadHandle*$_{\text{OUT}}$ parameter. The root server can use this to securely identify threads. Note that the calling thread must have access to a kernel heap to create a thread.

A-12.14.2 Modifying Threads

An existing thread may be modified by calling ThreadControl with the desired thread as the *Target*$_{\text{IN}}$ parameter. The *Space*$_{\text{IN}}$ parameter must be set to NilSpace and the *Utcb*$_{\text{IN}}$ parameter must be set to a valid UTCB address. The UTCB, Pager, ExceptionHandler and thread resources may be modified. Note that the calling thread must have access to the appropriate thread capability to modify a thread.

A-12.14.3 Deleting Threads

An existing thread may be deleted by calling ThreadControl with the desired thread as the *Target*$_{\text{IN}}$ parameter. The *Space*$_{\text{IN}}$ parameter must be set to NilSpace and the *Utcb*$_{\text{IN}}$ parameter must be set to 0. Upon deletion, all registers and resources of the thread are discarded. Note that the calling thread must have access to the appropriate thread capability to delete a thread.

A-12.14.4 Setting The Thread's Pager

Each thread is associated with a *pager thread* responsible for constructing and maintaining the address space associated with that thread. The pager is responsible for servicing page fault messages issued by the kernel on behalf of a thread as described in Section A-13.2. The Pager register of the thread is described in Section A-11.11.

A single address space may be shared by multiple threads associated with distinct pagers. The pagers for these threads must co-operate to maintain coherency of the address space function. The current pager of an existing user thread may be obtained using the ExchangeRegisters system call as described in Section A-12.3.

The pager thread of a user thread may be modified by calling ThreadControl with the *Pager*$_{\text{IN}}$ set to an appropriate value. This is may be used to atomically initialize the pager during creation of a new thread. Setting *Pager*$_{\text{IN}}$ to NilThread leaves the Pager register of the thread unmodified. Once set, the value of the pager cannot be changed to NilThread. A thread without a valid pager thread will continue to page fault and will not make any progress, until such time its pager thread is configured to a valid thread. It should be noted that this may block lower priority threads from being scheduled on a single processor machine. Note that the calling thread must have access to the appropriate thread capability to modify the pager of a thread.

A-12.14.5 Setting The Thread's Exception Handler

Each existing user thread is associated with an *exception handler thread* responsible for servicing exception messages issued by the microkernel on behalf of the target thread as described in Section A-13.1. The ExceptionHandler register of the thread is described in Section A-11.4.

The exception handler thread may be set by invoking ThreadControl with the *ExceptionHandler*$_{\text{IN}}$ parameter set to the thread identifier of the desired exception handler thread. Setting the *ExceptionHandler*$_{\text{IN}}$ parameter to NilThread will leave the exception handler unmodified. Once set, the value of the exception handler cannot be changed to NilThread. A thread without a valid exception handler thread will continue to generate exceptions and will not make any progress, until such time its exception handler thread is configured to a valid thread. It should be noted that this may block lower priority threads from being scheduled on a single processor machine. Thread scheduling is further described in Chapter A-5. Note that the calling thread must h ave access to the appropriate thread capability to modify the exception handler of a thread.

A-12.14.6 Setting The Thread's Resources

Each user thread is associated with a default set of resources which is architecture defined. This typically includes the threads general purpose registers and any flags registers. An architecture may support extra thread resources such as multi-media and floating point registers. Note that an architecture may also provide thread resources that are not of the register type.

The microkernel allows the system to restrict a thread to using all, some or none of the non-default registers. The *ThreadResources*$_{\text{IN}}$ parameter is provided to control access to these resources. The actual set of supported resources is architecture-defined and described in Part C of this manual.

Specifying thread resources may result in an error due to insufficient kernel resources.

Programs intended to be portable across multiple architectures should set the *ThreadResources*$_{\text{IN}}$ to DEFAULTTHREADRESOURCES. Note that the calling thread must have access to the appropriate thread capability to modify the resources a thread.

A-12.14.7 Setting The Thread's Utcb Register

Each user thread is associated with a *user thread control block* (UTCB) available to the thread as its Utcb register. The ThreadControl system call can only be used to specify the Utcb register on architectures that set the UtcbRegionWidth parameter to a non-zero value, where the association each thread with its UTCB must be maintained explicitly by the operating system personality. On such systems, the *Utcb*$_{\text{IN}}$ parameter is set to the the location of the UTCB object during thread creation. Setting the *Utcb*$_{\text{IN}}$ to MAXWORD and the *Pager*$_{\text{IN}}$ parameter to NILTHREAD leaves the Utcb register of the target thread unmodified.

Once a thread is created, its Utcb register may not be changed and must be specified as MAXWORD on subsequent invocations of ApiThreadControl.

Each UTCB object must fit within the UTCB region of the address space. Setting the *Utcb*$_{\text{IN}}$ parameter to a UTCB object that resides partially or entirely outside the UTCB area results in an error. Creating more than one thread in the same address space with the same *Utcb*$_{\text{IN}}$ value will result in undefined operation of the threads concerned. The UTCB region is further discussed in Section A-3.5. Note that the calling thread must have access to the appropriate thread capability to modify the Utcb area of a thread.

A-12.14.8 Errors

Upon successful completion of the requested operation, the ThreadControl system call sets *Result*$_{\text{OUT}}$ to TRUE and the ErrorCode register to an undefined value. The ThreadControl system call may fail for any of the following reasons:

INVALIDTHREAD This error may be the result of three possible causes. Firstly, the target thread identifier specified for the creation of a new thread is not valid. Secondly, a non-existing thread has been specified for a modify or delete operation. Lastly, the calling thread has specified itself for a delete operation.

INVALIDSPACE The *Space*$_{\text{IN}}$ parameter does not specify a valid space ID and is not set to NILSPACE.

INVALIDUTCB The *Utcb*$_{\text{IN}}$ parameter specifies an address of a memory object residing outside of the UTCB area of the thread's address space, or specifies a location of another active thread within the same address space. This error is also reported whenever ThreadControl attempts to modify the UTCB register of a thread that has already been created.

INVALIDPARAMETER The specified pager or exception handler is an invalid thread capability.

OUTOFMEMORY The kernel was unable to complete the operation due to a lack of available system resources.

In each case, the ThreadControl system call will set the *Result*$_{\text{OUT}}$ parameter to FALSE and the ErrorCode register of the calling thread to the specified value. If more than one of the above conditions hold, the ErrorCode register will be set to the to the first applicable condition from the above list. The target thread is not modified.

A-12.15 ThreadSwitch

INPUT PARAMETERS

Parameter:	Description:	Type:	Register:
Target$_{\text{IN}}$	Target thread or NILTHREAD	**THREADID**	Parameter$_0$

The ThreadSwitch system call can be used to donate the remainder of the current time slice of a thread to the thread specified by the *Target*$_{\text{IN}}$ parameter. The target thread receives the remainder of the caller's time slice in addition to its ordinary time allocation described in Chapter A-5. If the *Target*$_{\text{IN}}$ parameter is set to NILTHREAD, the remainder of the time slice is forfeited, causing a thread pre-emption event as described in Chapter A-5. The same effect is observed if the target thread is busy, inactive or suspended.

A-13 Communication Protocols

The majority of communication protocols involve an exchange of IPC messages between a user thread and the microkernel acting on behalf of another user.

When the OKL4 microkernel issues IPC *send* and *receive* operations on behalf of a thread, which appear to originate from the corresponding user. When an IPC message is received by the microkernel on behalf of a thread, the thread is typically blocked or inactive during the delivery the message. The microkernel then alters the values of the thread's registers as specified in the received message prior to resuming thread execution.

The format of IPC messages that are exchanged during a microkernel-defined communication protocol is presented using *message diagrams*. This diagram describes the content of the MessageData registers used in the exchange. MessageData$_0$ always contains the **MessageTag** object. Bit offsets from the data structure diagrams have been replaced with the names of the data fields. When presenting values of the **MessageTag** flags, 1 and 0 are used to represent the True and False constructors respectively. ~ is used to represent message components with undefined values ignored by the microkernel as described in Chapter B-1. The following message diagram describes a message consisting of two words:

<table>
<tr><td colspan="7">Second word of message data</td><td>MessageData$_2$</td></tr>
<tr><td colspan="7">First word of message data</td><td>MessageData$_1$</td></tr>
<tr><td>7
Label</td><td>0
S</td><td>0
R</td><td>0
N</td><td>0
M</td><td>~</td><td>2
Count</td><td>MessageData$_0$</td></tr>
</table>

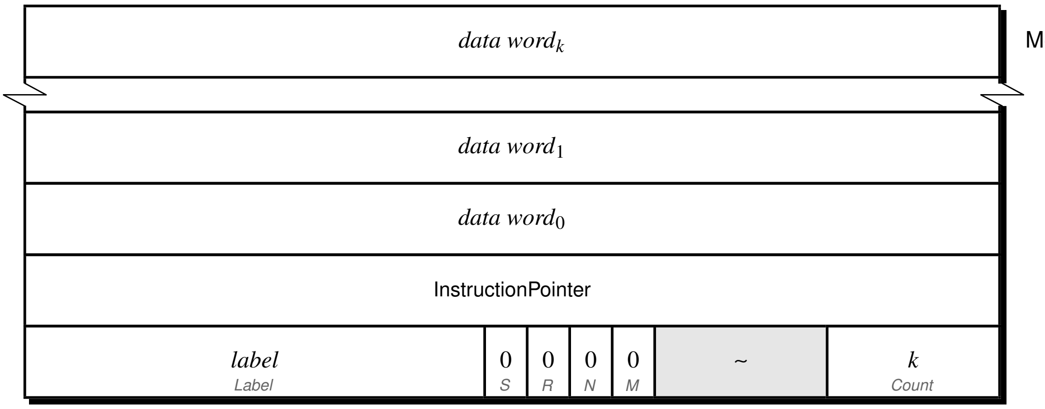
All **MessageTag** flags in the above message are cleared and the *Label* field is set to the value “7”. The same message format is used across all IPC based protocols though the width of message labels may be subject to variation.

Note, for kernel generated messages, OKL4 preserves the values of a thread's MessageData registers used in the exchange and returns them to their original values on completion of the IPC operation.

A-13.1 Exception Protocol

OKL4 uses IPC to deliver hardware exceptions to the *exception handler thread* associated with each user thread in the system. The current exception handler thread is located in the ExceptionHandler register of the thread. When the microkernel detects an exceptional condition within a thread, the thread is blocked and an *exception message* is delivered to its exception handler.

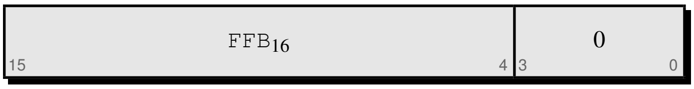
The exception message consists of a label identifying the architecture-defined exception type, the current value of the InstructionPointer register and zero or more architecture-defined data words as described in the Part C of this manual. The exception message has the following format:



data word$_k$	MessageData$_{k+1}$
data word$_1$	MessageData$_3$
data word$_0$	MessageData$_2$
InstructionPointer	MessageData$_1$

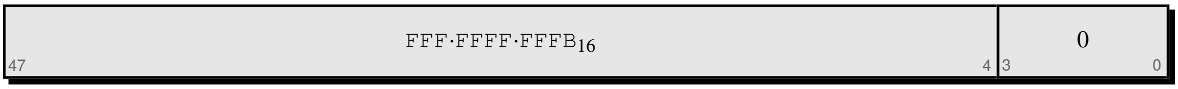
Label	S	R	N	M		Count	
label	0	0	0	0	~	*k*	MessageData$_0$

The number of data words in an exception message is represented by the constant *k* and is dependant on the architecture and exception type. Though delivered by the microkernel, this message will appear to originate from the thread causing the exception. Its *Label* field is set to −80. On 32-bit architectures, the *label* has the following format:



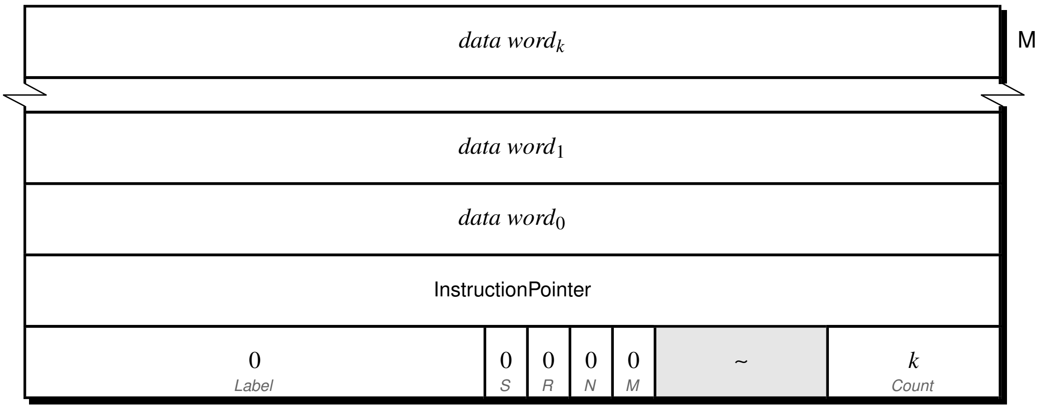
15–4	3–0
FFB$_{16}$	0

On 64-bit architectures, the *label* has the following format:



47–4	3–0
FFF·FFFF·FFFB$_{16}$	0

The field comprising of the four least-significant bits of the label is reserved by the microkernel for future use. Under the current OKL4 API, the microkernel always sets the four least significant bits of the label to 0 and stores the two's complement representation of −5 in the remaining bits of the label. The faulting thread remains suspended until an *exception acknowledgement message* is received from the exception handler of the thread. The *exception acknowledgement message* has the following format:



data word$_k$	MessageData$_{k+1}$
data word$_1$	MessageData$_3$
data word$_0$	MessageData$_2$
InstructionPointer	MessageData$_1$

Label	S	R	N	M		Count	
0	0	0	0	0	~	*k*	MessageData$_0$

The expected number of data words, the semantics of the individual data words and the behaviour of the microkernel when an exception acknowledgement is received with an incorrect number of data words, are all architecture-defined and discussed in Part C of this manual.

The exception handler should deliver the message using a normal user-level IPC operation to the exception-causing thread. Once the *exception acknowledgement message* is received by the microkernel, the InstructionPointer register of the thread is set to the value supplied in the message, other thread registers may also be modified, and the thread execution resumed.

A-13.2 Page Fault Protocol

An attempt by a thread to access an area of its address space that has been configured without the appropriate access permissions, generates a *page fault* handled by the microkernel. The exchange of *page fault messages* with the current pager of the thread forms the *page fault protocol*. When a user thread attempts to access, modify or execute the value or instruction of a memory object mapped without effective *read*, *write*, or *execute* permissions, the offending thread is blocked by the microkernel. A *page fault message* is then issued to the pager of the offending thread, identified using its Pager register.

The page fault message consists of a label identifying the operation type that caused page fault, the location of the memory object and the current value of the InstructionPointer register of the thread. The page fault message is delivered to the page fault handler and has the following format:

<table>
<tr><td colspan="7">InstructionPointer</td><td>MessageData$_2$</td></tr>
<tr><td colspan="7"><i>address</i></td><td>MessageData$_1$</td></tr>
<tr><td><i>label</i>
Label</td><td>0
S</td><td>0
R</td><td>0
N</td><td>0
M</td><td>~</td><td>2
Count</td><td>MessageData$_0$</td></tr>
</table>

Though delivered by the microkernel, this message will appear to have originated from the faulting thread. Its *Label* field is set to the type of the attempted memory operation. On 32-bit architectures, the *label* has the following format:

FFE$_{16}$	0	*op*
15 … 4	3	2 … 0

On 64-bit architectures, the *label* has the following format:

FFF·FFFF·FFFE$_{16}$	0	*op*
48 … 5	4	3 … 0

The *op* field is set to a **Permissions** object which specifies the minimum permissions required to complete the operation. Typically, the permission set will have a value of $\{read\}$, $\{write\}$ or $\{execute\}$, where only one of the available permissions flags is set. On architectures that include $\{execute\}$ in the PMask system parameter described in Section A-10.6, the *op* field may include $\{read\}$ instead, or in addition to $\{execute\}$ for page faults caused by attempts to execute a memory object. Bit 3 of the *label* field is reserved by the microkernel for future use. Under the current OKL4 API, the microkernel always sets bit 3 to 0 and stores the two's complement representation of -2 in the remaining bits of the label.

Execution of the faulting thread will remain suspended until a *page fault acknowledgement message* is received from the page fault handler of the thread, or the IPC operation is aborted using ExchangeRegisters as described in Section A-12.3.

<table>
<tr><td>0
Label</td><td>0
S</td><td>0
R</td><td>0
N</td><td>0
M</td><td>~</td><td>0
Count</td><td>MessageData$_0$</td></tr>
</table>

The page fault handler should send a reply message using user-level IPC operations to the faulting thread. Once the message has been delivered, the microkernel resumes execution of the target thread at the current value of the thread's InstructionPointer register. Unless explicitly changed using the ExchangeRegisters system call, this register contains the location of the faulting instruction which is reattempted by the faulting thread. Page faults and memory permissions are further discussed in Section A-3.2. The **PERMISSIONS** type is further described in Section A-9.31.

A-13.3 Thread Start Protocol

When a new or inactive thread is activated by setting its Pager register using the ThreadControl system call, it is scheduled for immediate execution by the OKL4 thread scheduler. At this stage the InstructionPointer register of the thread which normally contains the **CODE** object being executed is not initialized to a suitable value. Therefore, the OKL4 microkernel commences the thread's execution by issuing an IPC *wait* operation to the thread's pager on behalf of the newly-activated thread. The thread remains blocked until the pager delivers the initial values of the thread's InstructionPointer and StackPointer registers to the new thread in a *thread activation message*. This message has the following format:

<table>
<tr><td colspan="7">StackPointer</td><td>MessageData_2</td></tr>
<tr><td colspan="7">InstructionPointer</td><td>MessageData_1</td></tr>
<tr><td>0
Label</td><td>0
S</td><td>0
R</td><td>0
N</td><td>0
M</td><td>~</td><td>2
Count</td><td>MessageData_0</td></tr>
</table>

On most architectures, thread execution cannot proceed without initializing the StackPointer register to a suitable memory location within the user region of the target thread's address space. Therefore this register is also supplied in the thread activation message. The meaning and valid values of both parameters are architecture-defined.

Upon receiving an activation message, the microkernel copies the supplied values to the InstructionPointer and StackPointer registers of the thread and commences execution of the instruction sequence specified by the InstructionPointer.

A-14 Kernel Debugger

OKL4 provides support for debugging and monitoring using three different mechanisms. Interactive intrusive, external and internal debugging and monitoring are all supported and each may include multiple interfaces or sub-components. Each may be independently used and enabled depending on system configuration.

A-14.1 Console

OKL4 may include a built-in console driver which can be used to display run-time information and verbose booting, warning and assert messages. This feature may be optionally enabled at compile-time and in this case, the architecture or platform is required to implement one or more console device drivers. A number of other configuration options may additionally be selected, such as VERBOSE_INIT or KDB_COLOR_VT.

A-14.2 Kernel Asserts

The OKL4 kernel contains a configurable kernel assert implementation which may be enabled and configured as required by the user. OKL4 kernel asserts can either be disabled by setting the KDB_NO_ASSERTS compile option, or configured by setting ASSERT_LEVEL to one of of the following assert levels:

ASSERT_LEVEL	Type	Description
0	Always	Assert asserts only
1	General	Recommended production kernel
2	Normal	Recommended normal development
3	Debug	Use when debugging
4	Regression	Maximum, all asserts enabled

A-14.3 Command Line Interface

OKL4 provides a built-in command line interface (CLI) debugger that operates on a console device. This intrusive interactive feature is useful for inspecting system state, setting break points and other architecture-specific purposes. This interface may be configured at build-time and is dependant on the console being enabled. The CLI interface operates in two modes. The default *keystroke* mode interprets key-presses as commands, and the *command line* mode takes string-based commands.

The CLI may be configured such that it is entered when certain criteria are met. Firstly, the CLI may be entered upon system-boot prior to threads starting execution. This may be useful in order to enable breakpoints and trace-points prior to starting the system. The CLI may alternatively be configured such that the debugger is entered when a special key-code or a combination of key-codes, is detected. This enables a user to "break-in" to the CLI interface at any time. Finally, the CLI can be enabled by any kernel `ASSERT` or explicit kernel `enter_kdebug` call as well as by a special kernel debug system call which may be used in user code.

The set of available commands in the CLI interface is dependant on the architecture and compile-time options. The list of available commands may be accessed by pressing "`?`" in the keystroke mode or typing "`help`" in the command-line mode.

A-14.4 External Debugging

The OKL4 kernel supports debugging via the use of external devices such as in-circuit debuggers (ICD) or remote memory interfaces such as Firewire and host-mapped PCI. The kernel also allows a number of internal debugging and statistical data-structures to be included at compile-time irrespective of whether the system is configured to include a CLI or other interface. This allows the external device to inspect or modify these data structures for a number of purposes as well as access any other OKL4 internal data structures such as scheduling queues etc.

Standard configuration options include:

- `KMEM_TRACE` enables kernel memory usage statistics
- `TRACEPOINTS` enables kernel tracepoints and the corresponding structures for enabling fine grained enable/disable control
- `TRACEBUFFER` enables kernel trace-buffer support and control structures
- `DEBUG` enables various features such as *thread lists* and *thread names*

A-14.5 Internal Debugging

OKL4 also provides mechanisms for debugging and inspecting kernel state from code running on the system. The two main interfaces are the *kdebug* interface and the *trace-buffer* interface, which may be enabled using the `DEBUG` and `TRACEBUFFER` defines, respectively. These interfaces share access to the same OKL4 internal data structures available to external debuggers.

The kdebug interface allows the user access to various services including controlling tracepoints, inspecting kernel memory usage information, debug console IO and maintaining thread name information. Many of these services can be enabled and disabled at compile time.

The trace-buffer interface is described below.

A-14.6 Kernel Tracing

The OKL4 kernel provides a tracing API which may be accessed via the use of interactive, external and internal interfaces. The tracing infrastructure is designed to be non-blocking and asynchronous, allowing for the real-time tracing of kernel and user operations with minimal impact on the system. The tracing system provides flexible dynamic control of the set of components and events being traced and is independent of the tracepoint system used by the CLI interface.

The tracing system comprises of three components for control and retrieval of trace data.

Tracebuffer The tracebuffer is a contiguous region of memory used for recording event information. Each active tracepoint inserts a new entry in the active buffer as it is passed. A buffer is filled until sufficient space is unavailable for new entries. The buffer is then marked as full. A buffer can only be reused once its full status is cleared.

Control structure The control structure is a data structure used in controlling the trace system. The control structure allows access to information regarding the trace system as well as controlling various aspects of the trace system. It is primarily used to configure the set of active tracepoints and controlling active buffer allocation.

Trace interrupt The trace system provides a means of communicating changes in the trace system to a user thread. A virtual interrupt source is implemented which operates in a similar manner to hardware interrupt threads in **??**. The virtual interrupt is used to signal buffer-full events to the designated trace handler thread.

The trace system allows for up to 32 trace-buffers on a 32-bit architecture, and 64 trace-buffers on a 64-bit architecture. Whenever a trace-buffer becomes full, the kernel switches to using the next available trace-buffer and triggers a virtual trace interrupt. If no unused trace-buffers are available, no new trace events will be recorded. This is known as a trace-buffer overflow.

The main task of the kernel is to write trace events into a trace-buffer and notify the handler thread when necessary. Typically when debugging, the trace handler thread has highest priority in the system so that events are handled prior to trace-buffer overflow and subsequent loss of trace data is prevented. If no handler thread is associated with the virtual trace interrupt, these interrupts are effectively disabled. It is the responsibility of the trace handler thread to mark buffers as available, however an external debugger may also be used for this purpose.

The trace handler thread, when signalled, is free to inspect, store, collect statistics from or discard the trace event data in the trace-buffer before marking the buffer as available for future events.

A-14.6.1 Control structure

The TraceControl structure contains the information and controlling parameters and is located in physical memory provided by the kernel memory resources list described in Section **??**.

The first WORD of the structure contains a signature used to verify the existence of the trace interface. Its value depends on the machine word width. The corresponding signature values are provided below.

	32BIT	**64BIT**
SIGNATURE	`3B2B·1B0B`$_{16}$	`7B6B·5B4B·3B2B·1B0B`$_{16}$

A-14.6.1.1 TRACECONTROL

Constructor:	**Description:**	
TRACECONTROL	The TRACECONTROL Constructor	
Data Field:	**Type:**	**Description:**
Signature	**WORD**	The trace control structure signature
Version	**WORD**	The trace-buffer system version
Identifier	**WORD**	The trace-buffer system identifier
NumBuffers	**WORD**	The number of provided trace-buffers
LogMask	**FLAGS**	The set of enabled major ids
ActiveBuffer	**WORD**	The current active trace-buffer
EmptyBuffers	**FLAGS**	The set of empty (available) trace-buffers
BufferSize	**WORD**	The size of each trace-buffer
BufferPtr$_{list}$	**WORD[]**	The pointers to each buffer
BufferHead$_{list}$	**WORD[]**	The head pointer of each buffer

The *Version* field identifies the version of the trace-buffer interface while the *Identifier* describes the system component associated with the TraceControl structure. The *Identifier* parameter allows the kernel and other components in the system to use the same tracing API. This section however only concerns the kernel provided trace system.

The *LogMask* field provides a mechanism for the coarse grain selection of a set of enabled tracepoints. Each tracepoint has an associated *major* and *minor* number. There are a total of 32 or 64 major numbers for 32-bit and 64-bit architectures, respectively. The *LogMask* is encoded as a bit-mask occupying a single word, where each set-bit enables the major number corresponding to the bit number of the set-bit. Similarly, there is a maximum of 32 or 64 trace-buffers with the actual number specified in *NumBuffers*. The *EmptyBuffers* field is a bit-mask where each set-bit represents an available trace-buffer.

A-14.6.2 Trace-buffer Entries

Trace-buffer entries are written sequentially to a trace-buffer in event arrival order. Entries are always word aligned and consists a header and one or more words of data. The format of the TraceEntry structure is described below.

A-14.6.2.1 TRACEENTRY

Constructor:	Description:	
TRACEENTRY	The TRACEENTRY Constructor	
Data Field:	**Type:**	**Description:**
TimeStamp	**WORD**$_{64}$	The event time-stamp in microseconds
Id	**WORD**$_{8}$	The tracepoint event id
User	**FLAG**	The event source: user or kernel
RecordSize	**WORD**$_{7}$	The entry record size in words
Args	**WORD**$_{5}$	The number of record arguments / parameters
Major	**WORD**$_{5}$	The event's major number
Minor	**WORD**$_{6}$	The event's minor number
Data	**WORD[]**	The trace entry data

The TraceEntry structure describes the data associated with each trace buffer entry. The *TimeStamp* field indicates the time at which the event was recorded, in microseconds starting from the time of system initialisation. The *Id* field uniquely identifies an event and the *Major* and *Minor* numbers are used for event logging. The *User* flag is provided to support the sharing of a trace-buffer between user and kernel components, though this feature is not currently used. The *RecordSize* field encodes the size of the event record in number of words and the *Args* field determines the number of arguments provided by the tracepoint. The *Args* and *Id* fields can be used to determine the format of data in the trace entry.

A-14.7 Configuration Options

The following is a list of configuration options supported by OKL4.

Configuration option	Description
VERBOSE_INIT	Enable printing of kernel initialization messages
ENTER_KDB	Enter kernel KDB CLI at system startup
KDB_BREAKIN	Enable kernel KDB breakin checking
KDB_CLI	Enable the kernel KDB command line interface
KDB_CONS	Enable kernel KDB console for textual output
TRACEPOINTS	Enable tracepoint support
TRACEBUFFER	Enable tracebuffer support
TRACEBUFFER_PAGES	Configure number of pages for tracebuffer memory
KMEM_TRACE	Enable tracing of kernel memory
KDB_NO_ASSERTS	Disable all kernel asserts
ASSERT_LEVEL	Configure level of assert checking (0..4)
KDB_COLOR_VT	Allow colour output on KDB console
THREAD_NAMES	Support textual names associated with threads
SPACE_NAMES	Support textual names associated with address spaces

PART B

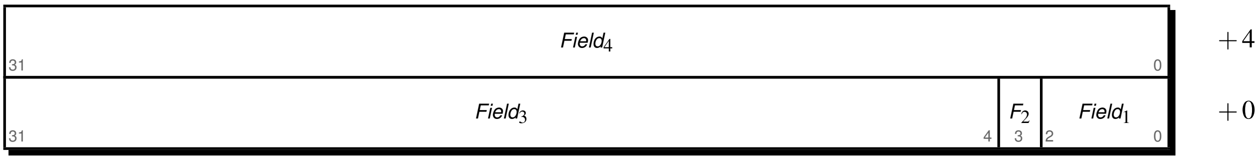
Common Binary Interface

B-1 Overview

The *common binary interfaces* describes the standard encoding of the data constructors defined in Chapter A-9. It is intended to simplify the description of architecture bindings for the OKL4 API discussed in Part C of this manual by factoring out the partial bindings common to all architectures.

The common binary interface is defined for both 32-bit and 64-bit architectures in Chapters B-3 and B-4, respectively. The encoding of all constructors that are identical for both architectures are described in Chapter B-2.

The encoding of objects in memory is described using *constructor diagrams*. Each row represents a single word of information. When a constructor comprising of multiple words is stored in memory, the rows are stored in the little-endian format. That is, the first word of the constructor, represented by the bottom row in the diagram, is stored at the lowest address. Each row is annotated at the right-hand margin with its location within the complete object as a byte offset from the beginning of the object. Bytes within a single word are stored in the architecture-specific byte order. Bits within a word are numbered consecutively, with the least-significant bit stored at bit offset 0. The following *constructor diagram* describes a constructor with four fields *Field*$_1$, *F*$_2$, *Field*$_3$ and *Field*$_4$ on a 32-bit architecture.



Each field is stored in the above structure as follows:

Field$_1$ a 3-bit field stored in bits 0–2 of the first word.

F$_2$ a 1-bit field stored in bit 3 the first word.

Field$_3$ a 28-bit field stored in bits 4–31 of the first word.

Field$_4$ a 32-bit field stored in bits 0–31 of the second word.

The following labels are used to represent fields with common properties:

$1F_{16}$

Constant Fields. When a field marked with a numeric value is read, it always contains the specified value. For example, reading the field in the box to the left will always return the hexadecimal value `1F`$_{16}$. When such a field is modified, the supplied value must always be stored in the field. Unless otherwise stated, storing any other value in that field will render the state of the current address space undefined.

~

Reserved Fields. When a field marked with the "~" sign is read, the returned value is not specified by the current version of the OKL4 API and any bit sequence of the specified size may be returned. When such a field is modified, the value of this field is ignored by OKL4. This field may be modified at any time by the microkernel without user action.

B-2 Common Type Encoding

This chapter describes the encoding of the types defined in Chapter A-9 where the implementation is not dependent on the word size.

B-2.1 Numeric Types

The following types have numeric values which are implemented directly as a sequence of bits representing integer values in binary notation:

WORD$_k$ The **WORD**$_k$ type is encoded directly as a binary k-bit integer. When used for a field or parameter value with a width k' greater than k, an object of type **WORD**$_k$ is automatically converted to an object of type **WORD**$_{k'}$ by setting the $k' - k$ most-significant bits in the resulting object to 0. This conversion is also known as *zero-extension*.

CODE The **CODE** type is encoded directly as a **WORD** object representing the address of the first instruction of the required instruction sequence within the address space of the current thread.

PAGENO The **PAGENO** type is encoded directly as a **WORD**$_{22}$ and **WORD**$_{54}$ object on 32-bit and 64-bit architectures, respectively. The value of the object is the page address divided by 2^{10}. All valid page numbers are an integral multiple of the associated page size as described in Section A-9.30.

B-2.2 Enumerated Types

Many of the types defined in Chapter A-9 have constructors that do not take any parameters. These types are referred to as *enumerated data types* and can be encoded by assigning a unique integer value to each constructor.

B-2.2.1 CACHEOP Encoding

Object Width: 6 bits

Constructor:	**Value:**
FLUSHCACHE	1
FLUSHICACHE	4
FLUSHDCACHE	5
FLUSHALLCACHES	6
LOCKCACHE	8
UNLOCKCACHE	9

B-2.2.2 CachingPolicy Encoding

Object Width: 6 bits

Constructor:	**Value:**
DefaultPolicy	0
CachedPolicy	1
UncachedPolicy	2
WriteBackPolicy	3
WriteThroughPolicy	4
CoherentPolicy	5
DevicePolicy	6
WriteCombiningPolicy	7

B-2.2.3 CapIdType Encoding

Object Width: 4 bits

Constructor:	**Value:**
Access	0_{16}
PhysSegment	5_{16}
ReplyCapability	8_{16}
Special	F_{16}

B-2.2.4 CapOp Encoding

Object Width: 3 bits

Constructor:	**Value:**
CreateClist	0
DeleteClist	1
CreateIpcCap	4
DeleteCap	5

B-2.2.5 ERRORCODE Encoding

Object Width: 8 bits

Constructor:	**Value:**
INVALIDTHREAD	2
INVALIDSPACE	3
INVALIDPARAMETER	5
INVALIDUTCB	6
OUTOFMEMORY	8
SPACENOTEMPTY	9
CLISTNOTEMPTY	9
INVALIDMUTEX	10
MUTEXBUSY	11
INVALIDCLIST	12
INVALIDCAP	13
DOMAINCONFLICT	14
NOTIMPLEMENTED	15

The encoding of IPCERROR is described separately in Chapters B-3 and B-4.

B-2.2.6 DIRECTION Encoding

Object Width: 2 bits

Constructor:	**Value:**
MEMORYCOPYFROM	1
MEMORYCOPYTO	2
MEMORYCOPYBOTH	3

B-2.2.7 FLAG Encoding

Object Width: 1 bits

Constructor:	**Value:**
FALSE	0
TRUE	1

B-2.2.8 IPCERRORCODE Encoding

Object Width: 4 bits

Constructor:	Value:
NOPARTNER	1
INVALIDPARTNER	2
IPCCANCELLED	3
MESSAGEOVERFLOW	4
IPCREJECTED	5
IPCABORTED	7

B-2.2.9 IRQOPERATION Encoding

Object Width: 2 bits

Constructor:	Value:
REGISTER	0
UNREGISTER	1
ACKNOWLEDGE	2
ACKNOWLEDGEWAIT	3

B-2.2.10 THREADSTATE Encoding

Object Width: 8 bits

Constructor:	Value:
UNKNOWNSTATE	0
DEAD	1
INACTIVE	2
RUNNING	3
POLLING	4
SENDING	5
WAITING	6
RECEIVING	7
WAITINGNOTIFY	8
WAITINGXCPU	9

B-3 The 32-Bit Encoding of Data Constructors

This chapter describes the encoding of the remaining data constructors described in Chapter A-9 for 32-bit architectures.

B-3.1 ACCEPTOR Encoding

~	0	0	*Notify*	0
31 4	3	2	1	0

Fields:

Notify	Accept asynchronous notifications

B-3.2 CACHECONTROL Encoding

0	*L6*	*L5*	*L4*	*L3*	*L2*	*L1*	*Op*	*Count*
31 18	17	16	15	14	13	12	11 6	5 0

Fields:

Count	Number of cache regions
Op	The cache operation
L1	Select L1 cache
L2	Select L2 cache
L3	Select L3 cache
L4	Select L4 cache
L5	Select L5 cache
L6	Select L6 cache

B-3.3 CACHEREGIONOP Encoding

I	*C*	*DC*	*IC*	*Count*
31	30	29	28	27 0

Fields:

Count	Number of consecutive 16-byte sized lines
IC	Include instruction cache
DC	Include data cache
C	Clean cache lines
I	Invalidate cache lines

B-3.4 CAPCONTROL Encoding

~	*Op*
31 3	2 0

Fields:

Op	The capability operation

B-3.5 CapId Encoding

The CapId Constructor:

Type	*Index*
31 28	27 0

Fields:

Index	Capability identifier index
Type	Capability identifier type

The ThreadId Constructor:

0	ThreadNo
31 28	27 0

The NilThread Constructor:

$\texttt{F}_{16}$	$\texttt{FFF·FFFC}_{16}$
31 28	27 0

The MySelf Constructor:

$\texttt{F}_{16}$	$\texttt{FFF·FFFD}_{16}$
31 28	27 0

The AnyThread Constructor:

$\texttt{F}_{16}$	$\texttt{FFF·FFFF}_{16}$
31 28	27 0

The WaitNotify Constructor:

$\texttt{F}_{16}$	$\texttt{FFF·FFFE}_{16}$
31 28	27 0

B-3.6 CopFlags Encoding

C7	*C6*	*C5*	*C4*	*C3*	*C2*	*C1*	*C0*
7	6	5	4	3	2	1	0

Fields:

C0	Coprocessor 0
C1	Coprocessor 1
C2	Coprocessor 2
C3	Coprocessor 3
C4	Coprocessor 4
C5	Coprocessor 5
C6	Coprocessor 6
C7	Coprocessor 7

B-3.7 Date Encoding

Year	*Month*	*Day*
15 9	8 5	4 0

Fields:

Day	Day component
Month	Month component
Year	Year component

B-3.8 ERRORCODE Encoding

The IPCERROR Constructor:

~	*Cause*	*P*
31 5	4 1	0

Fields:

P	Phase flag
Cause	Error code

B-3.9 FPAGE Encoding

The FPAGE Constructor:

Base	*Width*	*Meta*	*Access*
31 10	9 4	3	2 0

Fields:

Access	The required permissions (*P*)
Meta	Extended rights attribution
Width	Width of the page (*w*)
Base	The page number (*b*)

The WHOLESPACE Constructor:

0	1	0	*Access*
31 10	9 4	3	2 0

Fields:

Access	The required permissions (*P*)

The NILPAGE Constructor:

0	0	0
31 10	9 4	3 0

B-3.10 INTCONTROL Encoding

NotifyBit	~	*Request*	*Operation*	*Count*
31 27	26	25 8	7 6	5 0

Fields:

Count	Highest message register
Operation	Operation type
Request	Platform specific request parameter

B-3.11 MAPCONTROL Encoding

M	*Q*	*Win*	0	*Count*
31	30	29	28 6	5 0

Fields:

Count	Number of map items - 1
Win	Perform window mapping
Q	Query current mapping data
M	Modify mapping data

B-3.12 MAPITEM Encoding

Seg	31	0	+8
Size	31	0	+4
Virt	31	0	+0

Fields:

Virt	Virtual descriptor
Size	Size descriptor
Seg	Segment descriptor

B-3.13 MEMORYCOPYDESCRIPTOR Encoding

Direction 31 30	~ 29 0	+8
Size 31 0		+4
Address 31 0		+0

Fields:

Address	Address of remote buffer
Size	Size of remote buffer
Direction	MemoryCopy directions

B-3.14 MSGTAGIN Encoding

Label	*S*	*R*	*N*	*M*	~	*Untyped*
31 16	15	14	13	12	11 6	5 0

Fields:

Untyped	Number of data words
M	The memory copy flag
N	The asynchronous notification flag
R	The *receive* blocking flag
S	The *send* blocking flag
Label	Message label

B-3.15 MSGTAGOUT Encoding

Label	*E*	*X*	~	~	~	*Untyped*
31 16	15	14	13	12	11 6	5 0

Fields:

Untyped	Number of data words
X	The remote-IPC flag
E	The error flag
Label	Message label

B-3.16 PERMISSIONS Encoding

X	*W*	*R*
2	1	0

Fields:

R	*read* permission
W	*write* permission
X	*execute* permission

B-3.17 PLATCONTROL Encoding

Request	*Count*
31 6	5 0

Fields:

Count	Number of input parameters
Request	Implementation specific

B-3.18 PREEMPTIONFLAGS Encoding

~	*Signalled*	~
7 6	5	4 0

Fields:

Signalled	The pre-emption notification flag

B-3.19 QUERYDESCRIPTOR Encoding

~	0	*PageRef*	*PageSize*	0	*Access*
31 14	13	12 10	9 4	3	2 0

Fields:

Access	Requested permissions for the mapping
PageSize	Size of pages to be used in mapping (power of 2)
PageRef	Current referenced bits

B-3.20 QUERYITEM Encoding

Seg 31 0	+8
Query 31 0	+4
Virt 31 0	+0

Fields:

Virt	Virtual descriptor
Query	Query descriptor
Seg	Segment descriptor

B-3.21 REGCONTROL Encoding

Source	Rset	Rget	~	CPY	D	H	Tls	UH	FL	IP	SP	AS	AR	HE
31 14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Fields:

HE	Enable halt/resume operation
AR	Abort receive operation
AS	Abort send operation
SP	Set StackPointer
IP	Set InstructionPointer
FL	Set Flags
UH	Set UserHandle
Tls	Set pointer to TLS
H	Halt/Resume
D	Deliver old register values
CPY	Copy thread's registers
Rget	Deliver thread's registers
Rset	Modify thread's registers
Source	Source thread number

B-3.22 SEGMENTDESCRIPTOR Encoding

Offset	Segment
31 10	9 0

Fields:

Segment	Memory to be mapped
Offset	Offset from the start of physical memory to be mapped

B-3.23 SIZEDESCRIPTOR Encoding

NumPages	PageSize	Valid	Access
31 10	9 4	3	2 0

Fields:

Access	Requested permissions for the mapping
Valid	Map (1) or unmap (0) operation
PageSize	Size of pages to be used in mapping (power of 2)
NumPages	Number of pages to be mapped

B-3.24 SPACECONTROL Encoding

MaxPriority	~	Resourced	~	Resources	Delete	New
31 24	23 5	4	3	2	1	0

Fields:

New	Create new address space
Delete	Delete address space
Resources	Modify and retrieve old space resources
Resourced	Give new space access to resources
MaxPriority	Maximum priority threads in space can assign

B-3.25 SPACEID Encoding

The ROOTSPACE Constructor:

0	
31	0

The NILSPACE Constructor:

FFFF·FFFF$_{16}$	
31	0

B-3.26 SPACERESOURCES Encoding

The DEFAULTSPACERESOURCES Constructor:

0	
31	0

B-3.27 THRDRESOURCES Encoding

The DEFAULTTHREADRESOURCES Constructor:

0	
31	0

B-3.28 VIRTUALDESCRIPTOR Encoding

VirtualBase	*Sparse*	*Replace*	*Attr*
31 10	9	8	7 0

Fields:

Attr	Caching attributes of the mapping
Replace	Replace an existent mapping
Sparse	Sparse unmapping
VirtualBase	Base of the virtual address

B-4 The 64-Bit Encoding of Data Constructors

This chapter describes the encoding of the remaining data constructors described in Chapter A-9 for 64-bit architectures.

B-4.1 ACCEPTOR Encoding

~	0	0	*Notify*	0
63 … 4	3	2	1	0

Fields:

Notify Accept asynchronous notifications

B-4.2 CACHECONTROL Encoding

0	*L6*	*L5*	*L4*	*L3*	*L2*	*L1*	*Op*	*Count*
63 … 18	17	16	15	14	13	12	11 … 6	5 … 0

Fields:

Count	Number of cache regions
Op	The cache operation
L1	Select L1 cache
L2	Select L2 cache
L3	Select L3 cache
L4	Select L4 cache
L5	Select L5 cache
L6	Select L6 cache

B-4.3 CACHEREGIONOP Encoding

I	*C*	*DC*	*IC*	*Count*
63	62	61	60	59 … 0

Fields:

Count	Number of consecutive 16-byte sized lines
IC	Include instruction cache
DC	Include data cache
C	Clean cache lines
I	Invalidate cache lines

B-4.4 CAPCONTROL Encoding

~	*Op*
63 … 3	2 … 0

Fields:

Op The capability operation

B-4.5 CapId Encoding

The CapId Constructor:

Type	*Index*
63 60	59 0

Fields:

Index	Capability identifier index
Type	Capability identifier type

The ThreadId Constructor:

0	ThreadNo
63 60	59 0

The NilThread Constructor:

$\mathtt{F}_{16}$	$\mathtt{FFF{\cdot}FFFF{\cdot}FFFF{\cdot}FFFC}_{16}$
63 60	59 0

The MySelf Constructor:

$\mathtt{F}_{16}$	$\mathtt{FFF{\cdot}FFFF{\cdot}FFFF{\cdot}FFFD}_{16}$
63 60	59 0

The AnyThread Constructor:

$\mathtt{F}_{16}$	$\mathtt{FFF{\cdot}FFFF{\cdot}FFFF{\cdot}FFFF}_{16}$
63 60	59 0

The WaitNotify Constructor:

$\mathtt{F}_{16}$	$\mathtt{FFF{\cdot}FFFF{\cdot}FFFF{\cdot}FFFE}_{16}$
63 60	59 0

B-4.6 CopFlags Encoding

C7	*C6*	*C5*	*C4*	*C3*	*C2*	*C1*	*C0*
7	6	5	4	3	2	1	0

Fields:

C0	Coprocessor 0
C1	Coprocessor 1
C2	Coprocessor 2
C3	Coprocessor 3
C4	Coprocessor 4
C5	Coprocessor 5
C6	Coprocessor 6
C7	Coprocessor 7

B-4.7 Date Encoding

Year	*Month*	*Day*
15 9	8 5	4 0

Fields:

Day	Day component
Month	Month component
Year	Year component

B-4.8 ERRORCODE Encoding

The IPCERROR Constructor:

~	*Cause*	*P*
63 … 5	4 … 1	0

Fields:

P	Phase flag
Cause	Error code

B-4.9 FPAGE Encoding

The FPAGE Constructor:

Base	*Width*	*Meta*	*Access*
63 … 10	9 … 4	3	2 … 0

Fields:

Access	The required permissions (*P*)
Meta	Extended rights attribution
Width	Width of the page (*w*)
Base	The page number (*b*)

The WHOLESPACE Constructor:

0	1	0	*Access*
63 … 10	9 … 4	3	2 … 0

Fields:

Access	The required permissions (*P*)

The NILPAGE Constructor:

0	0	0
63 … 10	9 … 4	3 … 0

B-4.10 INTCONTROL Encoding

NotifyBit	*Request*	*Operation*	*Count*
63 … 58	57 … 8	7 … 6	5 … 0

Fields:

Count	Highest message register
Operation	Operation type
Request	Platform specific request parameter

B-4.11 MAPCONTROL Encoding

M	*Q*	*Win*	0	*Count*
63	62	61	60 … 6	5 … 0

Fields:

Count	Number of map items - 1
Win	Perform window mapping
Q	Query current mapping data
M	Modify mapping data

B-4.12 MAPITEM Encoding

63	*Seg*	0	+16
63	*Size*	0	+8
63	*Virt*	0	+0

Fields:

Virt	Virtual descriptor
Size	Size descriptor
Seg	Segment descriptor

B-4.13 MEMORYCOPYDESCRIPTOR Encoding

Direction	63 62	~	61	0	+16
Size			63	0	+8
Address			63	0	+0

Fields:

Address	Address of remote buffer
Size	Size of remote buffer
Direction	MemoryCopy directions

B-4.14 MSGTAGIN Encoding

Label	*S*	*R*	*N*	*M*	~	*Untyped*
63 16	15	14	13	12	11 6	5 0

Fields:

Untyped	Number of data words
M	The memory copy flag
N	The asynchronous notification flag
R	The *receive* blocking flag
S	The *send* blocking flag
Label	Message label

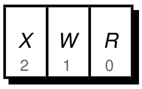
B-4.15 MSGTAGOUT Encoding

Label	*E*	*X*	~	~	~	*Untyped*
63 16	15	14	13	12	11 6	5 0

Fields:

Untyped	Number of data words
X	The remote-IPC flag
E	The error flag
Label	Message label

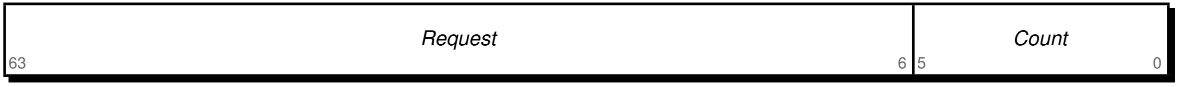
B-4.16 PERMISSIONS Encoding



Fields:

R	*read* permission
W	*write* permission
X	*execute* permission

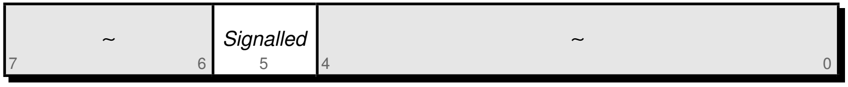
B-4.17 PLATCONTROL Encoding



Fields:

Count	Number of input parameters
Request	Implementation specific

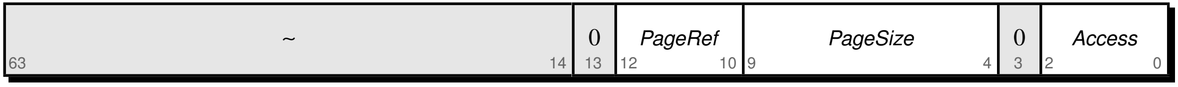
B-4.18 PREEMPTIONFLAGS Encoding



Fields:

Signalled	The pre-emption notification flag

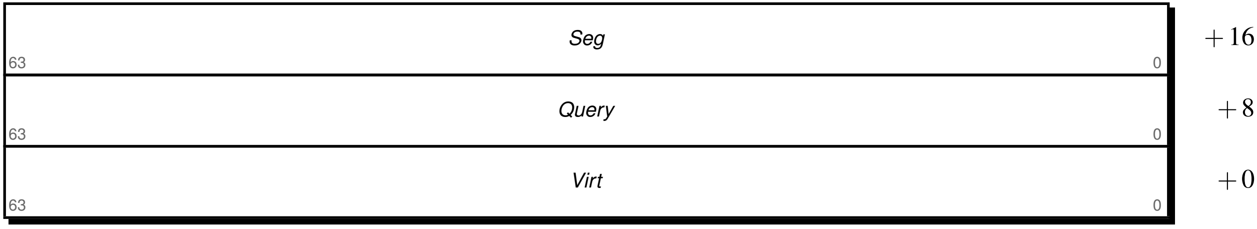
B-4.19 QUERYDESCRIPTOR Encoding



Fields:

Access	Requested permissions for the mapping
PageSize	Size of pages to be used in mapping (power of 2)
PageRef	Current referenced bits

B-4.20 QUERYITEM Encoding



Fields:

Virt	Virtual descriptor
Query	Query descriptor
Seg	Segment descriptor

B-4.21 RegControl Encoding

Source	0	*Rset*	*Rget*	~	*CPY*	*D*	*H*	*Tls*	*UH*	*FL*	*IP*	*SP*	*AS*	*AR*	*HE*
63 32	31 14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Fields:

HE	Enable halt/resume operation
AR	Abort receive operation
AS	Abort send operation
SP	Set StackPointer
IP	Set InstructionPointer
FL	Set Flags
UH	Set UserHandle
Tls	Set pointer to TLS
H	Halt/Resume
D	Deliver old register values
CPY	Copy thread's registers
Rget	Deliver thread's registers
Rset	Modify thread's registers
Source	Source thread number

B-4.22 SegmentDescriptor Encoding

Offset	*Segment*
63 10	9 0

Fields:

Segment	Memory to be mapped
Offset	Offset from the start of physical memory to be mapped

B-4.23 SizeDescriptor Encoding

NumPages	*PageSize*	*Valid*	*Access*
63 10	9 4	3	2 0

Fields:

Access	Requested permissions for the mapping
Valid	Map (1) or unmap (0) operation
PageSize	Size of pages to be used in mapping (power of 2)
NumPages	Number of pages to be mapped

B-4.24 SpaceControl Encoding

MaxPriori	~	*Resourced*	~	*Resources*	*Delete*	*New*
63 56	55 5	4	3	2	1	0

Fields:

New	Create new address space
Delete	Delete address space
Resources	Modify and retrieve old space resources
Resourced	Give new space access to resources
MaxPriority	Maximum priority threads in space can assign

B-4.25 SPACEID Encoding

The ROOTSPACE **Constructor:**

0
63 0

The NILSPACE **Constructor:**

FFFF·FFFF·FFFF·FFFF$_{16}$
63 0

B-4.26 SPACERESOURCES Encoding

The DEFAULTSPACERESOURCES **Constructor:**

0
63 0

B-4.27 THRDRESOURCES Encoding

The DEFAULTTHREADRESOURCES **Constructor:**

0
63 0

B-4.28 VIRTUALDESCRIPTOR Encoding

VirtualBase	*Sparse*	*Replace*	*Attr*
63 10	9	8	7 0

Fields:

Attr	Caching attributes of the mapping
Replace	Replace an existent mapping
Sparse	Sparse unmapping
VirtualBase	Base of the virtual address

B-5 Booting

PART C

Architecture Bindings

C-1 Overview

Architecture bindings to the OKL4 API describes the method used to provide architecture-specific parts of the OKL4 API on a particular architecture. As substantial variations exist between each different architecture, each is described separately. Each architecture binding described in this chapter conforms to either the 32-bit or 64-bit common binary interface described in Chapters B-3 and B-4, respectively.

Architecture bindings have been described using the native assembly language syntax published in the relevant architecture manual. The layout of memory-resident objects are presented using data structure diagrams described in Chapter B-1.

The implementation of the system call interface is described using the *system call diagrams*. For example, the following system call diagram describes a system call invoked using the instruction `syscall 0x10` with three input parameters $\mathit{ParamX}_{\mathrm{IN}}$, $\mathit{ParamY}_{\mathrm{IN}}$ and $\mathit{ParamZ}_{\mathrm{IN}}$ passed in registers `%r1`, `%r2` and `%r3` respectively, and two output parameters $\mathit{ParamA}_{\mathrm{OUT}}$ and $\mathit{ParamB}_{\mathrm{OUT}}$ returned in registers `%r1` and `%r2`:

$\mathit{ParamX}_{\mathrm{IN}}$	`%r1`	`syscall 0x10` →	`%r1`	$\mathit{ParamA}_{\mathrm{OUT}}$
$\mathit{ParamY}_{\mathrm{IN}}$	`%r2`		`%r2`	$\mathit{ParamB}_{\mathrm{OUT}}$
$\mathit{ParamZ}_{\mathrm{IN}}$	`%r3`			

The left-hand side of the figure contains the input parameters annotated with the names of registers or memory locations containing the data. The sequence of machine instructions used to invoke the system call is specified in the center. The values returned by the system call are listed on the right-hand side, accompanied by the names of registers or memory locations in which they will be stored by the call.

The following field labels are used to represent common parameters and return values in data structure diagrams:

$\boxed{1F_{16}}$ **Constants.** When an input parameter is marked with a numeric value, that value must always be supplied in the corresponding register to the system call. Unless otherwise stated, supplying any other value will render the state of the current address space *undefined*. Where an output parameter is marked with a numeric value, it will always contain the specified value upon returning from the system call.

$\boxed{\sim}$ **Reserved Registers.** When an input parameter is marked with the label " ~ ", the value of the corresponding register is ignored by the system call. When an output parameter is marked with the label "~", the system call may modify the corresponding register in an unpredictable fashion.

$\boxed{\equiv}$ **Preserved Registers.** When an output parameter is marked with the label " ≡ ", the microkernel preserves the value of the register across the system call.

Unless otherwise stated, all architecture and virtual registers not mentioned explicitly in the system call diagram are preserved by the system call.

C-2 ARM Architecture Bindings

This chapter describes the OKL4 API bindings for the family of ARM architectures. The ARM assembly language syntax defined by ARM Ltd. is used to describe the architecture bindings. For further information about the ARM architecture and its assembly language syntax, refer to *ARM Architecture Reference Manual* published by Addison-Wesley (ISBN 0-201-73719-1).

ARM is a 32-bit architecture that is designed for embedded applications. Some versions (ARMv4 and ARMv5) define a virtually-indexed memory cache design, that poses a number of challenges to the implementation of the OKL4 microkernel. This cache design makes it generally impossible to map the same physical page to multiple virtual addresses, irrespective of the address space, except where the user is able to ensure that all such address aliases are indexed in the same cache slot or have a suitable cache coherence protocol. This is only possible if a detailed knowledge of the cache implementation parameters is assumed. The OKL4 ARM bindings described in this chapter permits unrestricted aliasing of memory, leaving controlling the validity of such aliases to the operating system personality. An illegal address space configuration will render the state of that address space undefined.

The OKL4 kernel ensures that cache aliases occurring due to reusing virtual addresses in two or more distinct address spaces never results in memory corruption for those processors concerned. This however results in poor performance due to the resulting cache flushing on context switches. To alleviate this problem, the ARM architecture bindings provide two extensions to the OKL4 API, the *ARM-PID* extension and the *virtual space identifiers* extension.

The ARM-PID extension allows the operating system personality to annotate some address spaces with a 7-bit PID. The full impact of this extension is described in the *Fast Context Switch Extension* section of the *ARM Architecture Reference Manual*. All addresses supplied to and returned by the microkernel, including the UTCB location, correspond to the MVA as defined in the *ARM Architecture Reference Manual*.

The virtual space identifier extension allows operating system personalities to explicitly identify sets of non-overlapping user address spaces. The operating system personality may assign a unique identifier to each set of non-overlapping address spaces, which allows the microkernel to optimise system performance by eliminating redundant context switches. The responsibility for the management and recycling of virtual space identifiers is left to the operating system personality. Any attempt to mislead the microkernel by providing an incorrect PID or virtual space identifier will render the states of the involved address spaces undefined.

C-2.1 System Parameters

Parameter:	Value:
MaxMessageData	32
PageSizeMask	$\texttt{0011·1400}_{16}$ or $\texttt{0011·1000}_{16}$ or $\texttt{0111·1000}_{16}$
PMask	$\{$*write*$\}$ ≤ ARMv5, $\{$*write, execute*$\}$ ≥ ARMv6
UtcbAlignment	8
UtcbRegionWidth	0 ≤ ARMv5, 12 ≥ ARMv6
UtcbMultiplier	1

The ARM bindings of kernel parameter values fixed by the OKL4 API are listed in the above table. MinPageWidth and PageSizeMask parameters depend on the features of the particular ARM processor variant, specific CPU bindings may choose to enable or disable support for 1KB pages as appropriate. Values of all other system parameters are platform-defined and contained in the documentation accompanying the particular platform.

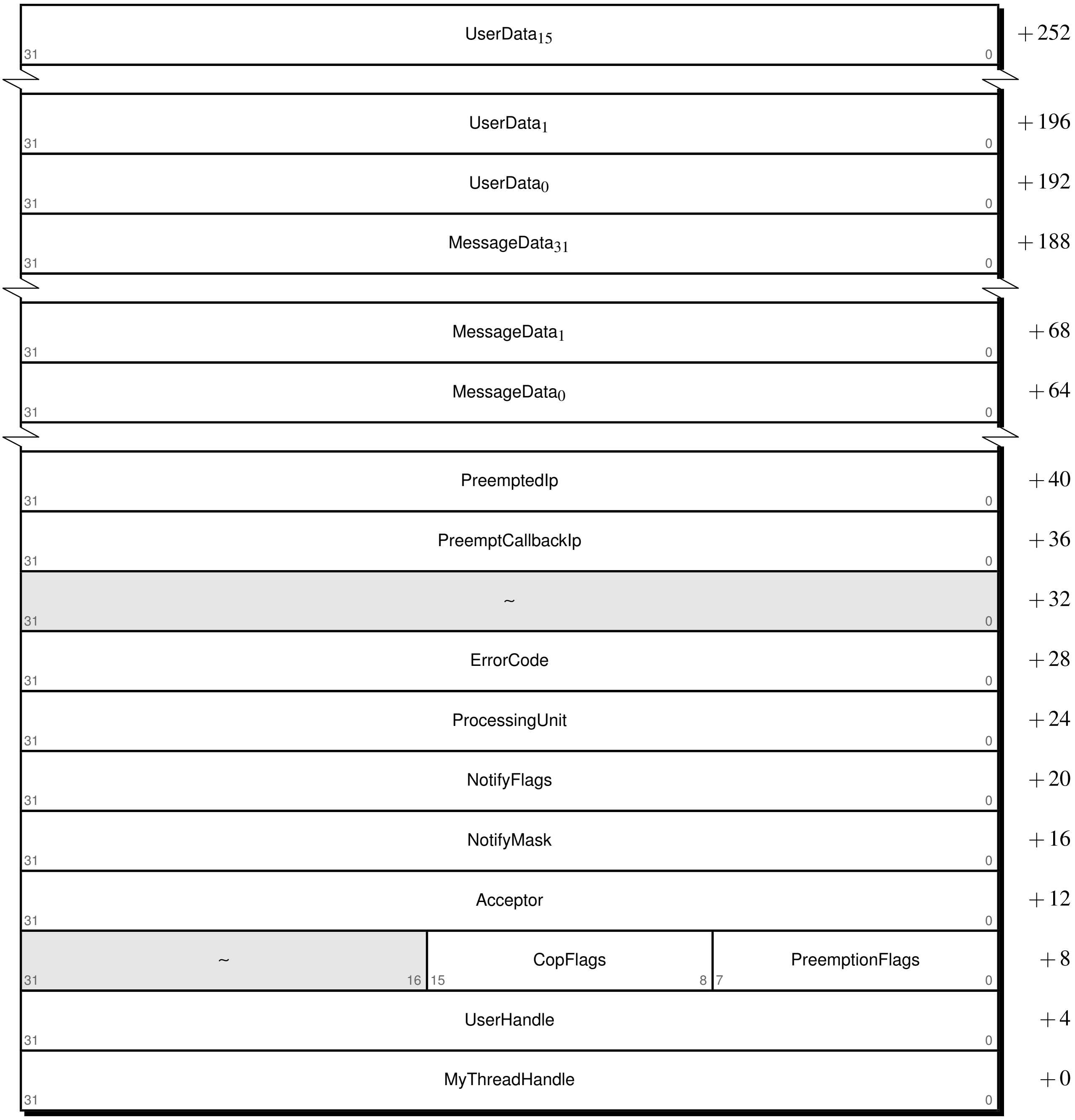
C-2.2 Virtual Registers

Virtual Register:	Hardware Register:
Flags	CPSR
InstructionPointer	R15
StackPointer	R13
Utcb	$\texttt{FF00·0FF0}_{16}$ - Virtual Address
Parameter_0	R0
Parameter_1	R1
Parameter_2	R2
Parameter_3	R3
Parameter_4	R4
Parameter_5	R5
Parameter_6	R6
Parameter_7	R7
Result_0	R0
Result_1	R1
Result_2	R2
Result_3	R3
Result_4	R4
Result_5	R5
Result_6	R6
Result_7	R7
MessageData_0	R3 - Ipc only
MessageData_1	R4 - Ipc only
MessageData_2	R5 - Ipc only
MessageData_3	R6 - Ipc only
MessageData_4	R7 - Ipc only
MessageData_5	R8 - Ipc only

On the ARM architecture, all input and result registers, the InstructionPointer, StackPointer and Flags are mapped to the general-purpose registers as defined in the above table. Additionally, the $\text{MessageData}_{0..5}$ are mapped onto hardware registers when invoking or returning from the Ipc system call. The Utcb register is stored at the fixed memory location $\texttt{FF00·0FF0}_{16}$ in the reserved region of the address space. All remaining registers are stored in the thread's UTCB as described in Section C-2.3.

Note that Parameter_3 to Parameter_7 and Result_3 to Result_7 registers are bound to the same hardware registers as the first six MessageData registers in the Ipc case. This is possible as the Ipc system call does not use both registers of an overlapping pair simultaneously.

C-2.3 The UTCB Entry Layout



On the ARM architecture, thread control blocks are managed internally by the kernel. The address of the current thread's UTCB can be read from the memory location `FF00·0FF0`$_{16}$. The UTCB is an object with the structure described in the above diagram.

The UTCB contains the values of all the MessageData registers, except when calling or returning from an Ipc operations. The value of the MessageData$_i$ is stored at byte offset $64+4i$ in the UTCB for values of i between 0 and MaxMessageData (6 and MaxMessageData for Ipc), inclusive. The 14 words at byte offsets 192–255 are used to provide additional UserData registers. The value of the UserData$_i$ register is stored at offset $192+4i$ for values of i between 0 and 15, inclusive.

An attempt to access undefined locations within the UTCB may result in undefined operation of the associated thread and may cause the delivery of page fault messages as described in Section A-3.2 unless otherwise stated in the documentation accompanying the particular OKL4 platform.

C-2.4 Caching Policies

All ARM architecture variants on OKL4 implement or emulate all caching policies described in Sections A-9.6 and A-3.3. Prior to disabling the caching of a particular memory resource, the user must ensure that all memory pages overlapping that resource are flushed from the cache using the CacheControl system call described in Section A-12.1.

C-2.5 The Flags Register

N	Z	C	V	Q	~	J	~	GE	~	E	~	T	~	CPSR
31	30	29	28	27	26 25	24	23 20	19 16	15 10	9	8 6	5	4 0	

All versions of the OKL4 microkernel on ARM architectures bind the Flags virtual register to the `CPSR` hardware registers. Most fields in the `CPSR` register cannot be controlled by software and are left unmodified following an ExchangeRegisters system call that modifies the Flags register of a user thread. Only those fields of the `CPSR` which do not represent a security problem may be modified, the specific fields available are defined in the *ARM Architecture Reference Manual* and CPU specific documentation.

C-2.6 The CopFlags Virtual Register

The ArmCopFlags Constructor:

~
7 0

The ARM architecture does not define any user-selectable subsets of the architecture that may be selectively disabled for a potential gain in performance. Consequently all flags in the CopFlags virtual register remain undefined.

C-2.7 Additional SpaceResources Constructors

The ArmSpaceResources Constructor:

VSpace	0	*PID*
31 16	15 7	6 0

The ARM bindings of the OKL4 microkernel define an additional ARM-specific constructor for the **SpaceResources** type used by the SpaceControl system call described in Section A-12.12. This constructor contains two additional fields, *PID* and *VSpace* used by ARM variants that have been configured to support the *ARM-PID* and *virtual space identifier* extensions, respectively. When both fields are set to 0, or where the API extensions corresponding to the non-zero fields supplied in the *SpaceResources*$_{\text{IN}}$ parameter are not supported by the ARM variant, the SpaceControl system call behaves as described in Section A-12.12. Otherwise, the behaviour of SpaceControl is affected as described in Sections C-2.7.1 and C-2.7.2.

C-2.7.1 The ARM-PID Extension

If the OKL4 kernel has been configured with the *ARM-PID* extension (ARMv4, ARMv5, ARMv6) and the *PID* field is set to a non-zero value in the ArmSpaceResources constructor supplied in the *SpaceResources*$_{\text{IN}}$ parameter to the SpaceControl system call, its value is used to configure the `PID` hardware register for all threads in the corresponding address space. Following this SpaceControl invocation, all addresses received and supplied by the microkernel must use the modified virtual addresses (MVA). The *ARM-PID* extension, is further described in the *Fast Context Switching Extension* chapter in the *ARM Architecture Reference Manual*.

C-2.7.2 The Virtual Space Identifier Extension

If the OKL4 ARM variant has been configured with the *virtual space identifier* extension (ARMv4, ARMv5) and the *VSpace* field is set to a non-zero value in the ARMSPACERESOURCES constructor supplied in the *SpaceResources*$_{\text{IN}}$ parameter to the SpaceControl system call, its value is used to identify groups of OKL4 address spaces with no conflicting aliases of the same memory-mapped resource. Specifically, the microkernel assumes that all memory-mapped resources accessible through all address spaces with the same virtual space identifier are cached at the same location within the ARM virtually-indexed memory caches. Accordingly the microkernel may optimise system performance by avoiding cache flushing during context switches between threads belonging to a single address space group. The responsibility for the management and recycling of virtual space identifiers is left entirely to the operating system personality. Any attempt to mislead the microkernel by grouping together address spaces with conflicting memory mappings renders the states of all address spaces configured with the corresponding virtual space identifier undefined.

C-2.8 Additional THRDRESOURCES Constructors

The ARMTHRDRESOURCES Constructor:

~	*IWMMX*	*VFP*	*FPA*
31 3	2	1	0

The ARM bindings of the OKL4 microkernel define an additional ARM-specific constructor for the **THRDRESOURCES** type used by the ThreadControl system call described in Section A-12.14. This constructor contains three additional flags, *FPA*, *VFP* and *IWMMX* used by ARM variants that have support for the FPA, VFP and IWMMX coprocessors respectively. All other locations in the **THRDRESOURCES** type are reserved for future use and must be set to zero.

Each of these flags, when set using the ThreadControl system call for a particular thread, enable access to the corresponding resource for the target thread. All threads are provided with an exclusive copy of the resource's state unless otherwise stated. This allows more than one thread to use a resource without affecting the visible state of the resource seen by any other threads. For example, all threads with VFP support have their own copies of the floating point registers and status word. Note, enabling access to a resource may fail due to a out-of-memory condition. This is indicated in the error output of ThreadControl.

Definitions:

- FPA - Floating Point Accelerator coprocessor
- VFP - Vector Floating Point coprocessor
- IWMMX - Intel Wireless MMX extensions coprocessor

C-2.9 Exception Protocol

<table>
<tr><td colspan="7">*cause*</td><td>MessageData$_5$</td></tr>
<tr><td colspan="7">*exception*</td><td>MessageData$_4$</td></tr>
<tr><td colspan="7">CPSR</td><td>MessageData$_3$</td></tr>
<tr><td colspan="7">R13 (SP)</td><td>MessageData$_2$</td></tr>
<tr><td colspan="7">R15 (PC)</td><td>MessageData$_1$</td></tr>
<tr><td>*label*
Label</td><td>0
S</td><td>0
R</td><td>0
N</td><td>0
M</td><td>~</td><td>5
Count</td><td>MessageData$_0$</td></tr>
</table>

All ARM variants of the OKL4 microkernel deliver all *undefined instruction* exceptions, external *data abort* exceptions and any other *data abort* or *coprocessor* exceptions that are not handled internally by the microkernel to the thread's exception handler as described in Section A-13.1. The format of the exception message is defined in the above message diagram. The *label* field in the exception message has the following format:

15 … 4	3 … 0
FFB$_{16}$	0

The *exception* field contains a value of 1 for *undefined instruction* exceptions or 100$_{16}$ +*fault status* for data abort exceptions. For *undefined instruction* exceptions, the *cause* field is set to the value of the instruction that caused the exception. For *data abort* exceptions, the *fault status* is encoded as the concatenation of bits [10,3:0] of the ARM DFSR or IFSR registers for ARMv6 processors. The *cause* field is set to the virtual address of the data abort which the thread attempted to access. All other exceptions are handled by the microkernel and are not delivered to the user threads. Hardware exceptions on ARM architectures are further described in the *ARM Architecture Reference Manual*.

On ARM processors that the support vector floating-point (VFP) coprocessor, the message format is modified to include additional processor state.

FPINST2						MessageData$_8$
FPSCR						MessageData$_7$
FPINST						MessageData$_6$
cause (FPEXC)						MessageData$_5$
exception						MessageData$_4$
CPSR						MessageData$_3$
R13 (SP)						MessageData$_2$
R15 (PC)						MessageData$_1$
label Label	0 S	0 R	0 N	0 M	~	8 Count	MessageData$_0$

The *exception* field contains a value of 8 for *VFP instruction* exceptions. The *label* has the same encoding as the normal exception message encoding described above. VFP exceptions on ARM architectures are further described in the *ARM Vector Floating-point Coprocessor* documentation.

C-2.10 System Call Emulation

`CPSR`	MessageData$_{13}$
instruction	MessageData$_{12}$
`R14` (`LR`)	MessageData$_{11}$
`R13` (`SP`)	MessageData$_{10}$
`R15` (`PC`)	MessageData$_{9}$
`R3`	MessageData$_{8}$
`R2`	MessageData$_{7}$
`R1`	MessageData$_{6}$
`R0`	MessageData$_{5}$
`R7`	MessageData$_{4}$
`R6`	MessageData$_{3}$
`R5`	MessageData$_{2}$
`R4`	MessageData$_{1}$

label	0	0	0	0	~	13	MessageData$_{0}$
Label	*S*	*R*	*N*	*M*		*Count*	

All ARM variants of the OKL4 microkernel convert non OKL4 system call uses of the `SWI` instruction into an exception delivered to the exception handler of the thread using the message format specified by the above message diagram. The *label* field in the message has the following format:

`FFB`$_{16}$	0
15 … 4	3 … 0

The *instruction* field is set to the value of the `SWI` instruction. The exception handler can obtain the system call number from the least-significant 24 bits of this field.

C-2.11 System Calls

All system calls destroy the values of all ARM general-purpose registers except for any other registers that are specified in the corresponding system call diagram. All OKL4 system calls use the `R13` register to encode the system call number and the `R12` register is used to store the saved `R13` register. After a system call, the saved value in `R12` is restored in `R13`. All system calls return to the next instruction after the instruction invoking the system call.

Note that OKL4 does not interpret the constant argument associated with the `SWI` instruction (for example, `SWI 0x1414`). The values are provided simply as an aid in debugging.

C-2.11.1 CacheControl

Target$_{IN}$	R0		R0	*Result*$_{OUT}$
Control$_{IN}$	R1		R1	~
~	R2		R2	~
~	R3		R3	~
~	R4		R4	~
~	R5		R5	~
~	R6	`move r12, r13`	R6	~
~	R7	`move r13, 0xFFFFFF20`	R7	~
~	R8	`SWI 0x1420`	R8	~
~	R9	⟶	R9	~
~	R10		R10	~
~	R11		R11	~
~	R12		R12	~
≡	R13		R13	≡
~	R14		R14	~
~	R15		R15	*next instruction*

C-2.11.2 CapControl

Clist$_{IN}$	R0		R0	*Result*$_{OUT}$
Control$_{IN}$	R1		R1	~
~	R2		R2	~
~	R3		R3	~
~	R4		R4	~
~	R5		R5	~
~	R6	`move r12, r13`	R6	~
~	R7	`move r13, 0xFFFFFF40`	R7	~
~	R8	`SWI 0x1440`	R8	~
~	R9	⟶	R9	~
~	R10		R10	~
~	R11		R11	~
~	R12		R12	~
≡	R13		R13	≡
~	R14		R14	~
~	R15		R15	*next instruction*

C-2.11.3 ExchangeRegisters

Target$_{\text{IN}}$	R0		R0	*Result*$_{\text{OUT}}$
Control$_{\text{IN}}$	R1		R1	*Control*$_{\text{OUT}}$
StackPointer$_{\text{IN}}$	R2		R2	*StackPointer*$_{\text{OUT}}$
InstructionPointer$_{\text{IN}}$	R3		R3	*InstructionPointer*$_{\text{OUT}}$
Flags$_{\text{IN}}$	R4		R4	*Flags*$_{\text{OUT}}$
UserHandle$_{\text{IN}}$	R5		R5	*UserHandle*$_{\text{OUT}}$
~	R6	`move r12, r13`	R6	~
~	R7	`move r13, 0xFFFFFF0C`	R7	~
~	R8	`SWI 0x140C`	R8	~
~	R9	⟶	R9	~
~	R10		R10	~
~	R11		R11	~
~	R12		R12	~
≡	R13		R13	≡
~	R14		R14	~
~	R15		R15	*next instruction*

C-2.11.4 InterruptControl

Target$_{\text{IN}}$	R0		R0	*Result*$_{\text{OUT}}$
Control$_{\text{IN}}$	R1		R1	~
~	R2		R2	~
~	R3		R3	~
~	R4		R4	~
~	R5		R5	~
~	R6	`move r12, r13`	R6	~
~	R7	`move r13, 0xFFFFFF3C`	R7	~
~	R8	`SWI 0x143C`	R8	~
~	R9	⟶	R9	~
~	R10		R10	~
~	R11		R11	~
~	R12		R12	~
≡	R13		R13	≡
~	R14		R14	~
~	R15		R15	*next instruction*

C-2.11.5 Ipc

$\mathit{SendTarget}_{\text{IN}}$	R0		R0	$\mathit{Result}_{\text{OUT}}$
$\mathit{ReceiveTarget}_{\text{IN}}$	R1		R1	~
~	R2		R2	~
Tag_{IN}	R3		R3	$\mathit{Tag}_{\text{OUT}}$
$\mathit{MessageData}_{1\,\text{IN}}$	R4		R4	$\mathit{MessageData}_{1\,\text{OUT}}$
$\mathit{MessageData}_{2\,\text{IN}}$	R5		R5	$\mathit{MessageData}_{2\,\text{OUT}}$
$\mathit{MessageData}_{3\,\text{IN}}$	R6	`move r12, r13`	R6	$\mathit{MessageData}_{3\,\text{OUT}}$
$\mathit{MessageData}_{4\,\text{IN}}$	R7	`move r13, 0xFFFFFF00`	R7	$\mathit{MessageData}_{4\,\text{OUT}}$
$\mathit{MessageData}_{5\,\text{IN}}$	R8	`SWI 0x1400`	R8	$\mathit{MessageData}_{5\,\text{OUT}}$
~	R9	⟶	R9	~
~	R10		R10	~
~	R11		R11	~
~	R12		R12	~
≡	R13		R13	≡
~	R14		R14	~
~	R15		R15	*next instruction*

C-2.11.6 MapControl

$\mathit{Target}_{\text{IN}}$	R0		R0	$\mathit{Result}_{\text{OUT}}$
$\mathit{Control}_{\text{IN}}$	R1		R1	~
~	R2		R2	~
~	R3		R3	~
~	R4		R4	~
~	R5		R5	~
~	R6	`move r12, r13`	R6	~
~	R7	`move r13, 0xFFFFFF14`	R7	~
~	R8	`SWI 0x1414`	R8	~
~	R9	⟶	R9	~
~	R10		R10	~
~	R11		R11	~
~	R12		R12	~
≡	R13		R13	≡
~	R14		R14	~
~	R15		R15	*next instruction*

C-2.11.7 MemoryCopy

$Remote_{\mathrm{IN}}$	`R0`		`R0`	$Result_{\mathrm{OUT}}$
$Local_{\mathrm{IN}}$	`R1`		`R1`	~
$Size_{\mathrm{IN}}$	`R2`		`R2`	~
$Direction_{\mathrm{IN}}$	`R3`		`R3`	~
~	`R4`		`R4`	~
~	`R5`		`R5`	~
~	`R6`	`move r12, r13`	`R6`	~
~	`R7`	`move r13, 0xFFFFFF44`	`R7`	~
~	`R8`	`SWI 0x1444`	`R8`	~
~	`R9`	⟶	`R9`	~
~	`R10`		`R10`	~
~	`R11`		`R11`	~
~	`R12`		`R12`	~
≡	`R13`		`R13`	≡
~	`R14`		`R14`	~
~	`R15`		`R15`	*next instruction*

C-2.11.8 Mutex

$Target_{\mathrm{IN}}$	`R0`		`R0`	$Result_{\mathrm{OUT}}$
$Control_{\mathrm{IN}}$	`R1`		`R1`	~
~	`R2`		`R2`	~
~	`R3`		`R3`	~
~	`R4`		`R4`	~
~	`R5`		`R5`	~
~	`R6`	`move r12, r13`	`R6`	~
~	`R7`	`move r13, 0xFFFFFF34`	`R7`	~
~	`R8`	`SWI 0x1434`	`R8`	~
~	`R9`	⟶	`R9`	~
~	`R10`		`R10`	~
~	`R11`		`R11`	~
~	`R12`		`R12`	~
≡	`R13`		`R13`	≡
~	`R14`		`R14`	~
~	`R15`		`R15`	*next instruction*

C-2.11.9 MutexControl

Target$_{\text{IN}}$	R0		R0	*Result*$_{\text{OUT}}$
Control$_{\text{IN}}$	R1		R1	~
~	R2		R2	~
~	R3		R3	~
~	R4		R4	~
~	R5		R5	~
~	R6	`move r12, r13`	R6	~
~	R7	`move r13, 0xFFFFFF38`	R7	~
~	R8	`SWI 0x1438`	R8	~
~	R9	→	R9	~
~	R10		R10	~
~	R11		R11	~
~	R12		R12	~
≡	R13		R13	≡
~	R14		R14	~
~	R15		R15	*next instruction*

C-2.11.10 PlatformControl

Target$_{\text{IN}}$	R0		R0	*Result*$_{\text{OUT}}$
Param1$_{\text{IN}}$	R1		R1	~
Param2$_{\text{IN}}$	R2		R2	~
Param3$_{\text{IN}}$	R3		R3	~
~	R4		R4	~
~	R5		R5	~
~	R6	`move r12, r13`	R6	~
~	R7	`move r13, 0xFFFFFF2C`	R7	~
~	R8	`SWI 0x142C`	R8	~
~	R9	→	R9	~
~	R10		R10	~
~	R11		R11	~
~	R12		R12	~
≡	R13		R13	≡
~	R14		R14	~
~	R15		R15	*next instruction*

C-2.11.11 Schedule

Target$_{\text{IN}}$	`R0`		`R0`	*Result*$_{\text{OUT}}$
Slice$_{\text{IN}}$	`R1`		`R1`	*Slice*$_{\text{OUT}}$
HwThreadMask$_{\text{IN}}$	`R2`		`R2`	~
Domain$_{\text{IN}}$	`R3`		`R3`	~
Priority$_{\text{IN}}$	`R4`		`R4`	~
Reserved$_{\text{IN}}$	`R5`		`R5`	~
~	`R6`	`move r12, r13`	`R6`	~
~	`R7`	`move r13, 0xFFFFFF10`	`R7`	~
~	`R8`	`SWI 0x1410`	`R8`	~
~	`R9`	⟶	`R9`	~
~	`R10`		`R10`	~
~	`R11`		`R11`	~
~	`R12`		`R12`	~
≡	`R13`		`R13`	≡
~	`R14`		`R14`	~
~	`R15`		`R15`	*next instruction*

C-2.11.12 SpaceControl

Target$_{\text{IN}}$	`R0`		`R0`	*Result*$_{\text{OUT}}$
Control$_{\text{IN}}$	`R1`		`R1`	*SpaceResources*$_{\text{OUT}}$
Utcb$_{\text{IN}}$	`R2`		`R2`	~
Resources$_{\text{IN}}$	`R3`		`R3`	~
~	`R4`		`R4`	~
~	`R5`		`R5`	~
~	`R6`	`move r12, r13`	`R6`	~
~	`R7`	`move r13, 0xFFFFFF18`	`R7`	~
~	`R8`	`SWI 0x1418`	`R8`	~
~	`R9`	⟶	`R9`	~
~	`R10`		`R10`	~
~	`R11`		`R11`	~
~	`R12`		`R12`	~
≡	`R13`		`R13`	≡
~	`R14`		`R14`	~
~	`R15`		`R15`	*next instruction*

C-2.11.13 ThreadControl

Target$_{\text{IN}}$	R0		R0	*Result*$_{\text{OUT}}$
Space$_{\text{IN}}$	R1		R1	~
Dummy$_{\text{IN}}$	R2		R2	~
Pager$_{\text{IN}}$	R3		R3	~
ExceptionHandler$_{\text{IN}}$	R4		R4	~
ThreadResources$_{\text{IN}}$	R5		R5	~
Utcb$_{\text{IN}}$	R6	`move r12, r13`	R6	~
~	R7	`move r13, 0xFFFFFF08`	R7	~
~	R8	`SWI 0x1408`	R8	~
~	R9	⟶	R9	~
~	R10		R10	~
~	R11		R11	~
~	R12		R12	~
≡	R13		R13	≡
~	R14		R14	~
~	R15		R15	*next instruction*

C-2.11.14 ThreadSwitch

Target$_{\text{IN}}$	R0		R0	~
~	R1		R1	~
~	R2		R2	~
~	R3		R3	~
~	R4		R4	~
~	R5		R5	~
~	R6		R6	~
~	R7	`move r12, r13`	R7	~
~	R8	`move r13, 0xFFFFFF04`	R8	~
~	R8	`SWI 0x1404` ⟶	R8	~
~	R9		R9	~
~	R10		R10	~
~	R11		R11	~
~	R12		R12	~
≡	R13		R13	≡
~	R14		R14	~
~	R15		R15	*next instruction*

C-2.12 ExchangeRegisters Register Access

The ArmRegsInMRs Format:

Flags	MessageData$_{16}$
R15	MessageData$_{15}$
R14	MessageData$_{14}$
R13	MessageData$_{13}$
R12	MessageData$_{12}$
R11	MessageData$_{11}$
R10	MessageData$_{10}$
R9	MessageData$_{9}$
R8	MessageData$_{8}$
R7	MessageData$_{7}$
R4	MessageData$_{6}$
R6	MessageData$_{5}$
R5	MessageData$_{4}$
R3	MessageData$_{3}$
R2	MessageData$_{2}$
R1	MessageData$_{1}$
R0	MessageData$_{0}$

The ArmRegsInMRs format is used when inspecting or modifying the state of a thread's registers using the RegControl.*Rget* and RegControl.*Rset* flags. The using the Rset flag, the Flags field is interpreted as specified in Section C-2.5, however the `CPSR.T` bit is derived from the least-significant-bit of the `R15` field.

PART D

Language Bindings

D-1 Overview

This chapter describes the set of combined C/C++ language bindings to the OKL4 API.

The remainder of this chapter is organised as follows:

- Section D-1.1 discusses the common variable naming conventions applied by the language bindings to the OKL4 API. This section is largely informative and may be skipped by most readers.
- Section D-1.2 outlines the configuration of the programming environment used to compile OKL4 programs written in the C programming environment.
- Section D-1.3 describes the list of C headers included in the language bindings library.

The content of each header is listed in Section D-1.3 and an alphabetical list of all macros, C types and functions defined in the language bindings library is provided in Chapter D-17.

D-1.1 Naming Conventions

The language bindings library uses the following naming conventions for all C entities that are documented in this part of the manual:

headers All headers are assigned names beginning with `l4/` and ending in `.h`. They are intended to be included using pre-processing directives of the form `#include <l4/`*header-name*`.h>`.

macros All C language binding elements defined as macros in this manual have names with the prefix `L4_` followed by one or more words in upper-case letters and separated by underscores. For example: `L4_BIG_ENDIAN`. Note that, other C language bindings may also be implemented as macros and may follow different naming conventions as appropriate.

types All C types defined in this chapter are given names that begin with `L4_` and end in `_t`. Individual words within the name are generally capitalised with no additional underscores appearing in the name. For example: `L4_ThreadId_t`.

functions and variables All other entities defined in this library are given names with the prefix `L4_`. As with type names, individual words within the name are usually capitalised and no further underscores appear in the name.

These capitalisation rules are violated by a few C entities defined in this chapter for historic reasons. Note that all types, constants and functions defined in this part of the manual may be implemented as pre-processing macros expanding to suitable C types or expressions. The library guarantees that all functions defined as part of the language bindings library should be valid immutable lvalues. That is, the user may obtain their address using the `&` operator. It is unspecified which, if any such addresses are equivalent to each other.

Header names of the form `<l4/`*header-name*`.h>` are implicitly reserved for use by the C language bindings to the OKL4 API. Applications relying on these bindings should not define headers of the form `#include <l4/`*header-name*`.h>` that may be included using pre-processing directives. Also, headers of the above form should not be included in source files except as defined in this chapter.

All identifiers with the prefixes `L4_`, `_L4_` and `__L4_` are reserved for use by the C language binding library, both as macro names and ordinary identifiers with any linkage. In addition, applications should not use any identifiers with a reserved prefix except as defined in this chapter. As any identifier with a reserved prefix may be defined as a macro name by the library, applications cannot use these identifiers in any scope, including as function prototypes and structure definitions.

D-1.2 Programming Environment Configuration

All programs using the OKL4 C language bindings must be linked against the `l4` language library, usually by including the `-ll4` option in the compiler's command line. It must also ensure that the directory containing the `l4` library is included in the library search path, and that the directory containing the `l4` header directory is included in the system header search path, typically using the `-I` option on the compiler's command line. The `l4` header directory itself should not be included in the header search path.

When using language bindings, all ThreadWord virtual registers are reserved for internal use by the OKL4 bindings library. Applications should not modify these registers, except indirectly through the interface elements defined in this chapter.

D-1.2.1 `__L4_Init()`

```
        extern "C" void __L4_Init(void);
```

The user must ensure that `__L4_Init()` is invoked prior to using the functions defined in the following chapters. This is normally arranged by the linker. On operating system personalities that do not support this functionality, it is possible to call `__L4_Init()` directly from the `main()` function of the application. Redundant calls to `__L4_Init()` are permitted and any subsequent invocation of `__L4_Init()` will have no effect. `__L4_Init()` is not defined in any header provided by the library. The user may invoke it directly using the above function prototype.

D-1.3 Header Files

The C bindings to the OKL4 API are contained in the following system headers:

`<l4/cache.h>` Provides the C definitions that are related to the OKL4 CacheControl system call described in Section A-12.1. The content of this header is defined in Chapter D-2.

`<l4/caps.h>` The `<l4/cap.h>` header defines the functions that provide access to the CapControl system call.

`<l4/ipc.h>` Provides the C definitions related to the OKL4 Ipc system call described in Section A-12.5 as well as the general OKL4 IPC mechanism discussed in Chapter A-6. The content of this header is defined in Chapter D-6.

`<l4/interrupt.h>` Provides the C definitions that are related to the InterruptControl system call described in Section A-12.4. The content of this header is defined in Chapter D-5.

`<l4/map.h>` Provides the C definitions related to the MapControl system call described in Section A-12.6. The content of this header is defined in Chapter D-7.

`<l4/message.h>` Provides a number of C function definitions useful for common message construction tasks. The content of this header is defined in Chapter D-8.

`<l4/misc.h>` Provides a number of miscellaneous C definitions. The content of this header is defined in Chapter D-16.

`<l4/schedule.h>` Provides the C definitions related to the Schedule system call described in Section A-12.11 and the ThreadSwitch system call described in Section A-12.15. The content of this header is defined in Chapter D-10.

`<l4/space.h>` Provides the C definitions related to the SpaceControl system call described in Section A-12.12. The content of this header is defined in Chapter D-11.

`<l4/thread.h>` Provides the C definitions related to the ThreadControl system call described in Section A-12.14. The content of this header is defined in Chapter D-13.

`<l4/types.h>` Provides miscellaneous C definitions used in the language bindings library. This header is automatically included by all headers listed above. The content of this header is defined in Chapter D-14.

D-2 The `<l4/cache.h>` Header

The `<l4/cache.h>` header contains the definitions relating to the CacheControl system call. Note that this header automatically includes the `<l4/types.h>` header, making available all types and constants defined in Chapter D-14.

D-2.1 System Calls

The `<l4/cache.h>` header defines the functions that provide access to the CacheControl system call.

D-2.1.1 `L4_CacheFlushAll()`

```
L4_Word_t L4_CacheFlushAll(void);
```

`L4_CacheFlushAll()` performs the *clean* and *invalidate* cache operations on the entire domain of each address space in the system using an appropriate invocation of the CacheControl system call. It returns the value of the $Result_{\text{OUT}}$ parameter from the CacheControl system call. A zero value indicates an error, all other values indicate the successful completion of the requested microkernel operation.

See also: the CacheControl system call on page 78 and the `L4_Word_t` type on page 234.

D-2.1.2 `L4_CacheFlushRange()`

```
L4_Word_t L4_CacheFlushRange(
        L4_SpaceId_t  space,
        L4_Word_t     start,
        L4_Word_t     end);
```

The `L4_CacheFlushRange(`*space*`, `*start*`, `*end*`)` function performs the *clean* and *invalidate* operations on all caches in the system for the region covering all bytes in the range between *start* and *end* − 1, inclusive, of the address space specified by the *space* argument. It is implemented using an appropriate invocation of the CacheControl system call and returns the value of the $Result_{\text{OUT}}$ parameter from the CacheControl system call. A zero value indicates an error, all other values indicate the successful completion of the requested microkernel operation.

See also: the CacheControl system call on page 78, the **SpaceId** data type on page 64,
the `L4_SpaceId_t` type on page 241 and the `L4_Word_t` type on page 234.

D-2.1.3 L4_CacheFlushIRange()

```
L4_Word_t L4_CacheFlushIRange(
        L4_SpaceId_t  space,
        L4_Word_t     start,
        L4_Word_t     end);
```

The L4_CacheFlushIRange(*space*, *start*, *end*) function performs the *invalidate* cache operation on all instruction caches in the system for the region covering all bytes in the range between *start* and *end* − 1, inclusive, of the address space specified by the *space* argument. It is implemented using an appropriate invocation of the CacheControl system call and returns the value of the $Result_{\text{OUT}}$ parameter from the CacheControl system call. A zero value indicates an error, all other values indicate the successful completion of the requested microkernel operation.

See also: the CacheControl system call on page 78, the **SpaceId** data type on page 64,
the L4_SpaceId_t type on page 241 and the L4_Word_t type on page 234.

D-2.1.4 L4_CacheFlushDRange()

```
L4_Word_t L4_CacheFlushDRange(
        L4_SpaceId_t  space,
        L4_Word_t     start,
        L4_Word_t     end);
```

L4_CacheFlushDRange(*space*, *start*, *end*) performs the *clean* and *invalidate* cache operations on all data caches in the system for the region covering all bytes in the range between *start* and *end* − 1, inclusive, of the address space specified by the *space* argument. It is implemented using an appropriate invocation of the CacheControl system call and returns the value of the $Result_{\text{OUT}}$ parameter from the CacheControl system call. A zero value indicates an error, all other values indicate the successful completion of the requested microkernel operation.

See also: the CacheControl system call on page 78, the **SpaceId** data type on page 64,
the L4_SpaceId_t type on page 241 and the L4_Word_t type on page 234.

D-2.1.5 L4_CacheFlushRangeInvalidate()

```
L4_Word_t L4_CacheFlushRangeInvalidate(
        L4_SpaceId_t  space,
        L4_Word_t     start,
        L4_Word_t     end);
```

The L4_CacheFlushRangeInvalidate(*space*, *start*, *end*) function performs the *invalidate* cache operation on all caches in the system for the region covering all bytes in the range between *start* and *end* − 1, inclusive, of the address space specified by the *space* argument. It is implemented using an appropriate invocation of the CacheControl system call and returns the value of the $Result_{\text{OUT}}$ parameter from the CacheControl system call. A zero value indicates an error, all other values indicate the successful completion of the requested microkernel operation.

See also: the CacheControl system call on page 78, the **SpaceId** data type on page 64,
the L4_SpaceId_t type on page 241 and the L4_Word_t type on page 234.

D-2.2 The CACHECONTROL Data Type

The `<l4/cache.h>` header defines the macros used to construct values of the OKL4 **CACHECONTROL** data type. Values of this type are supplied as the *control* argument to the `L4_CacheControl(`*target*`,` *control*`)` function. Valid values of this argument should be constructed by performing an inclusive-OR of an appropriate cache operation macro with one or more cache level macros and an integer representing the number of cache regions being supplied to CacheControl in the MessageData registers.

D-2.2.1 Cache Operations

The `<l4/cache.h>` header defines the macros that provide symbolic names for all valid values of the *Op* field of the OKL4 **CACHECONTROL** data type. These macros may be used for the direct construction of a **CACHECONTROL** value using the inclusive-OR (|) operator without any shift (<< or >>) operations.

`L4_CacheCtl_ArchSpecific`	Architecture specific cache control function.
`L4_CacheCtl_FlushRanges`	Flush specific cache regions.
`L4_CacheCtl_FlushIcache`	Flush the entire instruction cache.
`L4_CacheCtl_FlushDcache`	Flush the entire data cache.
`L4_CacheCtl_FlushCache`	Flush all instruction and data caches.
`L4_CacheCtl_CacheLock`	Lock a region in the cache.
`L4_CacheCtl_CacheUnLock`	Unlock a region in the cache.

See also: the CacheControl system call on page 78, the **CACHECONTROL** data type on page 50 and the **CACHEOP** data type on page 50.

D-2.2.2 Cache Levels

The `<l4/cache.h>` header defines the macros that provide symbolic names for the *Li* flags of the OKL4 **CACHECONTROL** data type. In all cases, these macros are suitable for direct construction of a **CACHECONTROL** value using the inclusive-OR (|) operator without any shift (<< or >>) operations.

`L4_CacheCtl_MaskL1`	The cache operation should be performed on an L1 cache.
`L4_CacheCtl_MaskL2`	The cache operation should be performed on an L2 cache.
`L4_CacheCtl_MaskL3`	The cache operation should be performed on an L3 cache.
`L4_CacheCtl_MaskL4`	The cache operation should be performed on an L4 cache.
`L4_CacheCtl_MaskL5`	The cache operation should be performed on an L5 cache.
`L4_CacheCtl_MaskL6`	The cache operation should be performed on an L6 cache.
`L4_CacheCtl_MaskAllLs`	The cache operation should be performed on all levels of caches in the system.

See also: the CacheControl system call on page 78 and the **CACHECONTROL** data type on page 50.

D-2.3 Cache Region Operations

The `<l4/cache.h>` header defines the macros used to construct values of the OKL4 **CACHEREGIONOP** data type. Values of this type are supplied to the CacheControl system call as the $RegionOp_{i_{IN}}$ parameter in the MessageData registers. Valid values of these parameters should be constructed by performing an inclusive-OR of one or more macros defined below with the size of the address space region on which the operation is performed.

`L4_CacheFlush_AttrI` Corresponds to the *IC* field in the CACHEREGIONOP data constructor, indicating the caching operation should cover the system instruction caches.

`L4_CacheFlush_AttrD` Corresponds to the *DC* field in the CACHEREGIONOP data constructor, indicating the caching operation should cover the system data caches.

`L4_CacheFlush_AttrClean` Corresponds to the *C* field in the CACHEREGIONOP data constructor, indicating that the *clean* cache operation should be performed on the specified caches.

`L4_CacheFlush_AttrInvalidate` Corresponds to the *I* field in the CACHEREGIONOP data constructor, indicating that the *invalidate* cache operation should be performed on the specified caches.

`L4_CacheFlush_AttrAll` Represents an inclusive-OR of all four macros defined above.

See also: the CacheControl system call on page 78 and the **CACHEREGIONOP** data type on page 51.

D-3 The <l4/caps.h> Header

D-3.1 System Calls

The <l4/cap.h> header defines the functions that provide access to the CapControl system call.

D-3.1.1 L4_CreateClist()

```
L4_Word_t L4_CreateClist(
        L4_ClistId_t    clist,
        L4_Word_t       entries);
```

The L4_CreateClist() function is used to create a new capability list with a clist ID specified by the *clist* argument. The size of the clist to be created is specified by the *entries* argument. This function returns 1 on successful completion and 0 on failure.

See also: the CapControl system call on page 80 and the L4_Word_t type on page 234.

D-3.1.2 L4_DeleteClist()

```
L4_Word_t L4_DeleteClist(
        L4_ClistId_t    clist);
```

The L4_DeleteClist() function is used to delete the capability list specified by the *clist* argument. Note that the specified capability list must be empty. This function returns 1 on successful completion and 0 on failure.

See also: the CapControl system call on page 80 and the L4_Word_t type on page 234.

D-3.1.3 L4_CreateIpcCap()

```
L4_Word_t L4_CreateIpcCap(
        L4_ThreadId_t  tid,
        L4_ClistId_t   tid_clist,
        L4_ThreadId_t  dest,
        L4_ClistId_t   dest_clist);
```

The L4_CreateIpcCap() function is used to copy the capability located at the index specified by *tid* in the capability list specified by the *tid_clist* argument into the capability list specified by the *dest_clist* argument at the index specified by the *dest* argument. This function returns 1 on successful completion and 0 on failure.

See also: the CapControl system call on page 80 and the L4_Word_t type on page 234.

D-3.1.4 L4_DeleteCap()

```
        L4_Word_t L4_DeleteCap(
                L4_ClistId_t    clist,
                L4_ThreadId_t   tid);
```

The L4_DeleteCap() function is used to delete the capability specified by the *tid* argument from the capability list specified by the *clist* argument. This function returns 1 on successful completion and 0 on failure.

See also: the CapControl system call on page 80 and the L4_Word_t type on page 234.

D-4 The <l4/config.h> Header

The <l4/config.h> header provides access to the L4 configuration information known at compile-time (either because these are static for the architecture being built, or that the configuration parameter is provided to the build system). It automatically includes all definitions provided by the <l4/types.h> header described in Chapter D-14.

D-4.1 Accessor Functions

The <l4/config.h> provides accessor functions to all configuration information provided.

D-4.1.1 L4_GetMessageRegisters()

```
L4_Word_t L4_LoadMR(void);
```

The L4_GetMessageRegisters() function provides the number of Message Registers available for IPC calls.

See also: the MessageData virtual register on page 73, the L4_LoadMR() function on page 247,
the L4_StoreMR() function on page 247 and the L4_Word_t type on page 234.

D-4.1.2 L4_GetUtcbSize()

```
L4_Word_t L4_GetUtcbSize(void);
```

The L4_StoreMR() function returns the size in bytes of an individual Utcb record.

See also: the L4_Word_t type on page 234.

D-4.1.3 L4_GetUtcbBits()

```
L4_Word_t L4_GetUtcbBits(void);
```

The L4_GetUtcbBits() function returns the number of bits used to specify the size of the Utcb, that is Log$_2$(L4$_G$etUtcbSize) with integer truncation.

See also: the L4_Word_t type on page 234.

D-4.1.4 L4_GetUtcbAreaSize()

```
L4_Word_t L4_GetUtcbAreaSize(void);
```

The L4_GetUtcbAreaSize() function returns the memory size used for storing Utcbs in bytes.

See also: the L4_Word_t type on page 234.

D-4.1.5 L4_GetPageMask()

```
L4_Word_t L4_GetPageMask(void);
```

The L4_Get_NotifyBits() function returns a bitmask of the valid page sizes available to user applications.

See also: the L4_Word_t type on page 234.

D-4.1.6 L4_GetPagePerms()

```
L4_Word_t L4_GetPagePerms(void);
```

The L4_GetPagePerms() function returns bitmask of the valid page permissions available to user applications.

See also: the L4_Word_t type on page 234.

D-4.1.7 L4_GetMinPageBits()

```
L4_Word_t L4_GetMinPageBits(void);
```

The L4_GetMinPageBits() function returns the Log$_2$ of the minimum page size supported by the kernel.

See also: the L4_Word_t type on page 234.

D-4.1.8 L4_UtcbIsKernelManaged()

```
int L4_UtcbIsKernelManged(void);
```

The L4_UtcbIsKernelManaged() function is used to determine whether the calling program specifies where the location of the mapping of the Utcb area in its address space or whether the location is determined by the kernel. It should be noted that the kernel determines the location of the mapping of the Utcb area on all architectures which do not provide hardware support for *Thread Local Storage* (TLS).

See also: the L4_Word_t type on page 234.

D-4.1.9 `L4_ApiVersion()`

```
L4_Word_t L4_ApiVersion(void);
```

The `L4_ApiVersion()` function returns the API version of the kernel.

See also: the `L4_Word_t` type on page 234.

D-5 The <l4/interrupt.h> Header

D-5.1 System Calls

The `<l4/interrupt.h>` header defines the functions that provide access to the InterruptControl system call.

D-5.1.1 `L4_RegisterInterrupt()`

```
L4_Word_t L4_RegisterInterrupt(
    L4_ThreadId_t    target,
    L4_Word_t        notify_bit,
    L4_Word_t        mrs,
    L4_Word_t        request
)
```

The `L4_RegisterInterrupt()` function registers a thread to handle the specified hardware interrupts. An asynchronous IPC message will be sent to the *Target*$_{\text{IN}}$ thread, with the *NotifyBit*$_{\text{IN}}$ set.

Input Parameter:	C Function Argument:
Target$_{\text{IN}}$	*target*
NotifyBit$_{\text{IN}}$	*notify_bit*
MessageRegisters$_{\text{IN}}$	*mrs*
Request$_{\text{IN}}$	*request*

The interrupt descriptor(s) are specified using message registers. The *MessageRegisters*$_{\text{IN}}$ argument is the last message register that contains interrupt information.

See also: the InterruptControl system call on page 85, the **CapId** data type on page 52,
the `L4_ThreadId_t` type on page 242 and the `L4_Word_t` type on page 234.

D-5.1.2 `L4_UnregisterInterrupt()`

```
L4_Word_t L4_UnregisterInterrupt(
    L4_ThreadId_t    target,
    L4_Word_t        mrs,
    L4_Word_t        request
)
```

The `L4_UnregisterInterrupt()` function stops a thread from handling the specified hardware interrupts.

The interrupt descriptor(s) are specified using message registers. The *MessageRegisters*$_{\text{IN}}$ argument is the last message register that contains interrupt information.

Input Parameter:	C Function Argument:
Target$_{\text{IN}}$	*target*
MessageRegisters$_{\text{IN}}$	*mrs*
Request$_{\text{IN}}$	*request*

See also: the InterruptControl system call on page 85, the **CapId** data type on page 52, the `L4_ThreadId_t` type on page 242 and the `L4_Word_t` type on page 234.

D-5.1.3 `L4_AcknowledgeInterrupt()`

```
L4_Word_t L4_AcknowledgeInterrupt(
    L4_Word_t        mrs,
    L4_Word_t        request
)
```

The `L4_AcknowledgeInterrupt()` function requests the kernel to acknowledge and clear the specified hardware interrupts.

The interrupt descriptor(s) are specified using message registers. The *MessageRegisters*$_{\text{IN}}$ argument is the last message register that contains interrupt information.

Input Parameter:	C Function Argument:
MessageRegisters$_{\text{IN}}$	*mrs*
Request$_{\text{IN}}$	*request*

See also: the InterruptControl system call on page 85 and the `L4_Word_t` type on page 234.

D-5.1.4 `L4_AcknowledgeInterruptOnBehalf()`

```
L4_Word_t L4_AcknowledgeInterruptOnBehalf(
    L4_ThreadId_t    on_behalf,
    L4_Word_t        mrs,
    L4_Word_t        request
)
```

The `L4_AcknowledgeInterruptOnBehalf()` function requests the kernel to acknowledge and clear the specified hardware interrupts. The kernel treats the request as coming from the *OnBehalf*$_{\text{IN}}$ thread.

The interrupt descriptor(s) are specified using message registers. The *MessageRegisters*$_{IN}$ argument is the last message register that contains interrupt information.

Input Parameter:	C Function Argument:
OnBehalf$_{IN}$	*on_behalf*
MessageRegisters$_{IN}$	*mrs*
Request$_{IN}$	*request*

See also: the InterruptControl system call on page 85 and the `L4_Word_t` type on page 234.

D-5.1.5 `L4_AcknowledgeWaitInterrupt()`

```
L4_Word_t L4_AcknowledgeWaitInterrupt(
    L4_Word_t        mrs,
    L4_Word_t        request
)
```

The `L4_AcknowledgeWaitInterrupt()` function requests the kernel to acknowledge and clear the specified hardware interrupts. The calling thread will be blocked waiting for the next asynchonous IPC message.

The interrupt descriptor(s) are specified using message registers. The *MessageRegisters*$_{IN}$ argument is the last message register that contains interrupt information.

Input Parameter:	C Function Argument:
MessageRegisters$_{IN}$	*mrs*
Request$_{IN}$	*request*

See also: the InterruptControl system call on page 85 and the `L4_Word_t` type on page 234.

D-6 The <l4/ipc.h> Header

The <l4/ipc.h> header provides access to the Ipc system call. It automatically includes all definitions provided by the <l4/types.h>, <l4/thread.h> and <l4/message.h> headers described in Chapters D-14, D-13 and D-8.

D-6.1 System Calls

The <l4/ipc.h> header defines a set of functions that provide access to the Ipc system call.

D-6.1.1 L4_Call()

```
L4_MsgTag_t L4_Call(
        L4_ThreadId_t target);
```

L4_Call(*target*) performs an IPC *call* operation, consisting of a *send* operation to the thread specified by *target*, followed by a *receive* operation from the *target* thread. It is implemented using an appropriate invocation of the Ipc system call. The content of the message is obtained from the MessageData registers of the caller, which should be modified using the L4_LoadMR() or L4_LoadMRs() function immediately prior to the invocation of L4_Call(). The received message is delivered in the MessageData registers of the caller and may be accessed using the L4_StoreMR() or L4_StoreMRs() functions immediately following the invocation of L4_Call(). The L4_Call() function returns the value of the Tag_{OUT} parameter from the Ipc system call.

See also: the Ipc system call on page 86, the **CapId** data type on page 52,
the L4_ThreadId_t type on page 242, the **MessageTag** data type on page 59,
the L4_MsgTag_t type on page 193, the MessageData virtual register on page 73,
the L4_LoadMR() function on page 247, the L4_LoadMRs() function on page 247,
the L4_StoreMR() function on page 247, the L4_StoreMRs() function on page 248,
section A-6.3 on page 42 and section A-6.5 on page 43.

D-6.1.2 L4_Lcall()

```
L4_MsgTag_t L4_Lcall(
        L4_ThreadId_t target);
```

The Lcall() function is an alias for the L4_Call() function. It is included in the language bindings for historical reasons and should not be used by new programs.

See also: the L4_Call() function on page 179.

D-6.1.3 L4_Send()

```
L4_MsgTag_t L4_Send(
        L4_ThreadId_t target);
```

L4_Send(*target*) performs an IPC *send* operation to the thread specified by *target*. It is implemented using an appropriate invocation of the Ipc system call. The content of the message is obtained from the MessageData registers of the caller. These should be modified using the L4_LoadMR() or L4_LoadMRs() function immediately prior to the invocation of L4_Send(). The L4_Send() function returns the value of the Tag_{OUT} parameter from the Ipc system call.

See also: the Ipc system call on page 86, the **CapId** data type on page 52,
the L4_ThreadId_t type on page 242, the **MessageTag** data type on page 59,
the L4_MsgTag_t type on page 193, the MessageData virtual register on page 73,
the L4_LoadMR() function on page 247, the L4_LoadMRs() function on page 247 and
section A-6.3 on page 42.

D-6.1.4 L4_Reply()

```
L4_MsgTag_t L4_Reply(
        L4_ThreadId_t target);
```

L4_Reply(*target*) performs an IPC *reply* operation to the thread specified by the *target* argument. It is implemented using an appropriate invocation of the Ipc system call. The content of the message is obtained from the MessageData registers of the caller. These registers should be modified using the L4_LoadMR() or L4_LoadMRs() function immediately prior to the invocation of L4_Reply(). The L4_Reply() function returns the value of the Tag_{OUT} parameter from the Ipc system call.

See also: the Ipc system call on page 86, the **CapId** data type on page 52,
the L4_ThreadId_t type on page 242, the **MessageTag** data type on page 59,
the L4_MsgTag_t type on page 193, the MessageData virtual register on page 73,
the L4_LoadMR() function on page 247, the L4_LoadMRs() function on page 247 and
section A-6.3 on page 42.

D-6.1.5 L4_Send_Nonblocking()

```
L4_MsgTag_t L4_Send_Nonblocking(
        L4_ThreadId_t target);
```

The L4_Send_Nonblocking() function is an alias of the L4_Reply() function defined above. It is included in the language bindings for historical reasons and should not be used by new programs.

See also: the L4_Reply() function on page 180.

D-6.1.6 L4_Receive()

```
L4_MsgTag_t L4_Receive(
        L4_ThreadId_t target);
```

The L4_Receive(*target*) function performs an IPC *wait* operation from the thread specified by the *target* argument. It is implemented using an appropriate invocation of the Ipc system call. The content of the message is returned in the MessageData registers of the caller and may be accessed using the L4_StoreMR() or L4_StoreMRs() function immediately following the invocation of this function. L4_Receive() returns the value of the Tag_{OUT} parameter from the Ipc system call.

See also: the Ipc system call on page 86, the **CapId** data type on page 52,
the L4_ThreadId_t type on page 242, the **MessageTag** data type on page 59,
the L4_MsgTag_t type on page 193, the MessageData virtual register on page 73,
the L4_StoreMR() function on page 247, the L4_StoreMRs() function on page 248 and
section A-6.5 on page 43.

D-6.1.7 L4_Receive_Nonblocking()

```
L4_MsgTag_t L4_Receive_Nonblocking(
        L4_ThreadId_t target);
```

The L4_Receive_Nonblocking(*target*) function performs an IPC *receive* operation from the thread specified by the *target* argument. It is implemented using an appropriate invocation of the Ipc system call. The content of the message is returned in the MessageData registers of the caller and may be accessed using the L4_StoreMR() or L4_StoreMRs() functions immediately following the invocation of this function. L4_Receive_Nonblocking() returns the value of the Tag_{OUT} parameter from the Ipc system call.

See also: the Ipc system call on page 86, the **CapId** data type on page 52,
the L4_ThreadId_t type on page 242, the **MessageTag** data type on page 59,
the L4_MsgTag_t type on page 193, the MessageData virtual register on page 73,
the L4_StoreMR() function on page 247, the L4_StoreMRs() function on page 248 and
section A-6.5 on page 43.

D-6.1.8 L4_Wait()

```
L4_MsgTag_t L4_Wait(
        L4_ThreadId_t* target_ptr);
```

L4_Wait(*result_ptr*) performs an IPC *wait* operation from any OKL4 thread. It is implemented using an appropriate invocation of the Ipc system call. The content of the message is returned in MessageData registers of the caller and may be accessed using the L4_StoreMR() or L4_StoreMRs() function immediately following the invocation of L4_Wait(). It returns the value of the Tag_{OUT} parameter from the Ipc system call. Upon successful completion, L4_Wait(*result_ptr*) stores the thread ID of the apparent sender of the message in the variable pointed to by the *result_ptr* argument.

See also: the Ipc system call on page 86, the **CapId** data type on page 52,
the L4_ThreadId_t type on page 242, the **MessageTag** data type on page 59,
the L4_MsgTag_t type on page 193, the MessageData virtual register on page 73,
the L4_StoreMR() function on page 247, the L4_StoreMRs() function on page 248 and
section A-6.5 on page 43.

D-6.1.9 `L4_ReplyWait()`

```
L4_MsgTag_t L4_ReplyWait(
        L4_ThreadId_t  target,
        L4_ThreadId_t* result_ptr);
```

The `L4_ReplyWait(`*target*`,` *result_ptr*`)` function performs an IPC *reply* operation to the thread specified by *target*, followed by an IPC *wait* operation from any OKL4 thread. It is implemented using an appropriate invocation of the Ipc system call. The content of the message is obtained from the MessageData registers of the caller, which may be modified using the `L4_LoadMR()` or `L4_LoadMRs()` function immediately prior to the invocation of `L4_ReplyWait()`. The received message is delivered in the MessageData registers of the caller and may be accessed using the `L4_StoreMR()` or `L4_StoreMRs()` function immediately following the invocation of `L4_ReplyWait()`. `L4_ReplyWait()` returns the value of the $\mathit{Tag}_{\text{OUT}}$ parameter from the Ipc system call.

See also: the Ipc system call on page 86, the **CapId** data type on page 52,
the `L4_ThreadId_t` type on page 242, the **MessageTag** data type on page 59,
the `L4_MsgTag_t` type on page 193, the MessageData virtual register on page 73,
the `L4_LoadMR()` function on page 247, the `L4_LoadMRs()` function on page 247,
the `L4_StoreMR()` function on page 247, the `L4_StoreMRs()` function on page 248,
section A-6.3 on page 42 and section A-6.5 on page 43.

D-6.1.10 `L4_LreplyWait()`

```
L4_MsgTag_t L4_LreplyWait(
        L4_ThreadId_t  target,
        L4_ThreadId_t* result_ptr);
```

The `L4_LreplyWait()` function is an alias of `L4_ReplyWait()`. It is included in the language bindings for historical reasons and should not be used by new programs.

See also: the `L4_ReplyWait()` function on page 182.

D-6.1.11 `L4_ReplyWait_Nonblocking()`

```
L4_MsgTag_t L4_ReplyWait_Nonblocking(
        L4_ThreadId_t  target,
        L4_ThreadId_t* result_ptr);
```

The `L4_ReplyWait_Nonblocking(`*target*`,` *result_ptr*`)` function performs an IPC *reply* operation to the thread specified by the *target* argument, followed by an IPC *receive* operation from any OKL4 thread. It is implemented using an appropriate invocation of the Ipc system call. The content of the message is obtained from the MessageData registers of the caller, which may be modified using the `L4_LoadMR()` or `L4_LoadMRs()` function immediately prior to the invocation of `L4_ReplyWait_Nonblocking()`. The received message is delivered in the MessageData registers of the caller and can be accessed using the `L4_StoreMR()` or `L4_StoreMRs()` function immediately following the invocation of `L4_ReplyWait_Nonblocking()`. `L4_ReplyWait()` returns the value of the $\mathit{Tag}_{\text{OUT}}$ parameter from the Ipc system call.

See also: the Ipc system call on page 86, the **CapId** data type on page 52,
the `L4_ThreadId_t` type on page 242, the **MessageTag** data type on page 59,
the `L4_MsgTag_t` type on page 193, the MessageData virtual register on page 73,

the `L4_LoadMR()` function on page 247, the `L4_LoadMRs()` function on page 247,
the `L4_StoreMR()` function on page 247, the `L4_StoreMRs()` function on page 248,
section A-6.3 on page 42 and section A-6.5 on page 43.

D-6.1.12 `L4_WaitForever()`

```
L4_MsgTag_t L4_WaitForever(void);
```

The `L4_WaitForever()` function performs an IPC *wait* operation from NILTHREAD, effectively blocking the calling thread until the IPC is aborted by another thread using the ExchangeRegisters system call. It is implemented using an appropriate invocation of the Ipc system call. Under normal conditions, this function never returns. However, if the IPC operation is aborted by a third party, `L4_WaitForever()` returns the value of the Tag_{OUT} parameter from the Ipc system call containing an error indicator.

See also: the Ipc system call on page 86, the NILTHREAD data constructor on page 52,
the **MESSAGETAG** data type on page 59, the `L4_MsgTag_t` type on page 193 and section A-6.5 on page 43.

D-6.1.13 Additional C++ Functions

```
L4_MsgTag_t L4_Send(
        L4_ThreadId_t  target,
        L4_MsgTag_t    tag);
L4_MsgTag_t L4_Receive(
        L4_ThreadId_t  target,
        L4_MsgTag_t    tag);
L4_MsgTag_t L4_ReplyWait(
        L4_ThreadId_t  target,
        L4_MsgTag_t    tag,
        L4_ThreadId_t* result_ptr);
```

When used with a C++ compiler, the `<l4/ipc.h>` header overloads a number of C++ functions with the prototypes specified above. Each of these C++ functions has identical semantics to the following C functions:

C++ Expression:	Equivalent C Expression:
`L4_Send(target, tag)`	`L4_Ipc(target, L4_nilthread, tag, NULL)`
`L4_Receive(target, tag)`	`L4_Ipc(L4_nilthread, target, tag, NULL)`
`L4_ReplyWait(target, tag, r_ptr)`	`L4_Ipc(target, L4_anythread, tag, r_ptr)`

D-6.2 Message Tags

The `<l4/ipc.h>` header defines a number of types and functions used to construct and analyse values of the OKL4 **MESSAGETAG** data type represented by the `L4_MsgTag_t` C type defined in the `<l4/message.h>` header.

D-6.2.1 L4_IpcSucceeded()

```
L4_Bool_t L4_IpcSucceeded(
        L4_MsgTag_t tag);
```

The L4_IpcSucceeded(*tag*) function returns a non-zero value if the *S* flag in the *tag* argument is cleared. Otherwise, L4_IpcSucceeded(*tag*) returns zero.

See also: the **MESSAGETAG** data type on page 59, the L4_MsgTag_t type on page 193 and the L4_Bool_t type on page 234.

D-6.2.2 L4_IpcFailed()

```
L4_Bool_t L4_IpcFailed(
        L4_MsgTag_t tag);
```

The L4_IpcFailed(*tag*) function returns the value of the *S* flag in the *tag* argument.

See also: the **MESSAGETAG** data type on page 59, the L4_MsgTag_t type on page 193 and the L4_Bool_t type on page 234.

D-6.2.3 L4_IpcXcpu()

```
L4_Bool_t L4_IpcXcpu(
        L4_MsgTag_t tag);
```

The L4_IpcXcpu(*tag*) function returns the value of the *R* flag in the *tag* argument.

See also: the **MESSAGETAG** data type on page 59, the L4_MsgTag_t type on page 193 and the L4_Bool_t type on page 234.

D-6.2.4 L4_Set_Notify()

```
void L4_Set_Notify(
        L4_MsgTag_t* tag_ptr);
```

The L4_Set_Notify(*tag_ptr*) function sets the *N* flag in the **MESSAGETAG** object pointed to by the *tag_ptr* argument.

See also: the **MESSAGETAG** data type on page 59, the L4_MsgTag_t type on page 193 and the L4_Bool_t type on page 234.

D-6.2.5 L4_Set_ReceiveBlock()

```
void L4_Set_ReceiveBlock(
        L4_MsgTag_t* tag_ptr);
```

The L4_Set_ReceiveBlock(*tag_ptr*) function sets the *R* flag in the **MessageTag** object pointed to by the *tag_ptr* argument.

See also: the **MessageTag** data type on page 59, the L4_MsgTag_t type on page 193 and
the L4_Bool_t type on page 234.

D-6.2.6 L4_Clear_ReceiveBlock()

```
void L4_Clear_ReceiveBlock(
        L4_MsgTag_t* tag_ptr);
```

The L4_Clear_ReceiveBlock(*tag_ptr*) function clears the *R* flag in the **MessageTag** object pointed to by the *tag_ptr* argument.

See also: the **MessageTag** data type on page 59, the L4_MsgTag_t type on page 193 and
the L4_Bool_t type on page 234.

D-6.2.7 L4_Set_SendBlock()

```
void L4_Set_SendBlock(
        L4_MsgTag_t* tag_ptr);
```

The L4_Set_SendBlock(*tag_ptr*) function sets the *S* flag in the **MessageTag** object pointed to by the *tag_ptr* argument.

See also: the **MessageTag** data type on page 59, the L4_MsgTag_t type on page 193 and
the L4_Bool_t type on page 234.

D-6.2.8 L4_Clear_SendBlock()

```
void L4_Clear_SendBlock(
        L4_MsgTag_t* tag_ptr);
```

The L4_Clear_SendBlock(*tag_ptr*) function clears the *S* flag in the **MessageTag** object pointed to by the *tag_ptr* argument.

See also: the **MessageTag** data type on page 59, the L4_MsgTag_t type on page 193 and
the L4_Bool_t type on page 234.

D-6.2.9 L4_Set_MemoryCopy()

```
void L4_Set_MemoryCopy(
        L4_MsgTag_t* tag_ptr);
```

The L4_Set_MemoryCopy(*tag_ptr*) function sets the *M* flag in the **MessageTag** object pointed to by to by the *tag_ptr* argument.

See also: the **MessageTag** data type on page 59, the L4_MsgTag_t type on page 193 and the L4_Bool_t type on page 234.

D-6.2.10 L4_Clear_MemoryCopy()

```
void L4_Clear_MemoryCopy(
        L4_MsgTag_t* tag_ptr);
```

The L4_Clear_MemoryCopy(*tag_ptr*) function clears the *M* flag in the **MessageTag** object pointed to by to by the *tag_ptr* argument.

See also: the **MessageTag** data type on page 59, the L4_MsgTag_t type on page 193 and the L4_Bool_t type on page 234.

D-7 The <l4/map.h> Header

The <l4/map.h> header provides the definitions of a number of macros, types, constants and functions used to access the functionality of the OKL4 MapControl system call described in Section A-12.6 on page 89. It automatically includes all definitions provided by the <l4/types.h> header described in Chapter D-14. Note that <l4/map.h> is automatically included by the <l4/space.h> header, which is the preferred way of accessing its definitions in user programs.

D-7.1 L4_MapItem_Map()

```
void L4_MapItem_Map(
      L4_MapItem_t     *item,
      L4_Word_t        segment_id,
      L4_Word_t        offset,
      L4_Word_t        vaddr,
      L4_Word_t        page_size,
      L4_Word_t        attr,
      L4_Word_t        rwx
);
```

The L4_MapItem_Map() function is used to create a MapItem specifying the characteristics of a mapping which may then be used by either the L4_ProcessMapItem() or L4_ProcessMapItems() function provided by the <l4/space.h> header to create new mapping.

See also: the **MAPCONTROL** data type on page 58 and the MapControl system call on page 89.

D-7.2 L4_MapItem_MapReplace()

```
void L4_MapItem_MapReplace(
      L4_MapItem_t     *item,
      L4_Word_t        segment_id,
      L4_Word_t        offset,
      L4_Word_t        vaddr,
      L4_Word_t        page_size,
      L4_Word_t        attr,
      L4_Word_t        rwx
);
```

The L4_MapItem_MapReplace() function is used to create a MapItem which may then be used by the L4_ProcessMapItem() or L4_ProcessMapItems() function provided by the <l4/space.h> header to replace an existing mapping.

See also: the **MAPCONTROL** data type on page 58 and the MapControl system call on page 89.

D-7.3 L4_MapItem_Unmap()

```
void L4_MapItem_Unmap(
      L4_MapItem_t     *item,
      L4_Word_t        vaddr,
      L4_Word_t        page_size
);
```

The `L4_MapItem_Unmap()` function is used to create MapItem which may then be used by either the `L4_ProcessMapItem()` or `L4_ProcessMapItems()` function provided by the `<l4/space.h>` header to unmap an existing mapping. The size of the mapping to be unmapped is specified by the *page_size* argument.

See also: the **MAPCONTROL** data type on page 58 and the MapControl system call on page 89.

D-7.4 L4_MapItem_SparseUnmap()

```
void L4_MapItem_SparseUnmap(
      L4_MapItem_t     *item,
      L4_Word_t        vaddr,
      L4_Word_t        page_size
);
```

The `L4_MapItem_SparseUnmap()` function is used to create a MapItem which may then be used by either the `L4_ProcessMapItem()` or `L4_ProcessMapItems()` function provided by the `<l4/space.h>` header to unmap a series of mappings located in an area of size, *page_size*. Note that the entire region does not need to be mapped.

See also: the **MAPCONTROL** data type on page 58 and the MapControl system call on page 89.

D-7.5 L4_MapItem_SetMultiplePages()

```
void L4_MapItem_SetMultiplePages(
      L4_MapItem_t     *item,
      L4_Word_t        num_pages
);
```

The `L4_MapItem_SetMultiplePages()` function is used to alter an existing MapItem so that it may be used to map multiple pages instead of a single page. This MapItem can then be supplied as an argument to either the `L4_ProcessMapItem()` or `L4_ProcessMapItems()` function provided by the `<l4/space.h>` header to execute the mapping.

See also: the **MAPCONTROL** data type on page 58 and the MapControl system call on page 89.

D-7.6 L4_PageRightsSupported()

```
L4_Bool_t L4_PageRightsSupported(
      L4_Word_t       rwx);
```

The L4_PageRightsSupported() function allows the user to check whether a particular read, write, execute combination is supported by the underlying hardware. This function returns *true* if the supplied combination is supported and *false* if it is not.

See also: the **MAPCONTROL** data type on page 58 and the MapControl system call on page 89.

D-8 The `<l4/message.h>` Header

The `<l4/message.h>` header complements the `<l4/ipc.h>` header with a number of C definitions that provide access to the IPC-related virtual registers, as well as simplifying the construction of IPC message components. It automatically includes all definitions provided by the `<l4/types.h>` header described in Chapter D-14. Note that `<l4/message.h>` is automatically included by the `<l4/ipc.h>` header and is the preferred way of accessing its definitions in user programs.

D-8.1 Virtual Registers

The `<l4/message.h>` header defines a number of C functions that provide access to the IPC-related OKL4 virtual registers defined in Chapter A-11.

D-8.1.1 `L4_Accepted()`

```
L4_Acceptor_t L4_Accepted(void);
```

The `L4_Accepted()` function returns the value of the Acceptor virtual register of the current thread.

See also: the Acceptor virtual register on page 72, the **ACCEPTOR** data type on page 50,
the `L4_Acceptor_t` type on page 197 and the `L4_Accept()` function on page 191.

D-8.1.2 `L4_Accept()`

```
void L4_Accept(
        L4_Acceptor_t value);
```

`L4_Accept(`*acceptor*`)` sets the Acceptor virtual register of the current thread to the value specified by the *value* argument.

See also: the Acceptor virtual register on page 72, the **ACCEPTOR** data type on page 50,
the `L4_Acceptor_t` type on page 197 and the `L4_Accepted()` function on page 191.

D-8.1.3 `L4_Get_NotifyMask()`

```
L4_Word_t L4_Get_NotifyMask(void);
```

The `L4_Get_NotifyMask()` function returns the value of the NotifyMask virtual register of the current thread.

See also: the NotifyMask virtual register on page 74, the `L4_Word_t` type on page 234 and
the `L4_Set_NotifyMask()` function on page 192.

D-8.1.4 L4_Set_NotifyMask()

```
void L4_Set_NotifyMask(
        L4_Word_t mask);
```

The L4_Set_NotifyMask(*mask*) function sets the NotifyMask virtual register of the current thread to the value specified by the *mask* argument.

See also: the NotifyMask virtual register on page 74, the L4_Word_t type on page 234 and the L4_Get_NotifyMask() function on page 191.

D-8.1.5 L4_Get_NotifyBits()

```
L4_Word_t L4_Get_NotifyBits(void);
```

The L4_Get_NotifyBits() function returns the value of the NotifyFlags virtual register of the current thread.

See also: the NotifyFlags virtual register on page 73, the L4_Word_t type on page 234 and the L4_Set_NotifyBits() function on page 192.

D-8.1.6 L4_Set_NotifyBits()

```
void L4_Set_NotifyBits(
        L4_Word_t flags);
```

The L4_Set_NotifyBits(*flags*) function sets the current thread's NotifyFlags virtual register to the value specified by the *flags* argument.

See also: the NotifyFlags virtual register on page 73, the L4_Word_t type on page 234 and the L4_Get_NotifyBits() function on page 192.

D-8.2 Message Tags

The <l4/message.h> header defines a number of C functions that provide access to the OKL4 **MessageTag** data type.

D-8.2.1 L4_MsgTag_t

Field:	Type:	Description:
raw	L4_Word_t	A raw binary representation of the **MessageTag** object.
X.label	L4_Word_t :*n*	Alternative access to the *Label* field of the MessageTag object. The width of this field (*n*) is calculated by the C expression L4_BITS_PER_WORD - 16.
X.flags	L4_Word_t :4	Alternative access to the *S*, *R*, *N* and *P* flags in the MessageTag object.
X.u	L4_Word_t :6	Alternative access to the *Untyped* field of the MessageTag object.

The L4_MsgTag_t provides a C representation of the OKL4 **MessageTag** data type. It is implemented as a C union type containing at least the members listed in the above table. A particular implementation may extend the definition of the L4_MsgTag_t type with additional members, provided that the total size of the L4_MsgTag_t type, as returned by the sizeof operator, is less than the size of L4_Word_t.

The recommended method of creating and modifying objects of the L4_MsgTag_t type is using the functions provided by the language bindings library. The user may choose to modify this object directly, assuming knowledge of the binary representation of the OKL4 **MessageTag** type defined in Part B of this manual. The language bindings also provide direct access to the individual fields of the OKL4 MessageTag constructor through the X.label, X.flags and X.u fields.

WARNING: The use of these fields is discouraged, as it relies on the unportable behaviour of common C/C++ compiler implementations. Several compilers, including some versions of GCC, may apply program transformations which may render values accessed or modified through these fields incorrect. Any use of these fields in conjunction with such compilers will render the state of the current address space undefined.

See also: the **MessageTag** data type on page 59, the **Word** data type on page 65 and the L4_Word_t type on page 234.

D-8.2.2 L4_Niltag

```
#define L4_Niltag implementation-defined value
```

The L4_Niltag macro provides a C representation of an OKL4 MessageTag constructor with all fields and flags set to 0. It expands to an implementation-defined value of the type L4_MsgTag_t.

See also: the **MessageTag** data type on page 59 and the L4_MsgTag_t type on page 193.

D-8.2.3 L4_Notifytag

```
#define L4_Notifytag implementation-defined value
```

The L4_Notifytag macro provides a C representation of an OKL4 MessageTag constructor with the *N* flag set and all other fields set to 0. It expands to an implementation-defined value of the type L4_MsgTag_t.

See also: the **MessageTag** data type on page 59 and the L4_MsgTag_t type on page 193.

D-8.2.4 `L4_Waittag`

```
#define L4_Waittag implementation-defined value
```

The `L4_Waittag` macro provides a C representation of an OKL4 MESSAGETAG constructor with the *R* flag set and all other fields set to 0. It expands to an implementation-defined value of the type `L4_MsgTag_t`.

See also: the **MESSAGETAG** data type on page 59 and the `L4_MsgTag_t` type on page 193.

D-8.2.5 `L4_Label()`

```
L4_Word_t L4_Label(
        L4_MsgTag_t tag);
```

The `L4_Label(`*tag*`)` function returns the value of the *Label* field of the *tag* argument.

See also: the **MESSAGETAG** data type on page 59, the `L4_MsgTag_t` type on page 193 and the `L4_Word_t` type on page 234.

D-8.2.6 `L4_UntypedWords()`

```
L4_Word_t L4_UntypedWords(
        L4_MsgTag_t tag);
```

The `L4_UntypedWords(`*tag*`)` function returns the value of the *Untyped* field of the *tag* argument.

See also: the **MESSAGETAG** data type on page 59, the `L4_MsgTag_t` type on page 193 and the `L4_Word_t` type on page 234.

D-8.2.7 `L4_Set_Label()`

```
void L4_Set_Label(
        L4_MsgTag_t* tag_ptr,
        L4_Word_t    label);
```

The `L4_Set_Label(`*tag_ptr*`, `*label*`)` function sets the value of the *Label* field in the MESSAGETAG object pointed to by the *tag_ptr* argument to the value specified by the *label* argument.

See also: the **MESSAGETAG** data type on page 59, the `L4_MsgTag_t` type on page 193 and the `L4_Word_t` type on page 234.

D-8.2.8 `L4_MsgTagAddLabel()`

```
L4_MsgTag_t L4_MsgTagAddLabel(
        L4_MsgTag_t tag,
        L4_Word_t   label);
```

The `L4_MsgTagAddLabel(`*tag*`,` *label*`)` function returns a copy of the MessageTag object specified by the *tag* argument with the *Label* field set to the value specified by the *label* argument.

See also: the **MessageTag** data type on page 59, the `L4_MsgTag_t` type on page 193 and the `L4_Word_t` type on page 234.

D-8.2.9 `L4_MsgTagAddLabelTo()`

```
L4_MsgTag_t L4_MsgTagAddLabelTo(
        L4_MsgTag_t* tag_ptr,
        L4_Word_t    label);
```

`L4_MsgTagAddLabelTo(`*tag_ptr*`,` *label*`)` sets the value of the *Label* field in the MessageTag object pointed to by the *tag_ptr* argument to the value specified by the *label* argument and returns the value of the modified message tag.

See also: the **MessageTag** data type on page 59, the `L4_MsgTag_t` type on page 193 and the `L4_Word_t` type on page 234.

D-8.2.10 `L4_Make_MsgTag()`

```
L4_MsgTag_t L4_Make_MsgTag(
        L4_Word_t label
        L4_Word_t flags);
```

The `L4_Make_MsgTag(`*label*`,` *flags*`)` function returns a new message tag with the label and flags set as specified by the arguments.

See also: the **MessageTag** data type on page 59 and the `L4_MsgTag_t` type on page 193.

D-8.2.11 `L4_IsMsgTagEqual()`

```
L4_Bool_t L4_IsMsgTagEqual(
        L4_MsgTag_t tag1,
        L4_MsgTag_t tag2);
```

The `L4_IsMsgTagEqual(textittag1, textittag2)` function returns `1` when the *tag1* and *tag2* arguments have identical values. Otherwise, `L4_IsMsgTagEqual(`*tag1*`,` *tag2*`)` returns `0`.

See also: the **MessageTag** data type on page 59, the `L4_MsgTag_t` type on page 193 and the `L4_Bool_t` type on page 234.

D-8.2.12 `L4_IsMsgTagNotEqual()`

```
L4_Bool_t L4_IsMsgTagNotEqual(
        L4_MsgTag_t tag1,
        L4_MsgTag_t tag2);
```

The `L4_IsMsgTagNotEqual(textittag1, textittag2)` function returns `1` when the *tag1* and *tag2* arguments have different values. Otherwise, `L4_IsMsgTagNotEqual(`*tag1*`,` *tag2*`)` returns `0`.

See also: the **MessageTag** data type on page 59, the `L4_MsgTag_t` type on page 193 and the `L4_Bool_t` type on page 234.

D-8.2.13 `L4_MsgTag()`

```
L4_MsgTag_t L4_MsgTag(
        L4_MsgTag_t tag);
```

The `L4_MsgTag()` function returns the value of the MessageData$_0$ virtual register of the current thread as an object of the type `L4_MsgTag_t`.

See also: the MessageData virtual register on page 73 and the `L4_Set_MsgTag()` function on page 196.

D-8.2.14 `L4_Set_MsgTag()`

```
void L4_Set_MsgTag(
        L4_MsgTag_t* tag
        L4_MsgTag_t msg);
```

The `L4_Set_MsgTag(`*tag*`,` *msg*`)` function sets the MessageData$_0$ virtual register specified by the *tag* argument to the value specified by the *msg* argument.

See also: the MessageData virtual register on page 73 and the `L4_MsgTag()` function on page 196.

D-8.2.15 Additional C++ Operators

```
L4_MsgTag_t operator + (
        L4_MsgTag_t tag,
        L4_Word_t   label);
L4_MsgTag_t operator += (
        L4_MsgTag_t& tag_ref,
        L4_Word_t    label);
L4_Bool_t operator == (
        L4_MsgTag_t  tag1,
        L4_MsgTag_t  tag2);
L4_Bool_t operator != (
        L4_MsgTag_t  tag1,
        L4_MsgTag_t  tag2);
```

When used with a C++ compiler, the `<l4/message.h>` header overloads C++ operators with the prototypes specified above. Each operator has identical semantics to the following functions:

C++ Expression:	Equivalent C Expression:
`tag + label`	`L4_MsgTagAddLabel(tag, label)`
`tag += label`	`L4_MsgTagAddLabelTo(&tag, label)`
`x == y`	`L4_IsMsgTagEqual(x, y)`
`x != y`	`L4_IsMsgTagNotEqual(x, y)`

D-8.3 Acceptors

The `<l4/message.h>` header defines a number of C functions that provide access to the OKL4 **ACCEPTOR** data type.

D-8.3.1 `L4_Acceptor_t`

Field:	Type:	Description:
`raw`	`L4_Word_t`	A raw binary representation of the **ACCEPTOR** object.
`X.a`	`L4_Word_t :1`	An alternative means of access to the *Notify* field of the ACCEPTOR object.

`L4_Acceptor_t` provides a C representation of the OKL4 **ACCEPTOR** data type. It is implemented as a C union type containing at least the members listed in the above table. A particular implementation may extend the definition of the `L4_Acceptor_t` type with additional members, provided that the total size of the `L4_Acceptor_t` type, as returned by the `sizeof` operator, is less than the size of the `L4_Word_t` type.

The recommended method of creating and modifying objects of the `L4_Acceptor_t` type is using the functions provided by the language bindings library. The user may choose to modify the `L4_Acceptor_t` member directly, assuming knowledge of the binary representation of the OKL4 **ACCEPTOR** type defined in Part B of this manual. The language bindings also provide direct access to the individual fields of the OKL4 ACCEPTOR constructor through the `X.a` field.

WARNING: The use of this field is discouraged, as it relies on the unportable behaviour of common C/C++ compiler implementations. Several compilers, including some versions of GCC, may apply program transformations which may render values accessed or modified through these fields incorrect. Any use of these fields in conjunction with such compilers will render the state of the current address space undefined.

See also: the Acceptor virtual register on page 72, the **WORD** data type on page 65 and the `L4_Word_t` type on page 234.

D-8.3.2 `L4_NotifyMsgAcceptor`

```
#define L4_NotifyMsgAcceptor implementation-defined value
```

The `L4_NotifyMsgAcceptor` macro provides a C representation of an OKL4 ACCEPTOR constructor with the *Notify* flag set and all other fields set to 0 and expands to an implementation-defined value of the type `L4_Acceptor_t`.

See also: the **ACCEPTOR** data type on page 50 and the `L4_Acceptor_t` type on page 197.

D-8.3.3 `L4_AsynchItemsAcceptor`

```
#define L4_AsynchItemsAcceptor L4_NotifyMsgAcceptor
```

`L4_AsynchItemsAcceptor` is an alias of the `L4_NotifyMsgAcceptor` macro and should not be used in new programs designed for operation in the OKL4 environment.

See also: the `L4_NotifyMsgAcceptor` constant on page 197.

D-8.3.4 `L4_AddAcceptor()`

```
L4_Acceptor_t L4_AddAcceptor(
        L4_Acceptor_t a1,
        L4_Acceptor_t a2);
```

`L4_AddAcceptor(`*a1*`, `*a2*`)` returns an ACCEPTOR object with the *Notify* flag set to a union (inclusive-OR) of the *Notify* flags in the ACCEPTOR objects represented by the *a1* and *a2* arguments.

See also: the **ACCEPTOR** data type on page 50 and the `L4_Acceptor_t` type on page 197.

D-8.3.5 `L4_AddAcceptorTo()`

```
L4_Acceptor_t L4_AddAcceptorTo(
        L4_Acceptor_t* a1_ptr,
        L4_Acceptor_t  a2);
```

`L4_AddAcceptorTo(`*a1_ptr*`, `*a2*`)` sets the *Notify* flag in the ACCEPTOR object pointed to by *a1_ptr* to a union (inclusive-OR) of its old value with the *Notify* flags in the ACCEPTOR object represented by the *a2* argument. It returns the value of the modified ACCEPTOR object.

See also: the **ACCEPTOR** data type on page 50 and the `L4_Acceptor_t` type on page 197.

D-8.3.6 `L4_RemoveAcceptor()`

```
L4_Acceptor_t L4_RemoveAcceptor(
        L4_Acceptor_t a1,
        L4_Acceptor_t a2);
```

`L4_AddAcceptor(`*a1*`, `*a2*`)` returns a copy of *a1* with the *Notify* flag cleared if and only if the corresponding flag is set in *a2*.

See also: the **ACCEPTOR** data type on page 50 and the `L4_Acceptor_t` type on page 197.

D-8.3.7 L4_RemoveAcceptorFrom()

```
L4_Acceptor_t L4_RemoveAcceptorFrom(
        L4_Acceptor_t* a1_ptr,
        L4_Acceptor_t  a2);
```

`L4_RemoveAcceptorFrom(`*a1_ptr*`,` *a2*`)` clears the *Notify* flag in the ACCEPTOR object pointed to by the *a1_ptr* argument if and only if the corresponding flag is set in *a2*. It returns the value of the modified ACCEPTOR object.

See also: the **ACCEPTOR** data type on page 50 and the `L4_Acceptor_t` type on page 197.

D-8.3.8 Additional C++ Functions

```
L4_Acceptor_t operator + (
        L4_Acceptor_t  a1,
        L4_Acceptor_t  a2);
L4_Acceptor_t operator += (
        L4_Acceptor_t& a1_ref,
        L4_Acceptor_t  a2);
L4_Acceptor_t operator - (
        L4_Acceptor_t  a1,
        L4_Acceptor_t  a2);
L4_Acceptor_t operator -= (
        L4_Acceptor_t& a1_ref,
        L4_Acceptor_t  a2);
```

When used with a C++ compiler, the `<l4/message.h>` header overloads C++ operators with the prototypes specified above. Each operator has identical semantics to the following objects:

C++ Expression:	Equivalent C Expression:
`x + y`	`L4_AddAcceptor(x, y)`
`x += y`	`L4_AddAcceptorTo(&x, y)`
`x - y`	`L4_RemoveAcceptor(x, y)`
`x -= y`	`L4_RemoveAcceptorFrom(&x, y)`

D-9 The <l4/mutex.h> Header

The <l4/mutex.h> header provides access to the Mutex and MutexControl system calls. This header automatically includes the <l4/types.h> header, making available all types and constants defined in Chapter D-14.

D-9.1 System Calls

The <l4/mutex.h> header defines a set of functions that provide access to the Mutex and MutexControl system calls.

D-9.1.1 L4_Lock()

```
word_t L4_Lock(
        L4_MutexId_t  target);
```

The L4_Lock() function will lock the specified *target* mutex or will block until it is acquired.

See also: the Mutex system call on page 92, the **MUTEXID** data type on page 60 and
the L4_MutexId_t type on page 240.

D-9.1.2 L4_TryLock()

```
word_t L4_TryLock(
        L4_MutexId_t  target);
```

The L4_TryLock() function attempts to lock the specified *target* mutex and returns with an error if the mutex is unavailable.

See also: the Mutex system call on page 92, the **MUTEXID** data type on page 60 and
the L4_MutexId_t type on page 240.

D-9.1.3 L4_HybridLock()

```
word_t L4_HybridLock(
       L4_MutexId_t    target,
       word_t          *state_p);
```

The L4_HybridLock() function is used to attempt to lock the mutex specified by the *target* argument. This function returns 1 on successful completion and 0 on failure.

See also: the Mutex system call on page 92, the **MUTEXID** data type on page 60 and
the L4_MutexId_t type on page 240.

D-9.1.4 `L4_HybridUnlock()`

```
word_t L4_HybridUnlock(
    L4_MutexId_t    target,
    word_t          *state_p);
```

The `L4_HybridUnlock()` function is used release the mutex specified by the *target* argument. This function returns 1 on successful completion and 0 on failure.

See also: the Mutex system call on page 92, the **MutexId** data type on page 60 and
the `L4_MutexId_t` type on page 240.

D-9.1.5 `L4_Unlock()`

```
word_t L4_Unlock(
        L4_MutexId_t target);
```

The `L4_Unlock()` function will attempt to unlock the mutex specified by the *target* parameter.

See also: the Mutex system call on page 92, the **MutexId** data type on page 60 and
the `L4_MutexId_t` type on page 240.

D-9.1.6 `L4_CreateMutex()`

```
word_t L4_CreateMutex(
        L4_MutexId_t target);
```

The `L4_CreateMutex()` function initializes the newly created mutex specified by the *target* argument.

See also: the Mutex system call on page 92, the **MutexId** data type on page 60 and
the `L4_MutexId_t` type on page 240.

D-9.1.7 `L4_DeleteMutex()`

```
word_t L4_DeleteMutex(
        L4_MutexId_t target);
```

The `L4_DeleteMutex()` function deletes the mutex specified by the *target* argument.

See also: the Mutex system call on page 92, the **MutexId** data type on page 60 and
the `L4_MutexId_t` type on page 240.

D-10 The `<l4/schedule.h>` Header

The `<l4/schedule.h>` contains the definitions relating to the control of the OKL4 scheduler and provides access to the Schedule and ThreadSwitch system calls. This header automatically includes the `<l4/types.h>` header, making available all types and constants defined in Chapter D-14.

D-10.1 System Calls

The `<l4/schedule.h>` header defines a set of functions that provide access to both the Schedule and ThreadSwitch system calls.

D-10.1.1 `L4_Set_Priority()`

```
L4_Word_t L4_Set_Priority(
        L4_ThreadId_t target,
        L4_Word_t     p);
```

The `L4_Set_Priority(`*target*`,` *p*`)` function sets the *priority* of the thread specified by the *target* argument to the value specified by *p* using an appropriate invocation of the OKL4 Schedule system call. It returns the *Result*$_{OUT}$ parameter from the Schedule system call. A zero value indicates an error, all other values represent a **THREADSTATE** constructor that identifies the state of the target thread prior to the invocation of the Schedule system call.

See also: the Schedule system call on page 95, the **CAPID** data type on page 52,
the `L4_ThreadId_t` type on page 242, the `L4_Word_t` type on page 234, chapter A-5 on page 37 and
section D-10.2 on page 205.

D-10.1.2 `L4_Set_ProcessorNo()`

```
L4_Word_t L4_Set_ProcessorNo(
        L4_ThreadId_t target,
        L4_Word_t     unit);
```

The `L4_Set_ProcessorNo(`*target*`,` *unit*`)` function sets the *domain* of the thread specified by the *target* argument to the set containing the processing unit specified by the *Domain* argument using an appropriate invocation of the OKL4 Schedule system call. It returns the *Result*$_{OUT}$ parameter from the Schedule system call. A zero value indicates an error, all other values represent a **THREADSTATE** constructor that identifies the state of the target thread prior to the invocation of the Schedule system call.

See also: the Schedule system call on page 95, the **CAPID** data type on page 52,
the `L4_ThreadId_t` type on page 242, the `L4_Word_t` type on page 234, chapter A-5 on page 37 and
section D-10.2 on page 205.

D-10.1.3 `L4_Timeslice()`

```
L4_Word_t L4_Timeslice(
        L4_ThreadId_t target,
        L4_Word_t*    slice_ptr);
```

`L4_Timeslice(`*target*`,` *slice_ptr*`)` obtains the current value of the *remaining time slice* of the thread specified by the *target* argument using an appropriate invocation of the OKL4 Schedule system call. This value is returned using the variable pointed to by the *slice_ptr* argument. `L4_Timeslice()` returns the $\mathit{Result}_{\text{OUT}}$ parameter from the Schedule system call. A zero value indicates an error, all other values represent a **ThreadState** constructor that identifies the state of the target thread prior to the invocation of the Schedule system call.

See also: the Schedule system call on page 95, the **CapId** data type on page 52,
the `L4_ThreadId_t` type on page 242, the `L4_Word_t` type on page 234, chapter A-5 on page 37 and section D-10.2 on page 205.

D-10.1.4 `L4_Set_Timeslice()`

```
L4_Word_t L4_Set_Timeslice(
        L4_ThreadId_t target,
        L4_Word_t     slice);
```

The `L4_Set_Timeslice(`*target*`,` *slice*`)` function sets the *time slice* of the thread specified by the *target* argument to the value specified by the *slice* argument using an appropriate invocation of the OKL4 Schedule system call. It returns the $\mathit{Result}_{\text{OUT}}$ parameter from the Schedule system call. A zero value indicates an error, all other values represent a **ThreadState** constructor that identifies the state of the target thread prior to the invocation of the Schedule system call.

See also: the Schedule system call on page 95, the **CapId** data type on page 52,
the `L4_ThreadId_t` type on page 242, the `L4_Word_t` type on page 234, chapter A-5 on page 37 and section D-10.2 on page 205.

D-10.1.5 `L4_Yield()`

```
void L4_Yield(void);
```

The `L4_Yield()` function invokes the OKL4 ThreadSwitch system call with the $\mathit{Target}_{\text{IN}}$ parameter set to NilThread.

See also: the ThreadSwitch system call on page 103 and chapter A-5 on page 37.

D-10.2 Thread States

The `<l4/schedule.h>` header defines a set of macros that provide symbolic names for all return values from `L4_Schedule()` and the functions derived from it. These macros represent the data constructors of the OKL4 **THREADSTATE** data type.

`L4_SCHEDRESULT_ERROR`	The microkernel was unable to perform the operation requested by the corresponding Schedule system call.
`L4_SCHEDRESULT_DEAD`	The target thread was in a new state.
`L4_SCHEDRESULT_INACTIVE`	The target thread was inactive or stopped.
`L4_SCHEDRESULT_RUNNING`	The target thread was executing at the time of the invocation of the Schedule system call.
`L4_SCHEDRESULT_PENDING_SEND`	The target thread has requested an IPC *send* operation.
`L4_SCHEDRESULT_SENDING`	The target thread is performing an IPC *send* or *reply* operation.
`L4_SCHEDRESULT_WAITING`	The target thread has requested an IPC *wait* operation.
`L4_SCHEDRESULT_RECEIVING`	The target thread is performing an IPC *wait* or *receive* operation.
`L4_SCHEDRESULT_WAITING_MUTEX`	The target thread is performing a *blocking* Mutex *lock* operation.

D-11 The `<l4/space.h>` Header

The `<l4/space.h>` header provides access to the MapControl call. It includes all the definitions provided by the `<l4/types.h>` header described in Chapter D-14 and the `<l4/map.h>` header described in Chapter D-7.

D-11.1 System Calls

The `<l4/space.h>` header defines the functions that provide access to the MapControl system call.

D-11.1.1 `L4_ProcessMapItem()`

```
L4_Word_t L4_ProcessMapItem(
                L4_SpaceId_t  s,
                L4_MapItem_t  map_item);
```

The `L4_ProcessMapItem()` function is used to process a single **MAPITEM** specified by the *map_item* argument with regards to the target address space specified by the argument, *s*, by calling the MapControl system call. This function returns 1 on successful completion and 0 on failure.

See also: the MapControl system call on page 89 and the **MAPITEM** data type on page 59.

D-11.1.2 `L4_ProcessMapItems()`

```
L4_Word_t L4_ProcessMapItems(
                L4_SpaceId_t  s,
                L4_Word_t     n,
                L4_MapItem_t  *map_items);
```

The `L4_ProcessMapItems()` function is used to process a *n* **MAPITEM** arguments specified by the **map_items* argument with regards to the target address space specified by the argument, *s*, by calling the MapControl system call. This function returns 1 on successful completion and 0 on failure.

See also: the MapControl system call on page 89 and the **MAPITEM** data type on page 59.

D-12 The `<l4/syscalls.h>` Header

The `<l4/syscalls.h>` header contains the definitions relating to OKL4 system calls.

D-12.1 System Calls

D-12.1.1 `L4_CacheControl()`

```
L4_Word_t L4_CacheControl(
        L4_SpaceId_t   target,
        L4_Word_t      control);
```

The `L4_CacheControl()` function provides direct access to the OKL4 CacheControl system call. The input parameters are supplied as the following arguments to this function:

Input Parameter:	C Function Argument:
Target$_{\text{IN}}$	*target*
Control$_{\text{IN}}$	*control*

The input parameters *RegionAddress*$_{i_{\text{IN}}}$ and *RegionOp*$_{i_{\text{IN}}}$ are supplied in MessageData registers, which may be configured using an appropriate invocation of the `L4_LoadMR()` or `L4_LoadMRs()` function immediately prior to invoking `L4_CacheControl()`. This function returns the value of the *Result*$_{\text{OUT}}$ parameter. A zero value indicates an error, all other values indicate the successful completion of the requested microkernel operation.

See also: the CacheControl system call on page 78, the **SpaceId** data type on page 64,
the `L4_SpaceId_t` type on page 241, the MessageData virtual register on page 73,
the `L4_LoadMR()` function on page 247, the `L4_StoreMR()` function on page 247 and
the `L4_Word_t` type on page 234.

D-12.1.2 `L4_Ipc()`

```
L4_MsgTag_t L4_Ipc(
        L4_ThreadId_t  target,
        L4_ThreadId_t  source,
        L4_MsgTag_t    tag,
        L4_ThreadId_t* result_ptr);
```

The `L4_Ipc()` function provides direct access to the OKL4 Ipc system call. The fixed input parameters are supplied as the following arguments to `L4_Ipc()`:

Input Parameter:	C Function Argument:
Target$_{\text{IN}}$	*target*
Source$_{\text{IN}}$	*source*
Tag$_{\text{IN}}$	*tag*

The remaining input parameters are supplied in the corresponding OKL4 virtual registers using the virtual register access functions defined in Section D-13.4.

The `L4_Ipc()` function returns the value of the *Tag*$_{\text{OUT}}$ parameter. The *Result*$_{\text{OUT}}$ parameter is returned in the variable pointed to by the *result_ptr* argument to `L4_Ipc()`. The remaining output parameters are returned by the Ipc system call in the corresponding OKL4 virtual registers. Their values may be obtained using the virtual register access functions defined in Section D-13.4.

See also: the Ipc system call on page 86, the **CapId** data type on page 52,
the `L4_ThreadId_t` type on page 242, the **MessageTag** data type on page 59,
the `L4_MsgTag_t` type on page 193, section A-12.5 on page 86 and chapter A-6 on page 41.

D-12.1.3 L4_Lipc()

```
L4_MsgTag_t L4_Lipc(
        L4_ThreadId_t  target,
        L4_ThreadId_t  source,
        L4_MsgTag_t    tag,
        L4_ThreadId_t* source_ptr);
```

The `L4_Lipc()` function is an alias for the `L4_Ipc()` function. It is included in the language bindings for historical reasons, and should not be used in new programs.

See also: the `L4_Ipc()` function on page 209.

D-12.1.4 L4_Notify()

```
L4_MsgTag_t L4_Notify(
        L4_ThreadId_t target,
        L4_Word_t     mask);
```

The `L4_Notify(`*target*`,` *mask*`)` function performs an IPC *notify* operation using an appropriate invocation of the Ipc system call. The notification is delivered to the thread specified by the *target* argument and consists of the set of notification flags specified by the *mask* argument. The `L4_Notify()` function returns the value of the *Tag*$_{\text{OUT}}$ parameter from the Ipc system call.

See also: the Ipc system call on page 86, the **CapId** data type on page 52,
the `L4_ThreadId_t` type on page 242, the **MessageTag** data type on page 59,
the `L4_MsgTag_t` type on page 193 and section A-6.4 on page 43.

D-12.1.5 `L4_WaitNotify()`

```
L4_MsgTag_t L4_WaitNotify(
        L4_Word_t* mask_ptr);
```

The `L4_WaitNotify(`*mask*`)` function performs an IPC *wait* operation, receiving a set of asynchronous notification flags using an appropriate invocation of the Ipc system call. All other message forms will be rejected by the calling thread. The notification is delivered to the caller in the variable pointed to by the *mask_ptr* argument. The `L4_WaitNotify()` function returns the value of the *Tag*$_{OUT}$ parameter from the Ipc system call.

See also: the Ipc system call on page 86, the **CapId** data type on page 52, the `L4_ThreadId_t` type on page 242, the **MessageTag** data type on page 59, the `L4_MsgTag_t` type on page 193 and section A-6.4 on page 43.

D-12.1.6 `L4_Mutex()`

```
L4_Word_t L4_Mutex(
        L4_MutexId_t target,
        L4_Word_t flags);
```

The `L4_Mutex(`*target*`)` function provides direct access to the OKL4 Mutex system call. The input parameters are supplied as the following arguments to `L4_Mutex()`:

Input Parameter:	C Function Argument:
Target$_{IN}$	*target*
Flags$_{IN}$	*flags*

See also: the Mutex system call on page 92, the **MutexId** data type on page 60 and the `L4_MutexId_t` type on page 240.

D-12.1.7 `L4_MutexControl()`

```
L4_Word_t L4_MutexControl(
        L4_MutexId_t target,
        L4_Word_t control);
```

The `L4_Mutex()` function provides direct access to the OKL4 MutexControl system call. The input parameters are supplied as the following arguments to `L4_MutexControl()`:

Input Parameter:	C Function Argument:
Target$_{IN}$	*target*
Control$_{IN}$	*control*

See also: the MutexControl system call on page 93, the **MutexId** data type on page 60 and the `L4_MutexId_t` type on page 240.

D-12.1.8 `L4_Schedule()`

```
L4_Word_t L4_Schedule(
        L4_ThreadId_t target,
        L4_Word_t     time_slice,
        L4_Word_t     hw_thread_bitmask,
        L4_Word_t     domain,
        L4_Word_t     priority,
        L4_Word_t     reserved
        L4_Word_t*    remaining_timeslice)
```

`L4_Schedule()` provides direct access to the OKL4 Schedule system call. The input parameters are supplied as the following arguments to `L4_Schedule()`:

Input Parameter:	C Function Argument:
Target$_{IN}$	*target*
TimeSlice$_{IN}$	*slice*
HwThreadBitMask$_{IN}$	*hw_thread_bitmask*
Domain$_{IN}$	*domain*
Priority$_{IN}$	*priority*
Reserved$_{IN}$	*reserved*

The *reserved* argument to the `L4_Schedule()` function is present for historical reasons. Under current implementations of the OKL4 microkernel, it should always be set to `0`.

`L4_Schedule()` returns the value of the *Result*$_{OUT}$ parameter. A zero value indicates an error, all other values represent a **THREADSTATE** constructor identifying the state of the target thread prior to invoking the Schedule system call. The remaining output parameters are returned in variables pointed to by the following arguments to `L4_Schedule()`:

Output Parameter:	C Function Argument:
Slice$_{OUT}$	**remaining_time_slice*

See also: the Schedule system call on page 95, the **CAPID** data type on page 52, the `L4_ThreadId_t` type on page 242, the `L4_Word_t` type on page 234 and chapter A-5 on page 37.

D-12.1.9 `L4_ThreadSwitch()`

```
void L4_ThreadSwitch(
        L4_ThreadId_t target);
```

The `L4_ThreadSwitch(`*target*`)` function provides direct access to the OKL4 ThreadSwitch system call, with the *Target*$_{IN}$ parameter set to the value specified by the *target* argument.

See also: the ThreadSwitch system call on page 103, the **CAPID** data type on page 52, the `L4_ThreadId_t` type on page 242 and chapter A-5 on page 37.

D-12.1.10 `L4_SpaceControl()`

```
L4_Word_t L4_SpaceControl(
        L4_SpaceId_t   target,
        L4_Word_t      control,
        L4_Fpage_t     utcb_region,
        L4_Word_t      resources,
        L4_Word_t*     resources_ptr);
```

The `L4_SpaceControl()` function provides direct access to the OKL4 SpaceControl system call. The input parameters are supplied as the following arguments to `L4_SpaceControl()`:

Input Parameter:	**C Function Argument:**
$Target_{IN}$	*target*
$Control_{IN}$	*control*
$UtcbRegion_{IN}$	*utcb_region*
$SpaceResources_{IN}$	*resources*

`L4_SpaceControl()` returns the value of the $Result_{OUT}$ parameter. A zero value indicates an error, all other values represent the successful completion of the requested operation. $SpaceResources_{OUT}$ is returned in the variable pointed to by the *resources_ptr* argument.

See also: the **SPACECONTROL** data type on page 63, the SpaceControl system call on page 97,
the `L4_SpaceId_t` type on page 241, the `L4_Fpage_t` type on page 234 and
the `L4_Word_t` type on page 234.

D-12.1.11 `L4_MapControl()`

```
L4_Word_t L4_MapControl(
        L4_SpaceId_t   target,
        L4_Word_t      control);
```

The `L4_MapControl()` function provides direct access to the OKL4 MapControl system call. The fixed input parameters are supplied as the following arguments to `L4_MapControl()`:

Input Parameter:	**C Function Argument:**
$Target_{IN}$	*target*
$Control_{IN}$	*control*

The remaining input parameters ($FPage_{i_{IN}}$ and $PPage_{i_{IN}}$) are supplied in the MessageData registers, which should be configured using an appropriate invocation of `L4_LoadMR()` or `L4_LoadMRs()` immediately prior to the invocation of the `L4_MapControl()` system call.

`L4_MapControl()` returns the value of the $Result_{OUT}$ parameter. A zero value indicates an error and all other values represent the successful completion of the requested operation. The output parameters $MapItem_{i_{OUT}}$ and $PPage_{i_{OUT}}$ are returned in MessageData registers. Their values may be obtained immediately following an invocation the MapControl system call using `L4_StoreMR()` or `L4_StoreMRs()`.

See also: the MapControl system call on page 89, the **MAPCONTROL** data type on page 58, **??** on page **??**,
?? on page **??**, the **SPACEID** data type on page 64, the `L4_SpaceId_t` type on page 241,

the **FPAGE** data type on page 56, the L4_Fpage_t type on page 234, **??** on page **??**, **??** on page **??**, the MessageData virtual register on page 73, the L4_LoadMR() function on page 247, the L4_StoreMR() function on page 247 and chapter A-3 on page 31.

D-13 The <l4/thread.h> Header

The <l4/thread.h> header contains the definitions relating to the manipulation of OKL4 threads. It also provides access to most OKL4 virtual registers as well as the ExchangeRegisters and ThreadControl system calls. This header automatically includes <l4/types.h>, making available all types and constants defined in Chapter D-14.

D-13.1 System Calls

The <l4/thread.h> header defines the functions that provide access to both the ExchangeRegisters and ThreadControl system calls.

D-13.1.1 L4_ExchangeRegisters()

```
L4_ThreadId_t L4_ExchangeRegisters(
        L4_ThreadId_t  target,
        L4_Word_t      control,
        L4_Word_t      sp,
        L4_Word_t      ip,
        L4_Word_t      flags,
        L4_Word_t      user_data,
        L4_Word_t*     control_ptr,
        L4_Word_t*     sp_ptr,
        L4_Word_t*     ip_ptr,
        L4_Word_t*     flags_ptr,
        L4_Word_t*     user_data_ptr);
```

The L4_ExchangeRegisters() function provides direct access to the OKL4 ExchangeRegisters system call. The input parameters are supplied as the following arguments to L4_ExchangeRegisters():

Input Parameter:	C Function Argument:
Target$_{\text{IN}}$	*target*
Control$_{\text{IN}}$	*control*
StackPointer$_{\text{IN}}$	*sp*
InstructionPointer$_{\text{IN}}$	*ip*
UserHandle$_{\text{IN}}$	*user_data*

L4_ExchangeRegisters() returns the value of the *Target*$_{\text{OUT}}$ parameter. The remaining output parameters are returned in variables pointed to by the following arguments to `L4_ExchangeRegisters()`:

Output Parameter:	**C Function Argument:**
$\mathit{Control}_{\mathrm{OUT}}$	$\star$*control_ptr*
$\mathit{StackPointer}_{\mathrm{OUT}}$	$\star$*sp_ptr*
$\mathit{InstructionPointer}_{\mathrm{OUT}}$	$\star$*ip_ptr*
$\mathit{UserHandle}_{\mathrm{OUT}}$	$\star$*user_data_ptr*

See also: the ExchangeRegisters system call on page 82, the **REGCONTROL** data type on page 62, the **CAPID** data type on page 52, the `L4_ThreadId_t` type on page 242, the StackPointer virtual register on page 76, the InstructionPointer virtual register on page 73, the UserHandle virtual register on page 76 and the `L4_Word_t` type on page 234.

D-13.1.2 `L4_ThreadControl()`

```
L4_Word_t L4_ThreadControl(
        L4_ThreadId_t target,
        L4_SpaceId_t  space,
        L4_ThreadId_t dummy,
        L4_ThreadId_t pager,
        L4_ThreadId_t exception_handler,
        L4_Word_t     resources,
        void*         utcb);
```

The `L4_ThreadControl()` function provides direct access to the OKL4 ThreadControl system call. The input parameters are supplied as the following arguments to `L4_ThreadControl()`:

Input Parameter:	**C Function Argument:**
$\mathit{Target}_{\mathrm{IN}}$	*target*
$\mathit{Space}_{\mathrm{IN}}$	*space*
$\mathit{Dummy}_{\mathrm{IN}}$	*ignored argument*
$\mathit{Pager}_{\mathrm{IN}}$	*pager*
$\mathit{ExceptionHandler}_{\mathrm{IN}}$	*exception_handler*
$\mathit{ThreadResources}_{\mathrm{IN}}$	*resources*
$\mathit{Utcb}_{\mathrm{IN}}$	*utcb*

The *reserved* argument to the `L4_ThreadControl()` function is present for historical reasons. Under all current implementations of the OKL4 microkernel, it should always be set to `0`. `L4_ThreadControl()` returns the value of the $\mathit{Result}_{\mathrm{OUT}}$ parameter. A zero value indicates an error, all other values represent the successful completion of the requested operation.

See also: the ThreadControl system call on page 100, the **CAPID** data type on page 52, the `L4_ThreadId_t` type on page 242, the **SPACEID** data type on page 64, the `L4_SpaceId_t` type on page 241 and section A-3.5 on page 34.

D-13.1.3 `L4_UserDefinedHandleOf()`

```
L4_Word_t L4_UserDefinedHandleOf(
        L4_ThreadId_t target);
```

`L4_UserDefinedHandleOf(`*target*`)` returns the current value of the UserHandle virtual register for the thread specified by the *target* argument using an appropriate invocation of the ExchangeRegisters system call. `L4_UserDefinedHandleOf(`*target*`)` does not allow for error detection as it returns an undefined value on the failure of the ExchangeRegisters system call. Therefore the `L4_ExchangeRegisters()` function should be used where error detection is required.

See also: the UserHandle virtual register on page 76, the ExchangeRegisters system call on page 82, the `L4_ExchangeRegisters()` function on page 215, the **CapId** data type on page 52, the `L4_ThreadId_t` type on page 242, the `L4_Word_t` type on page 234 and the `L4_Set_UserDefinedHandleOf()` function on page 217.

D-13.1.4 `L4_Set_UserDefinedHandleOf()`

```
void L4_Set_UserDefinedHandleOf(
        L4_ThreadId_t target,
        L4_Word_t     value);
```

The `L4_Set_UserDefinedHandleOf(`*target*`, `*value*`)` function sets the value of the UserHandle virtual register of the thread specified by the *target* argument to the value specified by the *value* argument using an appropriate invocation of the ExchangeRegisters system call.

See also: the UserHandle virtual register on page 76, the ExchangeRegisters system call on page 82, the `L4_ExchangeRegisters()` function on page 215, the **CapId** data type on page 52, the `L4_ThreadId_t` type on page 242, the `L4_Word_t` type on page 234 and the `L4_UserDefinedHandleOf()` function on page 217.

D-13.1.5 `L4_Set_PagerOf()`

```
void L4_Set_PagerOf(
        L4_ThreadId_t target,
        L4_ThreadId_t value);
```

The `L4_Set_PagerOf(`*target*`, `*value*`)` function sets the value of the Pager virtual register of the thread specified by the *target* argument to the value specified by the *value* argument using an appropriate invocation of the ThreadControl system call.

See also: the Pager virtual register on page 74, the ThreadControl system call on page 100, the `L4_ExchangeRegisters()` function on page 215, the **CapId** data type on page 52 and the `L4_ThreadId_t` type on page 242.

D-13.1.6 L4_Start()

```
void L4_Start(
        L4_ThreadId_t  target);
```

The L4_Start(*target*) function resumes execution of the thread specified by the *target* argument using an appropriate invocation of the ExchangeRegisters system call. If the target thread is not currently suspended, the function has no effect.

See also: the ExchangeRegisters system call on page 82,
the L4_ExchangeRegisters() function on page 215, the **CapId** data type on page 52 and
the L4_ThreadId_t type on page 242.

D-13.1.7 L4_Start_SpIp()

```
void L4_Start_SpIp(
        L4_ThreadId_t  target);
        L4_Word_t      sp,
        L4_Word_t      ip);
```

The L4_Start_SpIp(*target*, *sp*, *ip*) function sets the StackPointer and InstructionPointer registers of the thread specified by the *target* argument and resumes its execution if currently suspended, using an appropriate invocation of the ExchangeRegisters system call.

See also: the ExchangeRegisters system call on page 82,
the L4_ExchangeRegisters() function on page 215, the **CapId** data type on page 52,
the L4_ThreadId_t type on page 242, the L4_Word_t type on page 234,
the StackPointer virtual register on page 76 and the InstructionPointer virtual register on page 73.

D-13.1.8 L4_Start_SpIpFlags()

```
void L4_Start_SpIpFlags(
        L4_ThreadId_t  target);
        L4_Word_t      sp,
        L4_Word_t      ip,
        L4_Word_t      flags);
```

The L4_Start_SpIpFlags(*target*, *sp*, *ip*, *flags*) function sets the StackPointer, InstructionPointer and Flags registers of the thread specified by the *target* argument and resumes its execution if currently suspended, using an appropriate invocation of the ExchangeRegisters system call.

See also: the ExchangeRegisters system call on page 82,
the L4_ExchangeRegisters() function on page 215, the **CapId** data type on page 52,
the L4_ThreadId_t type on page 242, the L4_Word_t type on page 234,
the StackPointer virtual register on page 76, the InstructionPointer virtual register on page 73 and
the Flags virtual register on page 72.

D-13.1.9 `L4_Stop_Thread()`

```
void L4_Stop_Thread(
        L4_ThreadId_t target);
```

The `L4_Stop_Thread(`*target*`)` function suspends execution of the thread specified by the *target* argument using an appropriate invocation of the ExchangeRegisters system call. If the target thread is already suspended, this function has no effect.

See also: the ExchangeRegisters system call on page 82, the **CAPID** data type on page 52 and
the `L4_ThreadId_t` type on page 242.

D-13.1.10 `L4_Stop()`

```
L4_ThreadState_t L4_Stop(
        L4_ThreadId_t target);
```

The `L4_Stop(`*target*`)` function suspends execution of the thread specified by the *target* argument using an appropriate invocation of the ExchangeRegisters system call. If the target thread is already suspended, this function has no effect. It returns the value of the *Control*$_{\text{OUT}}$ parameter. This function does not allow for error detection as it returns an undefined value on the failure of the ExchangeRegisters system call. Therefore the `L4_ExchangeRegisters()` function should be used where error detection is required.

See also: the ExchangeRegisters system call on page 82, the **REGCONTROL** data type on page 62,
the `L4_ThreadState_t` type on page 228, the **CAPID** data type on page 52 and
the `L4_ThreadId_t` type on page 242.

D-13.1.11 `L4_Stop_SpIpFlags()`

```
L4_ThreadState_t L4_Stop_SpIpFlags(
        L4_ThreadId_t thread,
        L4_Word_t*    sp_ptr,
        L4_Word_t*    ip_ptr,
        L4_Word_t*    flags_ptr);
```

The `L4_Stop_SpIpFlags()` function suspends execution of the thread specified by the *target* argument using an appropriate invocation of the ExchangeRegisters system call. If the target thread is already suspended, this function has no effect. It returns the value of the *Control*$_{\text{OUT}}$ parameter and the current values of StackPointer, InstructionPointer and Flags registers in the variables pointed to by the *sp_ptr*, *ip_ptr* and *flags_ptr*, respectively. `L4_Stop_SpIpFlags()` does not allow error detection as it returns an undefined value upon failure of the ExchangeRegisters system call. Therefore the `L4_ExchangeRegisters()` function should be used where error detection is required.

See also: the ExchangeRegisters system call on page 82, the StackPointer virtual register on page 76,
the InstructionPointer virtual register on page 73, the Flags virtual register on page 72,
the **REGCONTROL** data type on page 62, the `L4_ThreadState_t` type on page 228 and
the `L4_ThreadId_t` type on page 242.

D-13.1.12 L4_AbortReceive_and_stop_Thread()

```
void L4_AbortReceive_and_stop_Thread(
        L4_ThreadId_t target);
```

L4_AbortReceive_and_stop_Thread(*target*) suspends the execution of the thread specified by the *target* argument and aborts any IPC *wait* and *receive* operations currently performed by that thread using an appropriate invocation of the ExchangeRegisters system call. If the target thread is already suspended, this function has no effect.

See also: the ExchangeRegisters system call on page 82, the **CAPID** data type on page 52,
the L4_ThreadId_t type on page 242 and chapter A-6 on page 41.

D-13.1.13 L4_AbortReceive_and_stop()

```
L4_ThreadState_t L4_AbortReceive_and_stop(
        L4_ThreadId_t target);
```

The L4_AbortReceive_and_stop(*target*) function suspends execution of the thread specified by the *target* argument and aborts any IPC *wait* and *receive* operations currently performed by that thread using an appropriate invocation of the ExchangeRegisters system call. If the target thread is already suspended, this function has no effect. It returns the value of the *Control*$_{\text{OUT}}$ parameter. This function does not allow error detection as it returns an undefined value upon failure of the ExchangeRegisters system call. Therefore the L4_ExchangeRegisters() function should be used where error detection is required.

See also: the ExchangeRegisters system call on page 82, the **REGCONTROL** data type on page 62,
the L4_ThreadState_t type on page 228, the **CAPID** data type on page 52,
the L4_ThreadId_t type on page 242 and chapter A-6 on page 41.

D-13.1.14 L4_AbortReceive_and_stop_SpIpFlags()

```
L4_ThreadState_t L4_AbortReceive_and_stop_SpIpFlags(
        L4_ThreadId_t thread,
        L4_Word_t*    sp_ptr,
        L4_Word_t*    ip_ptr,
        L4_Word_t*    flags_ptr);
```

The L4_AbortReceive_and_stop_SpIpFlags() function suspends execution of the thread specified by the *target* argument and aborts any IPC *wait* and *receive* operations currently performed by that thread using an appropriate invocation of the ExchangeRegisters system call. If the target thread is already suspended, this function has no effect. It returns the value of the *Control*$_{\text{OUT}}$ parameter and the current values of the StackPointer, InstructionPointer and Flags registers in the variables pointed to by *sp_ptr*, *ip_ptr* and *flags_ptr*. This function does not allow for error detection as it returns an undefined value on the failure of the ExchangeRegisters system call. Therefore the L4_ExchangeRegisters() function should be used where error detection is required.

See also: the ExchangeRegisters system call on page 82, the StackPointer virtual register on page 76,
the InstructionPointer virtual register on page 73, the Flags virtual register on page 72,
the **REGCONTROL** data type on page 62, the L4_ThreadState_t type on page 228,
the L4_ThreadId_t type on page 242 and chapter A-6 on page 41.

D-13.1.15 L4_AbortSend_and_stop_Thread()

```
void L4_AbortSend_and_stop_Thread(
        L4_ThreadId_t target);
```

The `L4_AbortSend_and_stop_Thread(`*target*`)` function suspends execution of the thread specified by the *target* argument and aborts any IPC *send* and *reply* operations currently performed by that thread using an appropriate invocation of the ExchangeRegisters system call. Where the target thread is currently suspended, this function has no effect.

See also: the ExchangeRegisters system call on page 82, the **CapId** data type on page 52,
the `L4_ThreadId_t` type on page 242 and chapter A-6 on page 41.

D-13.1.16 L4_AbortSend_and_stop()

```
L4_ThreadState_t L4_AbortSend_and_stop(
        L4_ThreadId_t target);
```

`L4_AbortSend_and_stop(`*target*`)` suspends the execution of the thread specified by the *target* argument and aborts any IPC *send* and *reply* operations currently performed by that thread using an appropriate invocation of the ExchangeRegisters system call. It returns the value of the $Control_{\text{OUT}}$ parameter and has no effect if the target thread is already suspended. This function does not allow error detection as it returns an undefined value on the failure of the ExchangeRegisters system call. Therefore the `L4_ExchangeRegisters()` function should be used where error detection is required.

See also: the ExchangeRegisters system call on page 82, the **RegControl** data type on page 62,
the `L4_ThreadState_t` type on page 228, the **CapId** data type on page 52,
the `L4_ThreadId_t` type on page 242 and chapter A-6 on page 41.

D-13.1.17 L4_AbortSend_and_stop_SpIpFlags()

```
L4_ThreadState_t L4_AbortSend_and_stop_SpIpFlags(
        L4_ThreadId_t thread,
        L4_Word_t*    sp_ptr,
        L4_Word_t*    ip_ptr,
        L4_Word_t*    flags_ptr);
```

The `L4_AbortSend_and_stop_SpIpFlags()` function suspends execution of the thread specified by the *target* argument and aborts any IPC *send* and *reply* operations currently performed by that thread using an appropriate invocation of the ExchangeRegisters system call. If the target thread is already suspended, this function has no effect. It returns the value of the $Control_{\text{OUT}}$ parameter and the current values of the target thread's StackPointer, InstructionPointer and Flags registers in the variables pointed to by the *sp_ptr*, *ip_ptr* and *flags_ptr*. This function does not allow error detection as it returns an undefined value on the failure of the ExchangeRegisters system call. Therefore the `L4_ExchangeRegisters()` function should be used where error detection is required.

See also: the ExchangeRegisters system call on page 82, the StackPointer virtual register on page 76,
the InstructionPointer virtual register on page 73, the Flags virtual register on page 72,
the **RegControl** data type on page 62, the `L4_ThreadState_t` type on page 228,
the `L4_ThreadId_t` type on page 242 and chapter A-6 on page 41.

D-13.1.18 L4_AbortIpc_and_stop_Thread()

```
void L4_AbortIpc_and_stop_Thread(
        L4_ThreadId_t target);
```

The L4_AbortIpc_and_stop_Thread(*target*) function suspends execution of the thread specified by the *target* argument and aborts all IPC operations currently performed by that thread using an appropriate invocation of the ExchangeRegisters system call. If the target thread is already suspended, this function has no effect.

See also: the ExchangeRegisters system call on page 82, the **CapId** data type on page 52,
the L4_ThreadId_t type on page 242 and chapter A-6 on page 41.

D-13.1.19 L4_AbortIpc_and_stop()

```
L4_ThreadState_t L4_AbortIpc_and_stop(
        L4_ThreadId_t target);
```

The L4_AbortIpc_and_stop(*target*) function suspends execution of the thread specified by the *target* argument and aborts all IPC operations currently performed by that thread using an appropriate invocation of the ExchangeRegisters system call. It returns the value of the *Control*$_{\text{OUT}}$ parameter and has no effect if the target thread is already suspended. This function does not allow error detection as it returns an undefined value on the failure of the ExchangeRegisters system call. Therefore the L4_ExchangeRegisters() function should be used where error detection is required.

See also: the ExchangeRegisters system call on page 82, the **RegControl** data type on page 62,
the L4_ThreadState_t type on page 228, the **CapId** data type on page 52,
the L4_ThreadId_t type on page 242 and chapter A-6 on page 41.

D-13.1.20 L4_AbortIpc_and_stop_SpIpFlags()

```
L4_ThreadState_t L4_AbortIpc_and_stop_SpIpFlags(
        L4_ThreadId_t thread,
        L4_Word_t*    sp_ptr,
        L4_Word_t*    ip_ptr,
        L4_Word_t*    flags_ptr);
```

The L4_AbortIpc_and_stop_SpIpFlags() function suspends execution of the thread specified by the *target* argument and aborts all IPC operations currently performed by that thread using an appropriate invocation of the ExchangeRegisters system call. If the target thread is already suspended, this function has no effect. It returns the value of the *Control*$_{\text{OUT}}$ parameter and the current values of the StackPointer, InstructionPointer and Flags registers in the variables pointed to by the *sp_ptr*, *ip_ptr* and *flags_ptr*. This function does not allow for error detection as it returns an undefined value on the failure of the ExchangeRegisters system call. Therefore the L4_ExchangeRegisters() function should be used where error detection is required.

See also: the ExchangeRegisters system call on page 82, the StackPointer virtual register on page 76,
the InstructionPointer virtual register on page 73, the Flags virtual register on page 72,
the **RegControl** data type on page 62, the L4_ThreadState_t type on page 228,
the L4_ThreadId_t type on page 242 and chapter A-6 on page 41.

D-13.1.21 `L4_CopyVolatile_regs()`

```
void L4_CopyVolatile_regs(
        L4_ThreadId_t source,
        L4_ThreadId_t target);
```

The `L4_CopyVolatile_regs(`*source*`,` *target*`)` overwrites all volatile general-purpose registers of the thread specified by the *target* argument with the values of the corresponding registers of the thread specified by the *source* argument.

See also: the ExchangeRegisters system call on page 82, the **CapId** data type on page 52 and
the `L4_ThreadId_t` type on page 242.

D-13.1.22 `L4_CopySaved_regs()`

```
void L4_CopySaved_regs(
        L4_ThreadId_t source,
        L4_ThreadId_t target);
```

The `L4_CopySaved_regs(`*source*`,` *target*`)` overwrites all saved general-purpose registers in the thread specified by the *target* argument with the values of the corresponding registers of the thread specified by the *source* argument.

See also: the ExchangeRegisters system call on page 82, the **CapId** data type on page 52 and
the `L4_ThreadId_t` type on page 242.

D-13.1.23 `L4_CopySaved_regs_SpIp()`

```
void L4_CopySaved_regs_SpIp(
        L4_ThreadId_t source,
        L4_ThreadId_t target,
        L4_Word_t     sp,
        L4_Word_t     ip);
```

The `L4_CopySaved_regs_SpIp()` function overwrites all saved general-purpose registers of the thread specified by the *target* argument with the values of the corresponding registers of the thread specified by the *source* argument. It also sets the StackPointer and InstructionPointer registers of the target thread to the values specified by the *sp* and *ip* arguments.

See also: the ExchangeRegisters system call on page 82, the StackPointer virtual register on page 76,
the InstructionPointer virtual register on page 73, the **CapId** data type on page 52 and
the `L4_ThreadId_t` type on page 242.

D-13.1.24 L4_Copy_regs()

```
void L4_Copy_regs(
        L4_ThreadId_t source,
        L4_ThreadId_t target);
```

The L4_Copy_regs(*source*, *target*) function overwrites all the general-purpose registers in the thread specified by the *target* argument with the values of the corresponding registers of the thread specified by the *source* argument.

See also: the ExchangeRegisters system call on page 82, the **CapId** data type on page 52 and
the L4_ThreadId_t type on page 242.

D-13.1.25 L4_Copy_regs_SpIp()

```
void L4_Copy_regs_SpIp(
        L4_ThreadId_t source,
        L4_ThreadId_t target,
        L4_Word_t     sp,
        L4_Word_t     ip);
```

The L4_Copy_regs_SpIp() function overwrites all general-purpose registers of the thread specified by *target* argument with the values of the corresponding registers of the thread specified by the *source* argument. It sets the StackPointer and InstructionPointer registers of the target thread to the values specified by the *sp* and *ip* arguments.

See also: the ExchangeRegisters system call on page 82, the StackPointer virtual register on page 76,
the InstructionPointer virtual register on page 73, the **CapId** data type on page 52 and
the L4_ThreadId_t type on page 242.

D-13.1.26 L4_Copy_regs_to_mrs()

```
void L4_Copy_regs_to_mrs(
        L4_ThreadId_t source);
```

The L4_Copy_regs_to_mrs(*source*) function delivers the values of all general-purpose registers of the thread specified by the *target* argument in the MessageData registers of the caller using the ExchangeRegisters system call.

See also: the ExchangeRegisters system call on page 82, the **CapId** data type on page 52,
the L4_ThreadId_t type on page 242, the MessageData virtual register on page 73,
the L4_StoreMR() function on page 247 and the L4_StoreMRs() function on page 248.

D-13.1.27 Additional C++ Functions

```
L4_Word_t L4_UserDefinedHandle(
        L4_ThreadId_t target);
void L4_Set_UserDefinedHandle(
        L4_ThreadId_t target,
        L4_Word_t     value);
void L4_Set_Pager(
        L4_ThreadId_t target,
        L4_ThreadId_t value);
void L4_Start(
        L4_ThreadId_t target,
        L4_Word_t     sp,
        L4_Word_t     ip);
void L4_Start(
        L4_ThreadId_t target,
        L4_Word_t     sp,
        L4_Word_t     ip,
        L4_Word_t     flags);
L4_ThreadState_t L4_Stop(
        L4_ThreadId_t target,
        L4_Word_t*    sp_ptr,
        L4_Word_t*    ip_ptr,
        L4_Word_t*    flags_ptr);
L4_ThreadState_t L4_AbortReceive_and_stop(
        L4_ThreadId_t target,
        L4_Word_t*    sp_ptr,
        L4_Word_t*    ip_ptr,
        L4_Word_t*    flags_ptr);
L4_ThreadState_t L4_AbortSend_and_stop(
        L4_ThreadId_t target,
        L4_Word_t*    sp_ptr,
        L4_Word_t*    ip_ptr,
        L4_Word_t*    flags_ptr);
L4_ThreadState_t L4_AbortIpc_and_stop(
        L4_ThreadId_t target,
        L4_Word_t*    sp_ptr,
        L4_Word_t*    ip_ptr,
        L4_Word_t*    flags_ptr);
```

When used with a C++ compiler, the <l4/thread.h> header overloads a number of C++ functions with the prototypes specified above. Each of these functions has identical semantics to the following functions:

C++ Expression:	Equivalent C Expression:
`L4_UserDefinedHandle(t)`	`L4_UserDefinedHandleOf(t)`
`L4_Set_UserDefinedHandle(t,x)`	`L4_Set_UserDefinedHandleOf(t,x)`
`L4_Set_Pager(t,x)`	`L4_Set_PagerOf(t,x)`
`L4_Start(t,s,i)`	`L4_Start_SpIp(t,s,i)`
`L4_Start(t,s,i,f)`	`L4_Start_SpIpFlags(t,s,i,f)`
`L4_Stop(t,s,i,f)`	`L4_Stop_SpIpFlags(t,s,i,f)`
`L4_AbortReceive_and_stop(t,s,i,f)`	`L4_AbortReceive_and_stop_SpIpFlags(t,s,i,f)`
`L4_AbortSend_and_stop(t,s,i,f)`	`L4_AbortSend_and_stop_SpIpFlags(t,s,i,f)`
`L4_AbortIpc_and_stop(t,s,i,f)`	`L4_AbortIpc_and_stop_SpIpFlags(t,s,i,f)`

D-13.2 The `L4_ExchangeRegisters()` *control* Argument

The `L4_ExchangeRegisters()` function uses a *control* argument of type `L4_Word_t` to represent the *Control*$_{\text{IN}}$ parameter to the OKL4 ExchangeRegisters system call. Since the binary encoding of the OKL4 **REGCONTROL** data type may differ between individual OKL4 implementations, it is recommended that all programs use the symbolic macros listed below to construct the desired value of the *control* argument to `L4_ExchangeRegisters()` using the inclusive-OR (`|`) operator.

L4_ExReg_sp Indicates that the ExchangeRegisters system call is being used to set the value of the StackPointer register of the target thread. It corresponds to the *S* flag in the REGCONTROL data constructor.

L4_ExReg_ip Indicates that the ExchangeRegisters system call is being used to set the InstructionPointer register of the target thread. This macro corresponds to the *I* flag in the REGCONTROL data constructor.

L4_ExReg_flags Indicates that the ExchangeRegisters system call is being used to set the Flags register of the target thread. This macro corresponds to the *F* flag in the REGCONTROL data constructor.

L4_ExReg_sp_ip Indicates that the ExchangeRegisters system call is being used to set the StackPointer and InstructionPointer registers of the target thread. Its value is formed by a bitwise inclusive-OR of the two macros `L4_ExReg_sp` and `L4_ExReg_ip`.

L4_ExReg_sp_ip_flags Indicates that the ExchangeRegisters system call is being used to set the StackPointer, InstructionPointer and Flags registers of the target thread. Its value is formed by an inclusive-OR of the three macros `L4_ExReg_sp`, `L4_ExReg_ip` and `L4_ExReg_flags`.

L4_ExReg_user Indicates that the ExchangeRegisters system call is being used to set the value of the UserHandle register of the target thread. This macro corresponds to the *U* flag in the REGCONTROL data constructor.

L4_ExReg_Resume Indicates that the ExchangeRegisters system call is being used to resume execution a halted target thread. This macro corresponds to the *HE* flag in the REGCONTROL data constructor.

L4_ExReg_Halt Indicates that the ExchangeRegisters system call is being used to suspend execution of the target thread. This macro corresponds to the *HE* and *H* flags in the REGCONTROL data constructor.

L4_ExReg_AbortSendIPC Indicates that the ExchangeRegisters system call is being used to abort the *send* and *reply* IPC operations performed by the target thread. This macro corresponds to the *AS* flag in the REGCONTROL data constructor.

L4_ExReg_AbortRecvIPC Indicates that the ExchangeRegisters system call is being used to abort the *wait* and *receive* IPC operations performed by the target thread. This macro corresponds to the *AR* flag in the REGCONTROL data constructor.

L4_ExReg_AbortIPC Indicates that the ExchangeRegisters system call is being used to abort all IPC operations performed by the target thread. Its value is an inclusive-OR of the L4_ExReg_AbortSendIPC and L4_ExReg_AbortRecvIPC macros.

L4_ExReg_Deliver Indicates that the ExchangeRegisters system call is being used to obtain the current values of the StackPointer, InstructionPointer, Flags and UserHandle virtual registers of the target thread. This macro corresponds to the *D* flag in the REGCONTROL data constructor.

L4_ExReg_VolatileRegs Indicates that the ExchangeRegisters system call is being used to set the values of all volatile registers of the target thread. It should always be combined with the ID of the source thread from which the register values are to be obtained using the L4_ExReg_SrcThread() function. This macro corresponds to the *VR* flag in the REGCONTROL data constructor.

L4_ExReg_SavedRegs Indicates that ExchangeRegisters is being used to set the values of all saved registers of the target thread. It should be always combined with the ID of the source thread from which the register values are to be obtained using the L4_ExReg_SrcThread() function. This macro corresponds to the *SR* flag in the REGCONTROL data constructor.

L4_ExReg_RegsToMRs Indicates that the ExchangeRegisters system call is being used to store all volatile and saved registers of the target thread in the MessageData registers of the caller. This macro corresponds to the *MR* flag in the REGCONTROL data constructor.

See also: the **REGCONTROL** data type on page 62, the ExchangeRegisters system call on page 82,
the L4_ExchangeRegisters() function on page 215, the L4_Word_t type on page 234 and
the L4_ExReg_SrcThread() function on page 227.

D-13.2.1 L4_ExReg_SrcThread()

```
L4_Word_t L4_ExReg_SrcThread(
        L4_ThreadId_t source);
```

L4_ExReg_SrcThread(*source*) is used to set the value of the *Source* field in the REGCONTROL OKL4 data constructor that is supplied as the *control* parameter to the L4_ExchangeRegisters() function. This field is only used when at least one of the corresponding *VR* or *SR* flags is set in the REGCONTROL constructor. These flags indicate that the ExchangeRegisters system call should overwrite the values of the saved or volatile registers of the target thread with the values of the corresponding registers of the thread specified in the *Source* argument. L4_ExReg_SrcThread(*source*) returns the *ThreadNo* field of the *source* argument as a L4_Word_t that may be combined with the macros defined in Section D-13.2.

See also: the **REGCONTROL** data type on page 62, the ExchangeRegisters system call on page 82,
the L4_ExchangeRegisters() function on page 215, the **CAPID** data type on page 52,
the L4_ThreadId_t type on page 242 and the L4_Word_t type on page 234.

D-13.3 The `L4_ThreadState_t` Type

The following C functions returns the thread state of the target thread in the form of a `L4_ThreadState_t` object:

- `L4_Stop()`
- `L4_Stop_SpIpFlags()`
- `L4_AbortReceive_and_stop()`
- `L4_AbortReceive_and_stop_SpIpFlags()`
- `L4_AbortSend_and_stop()`
- `L4_AbortSend_and_stop_SpIpFlags()`
- `L4_AbortIpc_and_stop()`
- `L4_AbortIpc_and_stop_SpIpFlags()`

In all cases, the returned value describes the state of the target thread prior to invoking the ExchangeRegisters system call. The language bindings library provides a set of functions for determining the status of such a thread from the `L4_ThreadState_t` object.

D-13.3.1 `L4_ThreadState_t`

Field:	**Type:**	**Description:**
`raw`	`L4_Word_t`	A raw binary representation of the REGCONTROL data constructor.

The `L4_ThreadState_t` type is used to describe the value of the *Control*$_{\text{OUT}}$ parameter returned by the OKL4 ExchangeRegisters system call. It is implemented as a structure or union with a single member of type `L4_ThreadState_t`. Individual OKL4 implementations may define additional members of the type `L4_ThreadState_t`, provided that the total size of `L4_ThreadState_t`, as returned by the `sizeof` operator, is less than `L4_Word_t`.

See also: the **REGCONTROL** data type on page 62, the ExchangeRegisters system call on page 82,
the `L4_ThreadWasHalted()` function on page 228,
the `L4_ThreadWasReceiving()` function on page 229,
the `L4_ThreadWasSending()` function on page 229,
the `L4_ThreadWasIpcing()` function on page 230, the `L4_Stop()` function on page 219,
the `L4_Stop_SpIpFlags()` function on page 219,
the `L4_AbortReceive_and_stop()` function on page 220,
the `L4_AbortReceive_and_stop_SpIpFlags()` function on page 220,
the `L4_AbortSend_and_stop()` function on page 221,
the `L4_AbortSend_and_stop_SpIpFlags()` function on page 221,
the `L4_AbortIpc_and_stop()` function on page 222 and
the `L4_AbortIpc_and_stop_SpIpFlags()` function on page 222.

D-13.3.2 `L4_ThreadWasHalted()`

```
L4_Bool_t L4_ThreadWasHalted(
        L4_ThreadState_t state);
```

`L4_ThreadWasHalted(`*state*`)` is called with a *state* parameter returned by a convenience function listed in Section D-13.3. It returns a non-zero value if the corresponding convenience function was called with a *target* parameter referring to a thread whose execution was suspended prior to the invocation of the ExchangeRegisters system call.

See also: the ExchangeRegisters system call on page 82, the **REGCONTROL** data type on page 62,
the `L4_Bool_t` type on page 234, the `L4_Stop()` function on page 219,

the L4_Stop_SpIpFlags() function on page 219,
the L4_AbortReceive_and_stop() function on page 220,
the L4_AbortReceive_and_stop_SpIpFlags() function on page 220,
the L4_AbortSend_and_stop() function on page 221,
the L4_AbortSend_and_stop_SpIpFlags() function on page 221,
the L4_AbortIpc_and_stop() function on page 222 and
the L4_AbortIpc_and_stop_SpIpFlags() function on page 222.

D-13.3.3 L4_ThreadWasReceiving()

```
L4_Bool_t L4_ThreadWasReceiving(
        L4_ThreadState_t state);
```

The L4_ThreadWasReceiving(*state*) function is called with a *state* parameter returned by one of the convenience functions listed in Section D-13.3. It returns a non-zero value if the corresponding convenience function was called with a *target* parameter referring to a thread that was performing an IPC *wait* or *receive* operation at the time of the invocation of the ExchangeRegisters system call.

See also: the ExchangeRegisters system call on page 82, the **RegControl** data type on page 62,
the L4_Bool_t type on page 234, the L4_Stop() function on page 219,
the L4_Stop_SpIpFlags() function on page 219,
the L4_AbortReceive_and_stop() function on page 220,
the L4_AbortReceive_and_stop_SpIpFlags() function on page 220,
the L4_AbortSend_and_stop() function on page 221,
the L4_AbortSend_and_stop_SpIpFlags() function on page 221,
the L4_AbortIpc_and_stop() function on page 222,
the L4_AbortIpc_and_stop_SpIpFlags() function on page 222 and chapter A-6 on page 41.

D-13.3.4 L4_ThreadWasSending()

```
L4_Bool_t L4_ThreadWasSending(
        L4_ThreadState_t state);
```

The L4_ThreadWasSending(*state*) function is called with a *state* parameter returned by a convenience function listed in Section D-13.3. It returns a non-zero value if the corresponding convenience function was called with a *target* parameter referring to a thread that was performing an IPC *send* or *reply* operation at the time of the invocation of the ExchangeRegisters system call.

See also: the ExchangeRegisters system call on page 82, the **RegControl** data type on page 62,
the L4_Bool_t type on page 234, the L4_Stop() function on page 219,
the L4_Stop_SpIpFlags() function on page 219,
the L4_AbortReceive_and_stop() function on page 220,
the L4_AbortReceive_and_stop_SpIpFlags() function on page 220,
the L4_AbortSend_and_stop() function on page 221,
the L4_AbortSend_and_stop_SpIpFlags() function on page 221,
the L4_AbortIpc_and_stop() function on page 222,
the L4_AbortIpc_and_stop_SpIpFlags() function on page 222 and chapter A-6 on page 41.

D-13.3.5 `L4_ThreadWasIpcing()`

```
L4_Bool_t L4_ThreadWasIpcing(
        L4_ThreadState_t state);
```

The `L4_ThreadWasIpcing(`*state*`)` function is called with a *state* parameter returned by a convenience function listed in Section D-13.3. It returns a non-zero value if the corresponding convenience function was called with a *target* parameter referring to a thread that was performing an IPC operation at the time of the invocation of the ExchangeRegisters system call.

See also: the ExchangeRegisters system call on page 82, the **REGCONTROL** data type on page 62,
the `L4_Bool_t` type on page 234, the `L4_Stop()` function on page 219,
the `L4_Stop_SpIpFlags()` function on page 219,
the `L4_AbortReceive_and_stop()` function on page 220,
the `L4_AbortReceive_and_stop_SpIpFlags()` function on page 220,
the `L4_AbortSend_and_stop()` function on page 221,
the `L4_AbortSend_and_stop_SpIpFlags()` function on page 221,
the `L4_AbortIpc_and_stop()` function on page 222,
the `L4_AbortIpc_and_stop_SpIpFlags()` function on page 222 and chapter A-6 on page 41.

D-13.4 Virtual Registers

The `<l4/thread.h>` header defines a set of functions that allow user threads to access and modify their virtual registers as described in Chapter A-11 of this manual.

D-13.4.1 `L4_Utcb()`

```
L4_Word_t L4_Utcb(void);
```

The `L4_Utcb()` function returns the value of the Utcb virtual register of the current thread.

See also: the Utcb virtual register on page 76 and section A-3.5 on page 34.

D-13.4.2 `L4_MyThreadHandle()`

```
L4_ThreadId_t L4_MyThreadHandle(void);
```

The `L4_MyThreadHandle()` function returns a constant value of L4_MYSELFCONST.

See also: the **CAPID** data type on page 52 and the `L4_ThreadId_t` type on page 242.

D-13.4.3 `L4_Myself()`

```
L4_ThreadId_t L4_Myself(void);
```

The `L4_Myself()` function returns a constant value of L4_MYSELFCONST.

See also: the **CAPID** data type on page 52 and the `L4_ThreadId_t` type on page 242.

D-13.4.4 L4_ProcessorNo()

```
L4_Word_t L4_ProcessorNo(void);
```

The `L4_ProcessorNo()` function returns the value of the ProcessingUnit virtual register of the current thread.

See also: the Pager virtual register on page 74 and the `L4_Word_t` type on page 234.

D-13.4.5 L4_UserDefinedHandle()

```
L4_Word_t L4_UserDefinedHandle(void);
```

The `L4_UserDefinedHandle()` function returns the value of the UserHandle virtual register of the current thread.

See also: the UserHandle virtual register on page 76,
the `L4_Set_UserDefinedHandle()` function on page 231 and the `L4_Word_t` type on page 234.

D-13.4.6 L4_Set_UserDefinedHandle()

```
void L4_Set_UserDefinedHandle(
        L4_Word_t value);
```

The `L4_Set_UserDefinedHandle(`*value*`)` function sets the UserHandle virtual register of the current thread to the value specified by the *value* argument.

See also: the UserHandle virtual register on page 76,
the `L4_UserDefinedHandle()` function on page 231 and the `L4_Word_t` type on page 234.

D-13.4.7 L4_ErrorCode()

```
L4_Word_t L4_ErrorCode(void);

#define L4_ErrInvalidThread     implementation-defined value
#define L4_ErrInvalidSpace      implementation-defined value
#define L4_ErrInvalidParam      implementation-defined value
#define L4_ErrUtcbArea          implementation-defined value
#define L4_ErrNoMem             implementation-defined value
```

The `L4_ErrorCode()` function returns the value of the ErrorCode virtual register of the current thread. The constructors for the **ErrorCode** OKL4 data type are represented directly by their binary encoding of the type `L4_Word_t`. The `<l4/thread.h>` header defines the following macros that provide symbolic names for the **ErrorCode** data constructors:

L4_ErrInvalidThread The caller specified an invalid thread ID.

L4_ErrInvalidSpace The caller specified a thread ID corresponding to an invalid address space.

L4_ErrInvalidParam The caller specified an invalid parameter to a system call.

L4_ErrUtcbArea The caller specified an invalid UTCB location.

`L4_ErrNoMem` The caller requested an operation that could not be completed due to a lack of system resources.

See also: the ErrorCode virtual register on page 72, the **ERRORCODE** data type on page 55 and the `L4_Word_t` type on page 234.

D-14 The `<l4/types.h>` Header

The `<l4/types.h>` header provides the definitions of a number of macros, types, constants and functions used throughout the language bindings to the OKL4 API. It is automatically included by all headers in the language bindings library and does not need to be included directly by the user.

D-14.1 Feature Identification Macros

`<l4/types.h>` defines three groups of *feature identification macros*. Each OKL4 implementation defines one macro from each group. These macros are intended to be used in conjunction with the `#ifdef` and `#ifndef` directives to identify characteristics of the underlying architecture. All feature identification macros expand to the decimal constant `1`.

D-14.1.1 Hardware Architecture Identification Macros

The `<l4/types.h>` header defines one of the following macros that may be used to identify the underlying architecture of an OKL4 implementation:

`L4_ARCH_ALPHA` This macro is defined if the OKL4 C API implements the Alpha version of the OKL4 API.

`L4_ARCH_AMD64` This macro is defined if the OKL4 C API implements the AMD64 version of the OKL4 API.

`L4_ARCH_ARM` This macro is defined if the OKL4 C API implements the ARM version of the OKL4 API.
See also: chapter C-2 on page 143.

`L4_ARCH_IA32` This macro is defined if the OKL4 C API implements the IA-32 version of the OKL4 API.

`L4_ARCH_IA64` This macro is defined if the OKL4 C API implements the IA-64 version of the OKL4 API.

`L4_ARCH_MIPS` This macro is defined if the OKL4 C API implements the MIPS version of the OKL4 API.

`L4_ARCH_MIPS64` This macro is defined if the OKL4 C API implements the MIPS-64 version of the OKL4 API.

`L4_ARCH_POWERPC` This macro is defined if the OKL4 C API implements the PowerPC version of the OKL4 API.

`L4_ARCH_POWERPC64` This macro is defined if the OKL4 C API implements the PowerPC-64 version of the OKL4 API.

`L4_ARCH_SPARC64` This macro is defined if the OKL4 C API implements the SPARC-64 version of the OKL4 API.

D-14.2 Basic Types

The `<l4/types.h>` header defines the following C types:

`L4_Word_t` `L4_Word_t` is the C representation of the OKL4 **WORD** type. It is an unsigned integral that is unchanged by the C integral promotion rules and is of size `4` on 32-bit architectures and `8` on 64-bit architectures as returned by the `sizeof` operator.
See also: the **WORD** data type on page 65.

`L4_Bool_t` `L4_Bool_t` is the C representation of the OKL4 **FLAG** type. It is an unsigned integral that is unchanged by the C integral promotion rules and is of size `4` on 32-bit architectures and `8` on 64-bit architectures as returned by the `sizeof` operator
See also: the **FLAG** data type on page 55 and
the **WORD** data type on page 65.

`L4_Size_t` `L4_Size_t` is an unsigned integral identical to the type `L4_Word_t`. It is used by the language bindings to represent array indexes and sizes of C objects.
See also: the **WORD** data type on page 65.

`L4_Word8_t` `L4_Word8_t` is an unsigned integral that is of size `1`, as returned by the `sizeof` operator.
See also: the **WORD** data type on page 65.

`L4_Word16_t` `L4_Word16_t` is an unsigned integral that is of size `2`, as returned by the `sizeof` operator.
See also: the **WORD** data type on page 65.

`L4_Word32_t` `L4_Word32_t` is an unsigned integral that is of size `4`, as returned by the `sizeof` operator.
See also: the **WORD** data type on page 65.

`L4_Word64` `L4_Word64` is an unsigned integral that is of size `8`, as returned by the `sizeof` operator.
See also: the **WORD** data type on page 65.

D-14.3 The `L4_Fpage_t` Type and Related Definitions

The `<l4/types.h>` header provides a number of definitions related to the **FPAGE** type.

D-14.3.1 `L4_Fpage_t`

Field:	Type:	Description:
`raw`	`L4_Word_t`	A raw binary representation of the **FPAGE** object.
`X.b`	`L4_Word_t :`*n*	An alternative means of access to the *Base* field of the FPAGE object. The width of this field (*n*) may be calculated by using `L4_BITS_PER_WORD - 10`.
`X.s`	`L4_Word_t :6`	An alternative means of access to the *Width* field of the FPAGE object.
`X.rwx`	`L4_Word_t :3`	An alternative means of access to the *Access* field of the FPAGE object.

`L4_Fpage_t` provides a C representation of the OKL4 **FPAGE** data type. It is implemented as a C union type containing at least the members listed in the above table. A particular implementation may extend the definition of the `L4_Fpage_t` type with additional members, provided that the total size of `L4_Fpage_t` is less than the size of `L4_Word_t`, as returned by the `sizeof` operator.

The recommended method of creating and modifying objects of the type L4_Fpage_t is using the functions provided by the language bindings library. Alternatively, the user may choose to modify this object directly, assuming knowledge of the binary representation of the OKL4 **FPAGE** type defined in Part B of this manual. The language bindings also provide direct access to the individual fields of the OKL4 FPAGE constructor through the X.b, X.s and X.rwx fields.

WARNING: The use of these fields is discouraged as it relies on the unportable behaviour of common C/C++ compiler implementations. Several compilers, including some versions of GCC, may apply program transformations which may render values accessed or modified through these fields incorrect. Any use of these fields in conjunction with such compilers will render the state of the current address space undefined.

See also: the **FPAGE** data type on page 56, the **WORD** data type on page 65 and
the L4_Word_t type on page 234.

D-14.3.2 L4_Nilpage

```
#define L4_Nilpage implementation-defined value
```

The L4_Nilpage macro provides a C representation of the OKL4 NILPAGE constructor of the **FPAGE** data type. It expands to an implementation-defined value of type L4_Fpage_t.

See also: the **FPAGE** data type on page 56, the L4_Fpage_t type on page 234 and
the NILPAGE data constructor on page 56.

D-14.3.3 L4_CompleteAddressSpace

```
#define L4_CompleteAddressSpace implementation-defined value;
```

The L4_CompleteAddressSpace macro provides a C representation of the WHOLESPACE constructor of the **FPAGE** data type. It expands to an implementation-defined value of type L4_Fpage_t.

See also: the **FPAGE** data type on page 56, the L4_Fpage_t type on page 234 and
the WHOLESPACE data constructor on page 56.

D-14.3.4 L4_Fpage()

```
L4_Fpage_t L4_Fpage(
        L4_Word_t start,
        L4_Word_t size);
```

The L4_Fpage(*start*, *size*) function returns an L4_Fpage_t object representing the FPAGE constructor, with the *Base* field set to the virtual page beginning at the address specified by the *start* argument, the *Width* field set to $\log_2(size)$ and the *Access* field set to an empty set of access permissions (∅).

If the *size* argument is not an integral power of 2, is less than $2^{\mathrm{MinPageWidth}}$, or if the *start* argument is not an integral multiple of *size*, L4_Fpage(*start*, *size*) returns an undefined value.

See also: the **FPAGE** data type on page 56, the L4_Fpage_t type on page 234,
the FPAGE data constructor on page 56 and chapter A-3 on page 31.

D-14.3.5 `L4_FpageLog2()`

```
L4_Fpage_t L4_FpageLog2(
        L4_Word_t start,
        int       width);
```

The `L4_FpageLog2`(*start*, *width*) function returns an `L4_Fpage_t` object representing the OKL4 FPAGE constructor, with the *Base* field set to the virtual page beginning at the address specified by *start*, the *Width* field set to the value of the *width* argument and the *Access* field set to an empty set of access permissions (∅).

If the *width* argument is less than MinPageWidth or if the *start* argument is not an integral multiple of 2^{width}, `L4_FpageLog2`(*start*, *width*) returns an undefined value.

See also: the **FPAGE** data type on page 56, the `L4_Fpage_t` type on page 234,
the FPAGE data constructor on page 56 and chapter A-3 on page 31.

D-14.3.6 `L4_IsNilFpage()`

```
L4_Bool_t L4_IsNilFpage(
        L4_Fpage_t fpage);
```

The `L4_IsNilFpage`(*fpage*) function returns `1` when the supplied *fpage* argument represents the NILPAGE data constructor. In all other cases, `L4_IsNilFpage`(*fpage*) returns `0`.

See also: the **FPAGE** data type on page 56, the `L4_Fpage_t` type on page 234,
the NILPAGE data constructor on page 56, the `L4_Nilpage` macro on page 235 and
the `L4_Bool_t` type on page 234.

D-14.3.7 `L4_Address()`

```
L4_Word_t L4_Address(
        L4_Fpage_t fpage);
```

`L4_Address`(*fpage*) returns the value of the *Base* field in the FPAGE constructor represented by the *fpage* argument. If *fpage* represents the NILPAGE or WHOLESPACE constructor, `L4_Address`(*fpage*) returns `0`. In all other cases, the returned value represents the address of the beginning of the virtual page with the given page number.

See also: the **FPAGE** data type on page 56, the `L4_Fpage_t` type on page 234,
the **PAGENO** data type on page 61 and the `L4_Word_t` type on page 234.

D-14.3.8 `L4_Size()`

```
L4_Word_t L4_Size(
        L4_Fpage_t fpage);
```

The `L4_Size`(*fpage*) function returns the number of bytes covered by the region of memory described by the FPAGE constructor represented by the *fpage* argument. If the supplied value of the *fpage* argument does not represent a FPAGE constructor, the returned value is undefined.

See also: the FPAGE data constructor on page 56, the `L4_Fpage_t` type on page 234 and
the `L4_Word_t` type on page 234.

D-14.3.9 `L4_SizeLog2()`

```
L4_Word_t L4_SizeLog2(
        L4_Fpage_t fpage);
```

`L4_SizeLog2(`*fpage*`)` returns the value of the *Width* field in the FPAGE constructor represented by the *fpage* argument. If *fpage* represents the NILPAGE or WHOLESPACE constructors, `L4_SizeLog2(`*fpage*`)` returns `0` and `127`, respectively.

See also: the **FPAGE** data type on page 56, the `L4_Fpage_t` type on page 234 and
the `L4_Word_t` type on page 234.

D-14.3.10 `L4_Rights()`

```
L4_Word_t L4_Rights(
        L4_Fpage_t fpage);
```

`L4_Rights(`*fpage*`)` returns the value of the *Access* field in the FPAGE or WHOLESPACE constructor that is represented by the *fpage* argument. If *fpage* represents the NILPAGE data constructor, the `L4_Rights(`*fpage*`)` function returns an empty set of permissions (∅). In all cases, the returned object is zero-extended to the `L4_Word_t` type.

See also: the **FPAGE** data type on page 56, the `L4_Fpage_t` type on page 234,
the FPAGE data constructor on page 56, the WHOLESPACE data constructor on page 56,
the **PERMISSIONS** data type on page 61 and the `L4_Word_t` type on page 234.

D-14.3.11 `L4_Set_Rights()`

```
L4_Fpage_t L4_Set_Rights(
        L4_Fpage_t* fpage_ptr,
        L4_Word_t   access);
```

The `L4_Set_Rights(`*fpage_ptr*`,` *access*`)` function sets the *Access* field in the FPAGE or WHOLESPACE constructor represented by the object stored at the address specified in the *fpage_ptr* argument to the set of access permissions specified by the *access* argument. It returns a copy of the modified `L4_Fpage_t` object.

If the *fpage_ptr* argument does not point to a valid FPAGE or WHOLESPACE constructor, or if the *access* argument does not specify a valid set of access permissions, `L4_Set_Rights(`*fpage_ptr*`,` *access*`)` returns and sets the supplied `L4_Fpage_t` object to an undefined value.

See also: the **FPAGE** data type on page 56, the `L4_Fpage_t` type on page 234,
the FPAGE data constructor on page 56, the WHOLESPACE data constructor on page 56,
the **PERMISSIONS** data type on page 61 and the `L4_Word_t` type on page 234.

D-14.3.12 L4_FpageAddRights()

```
L4_Fpage_t L4_FpageAddRights(
        L4_Fpage_t fpage,
        L4_Word_t  access);
```

L4_FpageAddRights(*fpage*, *access*) returns a copy of the FPage or WholeSpace constructor that is represented by the *fpage* argument with the *Access* field set to a union of the set of access permissions already specified in the *fpage* objects with those specified by the *access* argument.

If the *fpage* argument does not represent a valid FPage or WholeSpace constructor, or if the *access* argument does not specify a valid set of access permissions, L4_FpageAddRights(*fpage*, *access*) returns an undefined value.

See also: the **FPage** data type on page 56, the L4_Fpage_t type on page 234,
the FPage data constructor on page 56, the WholeSpace data constructor on page 56,
the **Permissions** data type on page 61 and the L4_Word_t type on page 234.

D-14.3.13 L4_FpageAddRightsTo()

```
L4_Fpage_t L4_FpageAddRightsTo(
        L4_Fpage_t* fpage_ptr,
        L4_Word_t   access);
```

L4_FpageAddRightsTo(*fpage_ptr*, *access*) sets the *Access* field in the OKL4 FPage or WholeSpace data constructor represented by the object stored at the address specified in *fpage_ptr* to a union of the set of access permissions already specified in that object with those specified by the *access* argument. It returns a copy of the modified L4_Fpage_t object.

If the *fpage_ptr* argument does not point to a valid FPage or WholeSpace constructor, or if the *access* argument does not specify a valid set of access permissions, L4_FpageAddRightsTo(*fpage_ptr*, *access*) returns and sets the supplied L4_Fpage_t object to an undefined value.

See also: the **FPage** data type on page 56, the L4_Fpage_t type on page 234,
the FPage data constructor on page 56, the WholeSpace data constructor on page 56,
the **Permissions** data type on page 61 and the L4_Word_t type on page 234.

D-14.3.14 L4_FpageRemoveRights()

```
L4_Fpage_t L4_FpageRemoveRights(
        L4_Fpage_t fpage,
        L4_Word_t  access);
```

The L4_FpageRemoveRights(*fpage*, *access*) function returns a copy of the FPage or WholeSpace constructor represented by the *fpage* argument with the *Access* field set to the difference of the set of access permissions already specified in the *fpage* objects and those specified by the *access* argument.

If the *fpage* argument does not represent a valid FPage or WholeSpace constructor, or if the *access* argument does not specify a valid set of access permissions, L4_FpageRemoveRights(*fpage*, *access*) returns an undefined value.

See also: the **FPage** data type on page 56, the L4_Fpage_t type on page 234,
the FPage data constructor on page 56, the WholeSpace data constructor on page 56,
the **Permissions** data type on page 61 and the L4_Word_t type on page 234.

D-14.3.15 `L4_FpageRemoveRightsFrom()`

```
L4_Fpage_t L4_FpageRemoveRightsFrom(
        L4_Fpage_t* fpage_ptr,
        L4_Word_t   access);
```

The `L4_FpageRemoveRightsFrom(`*fpage_ptr*`,` *access*`)` function sets the *Access* field in the FPage or WholeSpace constructor represented by the object stored at the address specified in the *fpage_ptr* argument to a difference of the set of access permissions already specified in that object and those specified by the *access* argument. It returns a copy of the modified `L4_Fpage_t` object.

If the *fpage_ptr* argument does not point to a valid FPage or WholeSpace constructor, or if the *access* argument does not specify a valid set of access permissions, `L4_FpageRemoveRightsFrom(`*fpage_ptr*`,` *access*`)` returns and sets the supplied `L4_Fpage_t` object to an undefined value.

See also: the **FPage** data type on page 56, the `L4_Fpage_t` type on page 234,
the FPage data constructor on page 56, the WholeSpace data constructor on page 56,
the **Permissions** data type on page 61 and the `L4_Word_t` type on page 234.

D-14.3.16 Access Permissions

```
#define L4_Readable        (0x04)
#define L4_Writable        (0x02)
#define L4_eXecutable      (0x01)
#define L4_FullyAccessible (0x07)
#define L4_ReadWriteOnly   (0x06)
#define L4_ReadeXecOnly    (0x05)
#define L4_NoAccess        (0x00)
```

The `<l4/types.h>` header defines the following constants representing the possible combinations of access permissions and parameter combinations to the Permissions constructor.

`L4_Readable` Represents the $\{read\}$ set of access permissions, corresponding to the *R* flag in the Permissions constructor.

`L4_Writable` Represents the $\{write\}$ set of access permissions, corresponding to the *W* flag in the Permissions constructor.

`L4_eXecutable` Represents the $\{execute\}$ set of access permissions, corresponding to the *X* flag in the Permissions constructor.

`L4_FullyAccessible` Represents the $\{read, write, execute\}$ set of access permissions.

`L4_ReadWriteOnly` Represents the $\{read, write\}$ set of access permissions.

`L4_ReadeXecOnly` Represents the $\{read, execute\}$ set of access permissions.

`L4_NoAccess` Represents the empty set of access permissions ($\emptyset$).

Permission sets can also be constructed algebraically using C bit-manipulation operators as follows:

Set Operation:	Expression:	Equivalent C Expression:
Union:	$P \cup Q$	`P	Q`
Intersection:	$P \cap Q$	`P & Q`
Difference:	$P \setminus Q$	`P & ~Q`
Complement:	$\overline{P}$	`P ^ L4_FullyAccessible`

Note that the ~ operator cannot be used directly to compute the complement of a set of permissions, as all permissions sets must remain zero-extended from a 3-bit value. Instead, the exclusive OR operator (^) can be used to achieve the desirable effect as shown above.

See also: the **PERMISSIONS** data type on page 61, the **FPAGE** data type on page 56, the L4_Fpage_t type on page 234 and the PMask system parameter on page 69.

D-14.3.17 Additional C++ Operators

```
L4_Fpage_t operator +  (
        const L4_Fpage_t& fpage,
        L4_Word_t access);
L4_Fpage_t operator += (
        L4_Fpage_t&       fpage,
        L4_Word_t         access);
L4_Fpage_t operator -  (
        const L4_Fpage_t& fpage,
        L4_Word_t         access);
L4_Fpage_t operator -= (
        L4_Fpage_t&       fpage,
        L4_Word_t         access);
```

When used with a C++ compiler, the <l4/types.h> header overloads four of the C++ operators with the prototypes specified above. Each of these operators has identical semantics to the following functions:

C++ Expression:	Equivalent C Expression:
f + a	L4_FpageAddRights(f, a)
f += a	L4_FpageAddRightsTo(&f, a)
f - a	L4_FpageRemoveRights(f, a)
f -= a	L4_FpageRemoveRightsFrom(&f, a)

D-14.4 The L4_MutexId_t Type

The <l4/types.h> header provides some of the definitions that are related to the **MUTEXID** type.

D-14.4.1 L4_MutexId_t

Field:	Type:	Description:
raw	L4_Word_t	A raw binary representation of the mutex identifier object.
space_no	L4_Word_t	An alternative means of access to the *MutexNo* field of the MUTEXNO object.

The L4_MutexNo_t provides a C representation of the OKL4 **MUTEXNO** data type. It is implemented as a C union type containing at least the members listed in the above table. A particular implementation may extend the definition of the L4_MutexId_t type with additional members, provided that the total size of the L4_MutexId_t type is less than the size of L4_Word_t, as returned by the sizeof operator.

The recommended method of creating and modifying objects of the L4_MutexId_t type is using the functions provided by the language bindings library. The user may choose to modify the L4_MutexId_t member directly, assuming knowledge of the binary representation of the **MUTEXNO** type defined in Part B.

WARNING: The use of the these fields is discouraged, as it relies on the unportable behaviour of common C/C++ compiler implementations. Several compilers, including some versions of GCC, may apply program transformations which may render values accessed or modified through these fields incorrect. Any use of these fields in conjunction with such compilers will render the state of the current address space undefined.

See also: the **MutexId** data type on page 60 and the `L4_Word_t` type on page 234.

D-14.5 The `L4_SpaceId_t` Type and Related Definitions

The `<l4/types.h>` header provides some of the definitions that are related to the **SpaceId** type.

D-14.5.1 `L4_SpaceId_t`

Field:	**Type:**	**Description:**
`raw`	`L4_Word_t`	A raw binary representation of the fpage object.
`space_no`	`L4_Word_t`	An alternative means of access to the *SpaceNo* field of the SpaceNo object.

The `L4_SpaceNo_t` provides a C representation of the OKL4 **SpaceNo** data type. It is implemented as a C union type containing at least the members listed in the above table. A particular implementation may extend the definition of the `L4_SpaceNo_t` type with additional members, provided that the total size of the `L4_SpaceNo_t` type is less than the size of `L4_Word_t`, as returned by the `sizeof` operator.

The recommended method of creating and modifying objects of the `L4_SpaceNo_t` type is using the functions provided by the language bindings library. The user may choose to modify the `L4_SpaceNo_t` member directly, assuming knowledge of the binary representation of the **SpaceNo** type defined in Part B.

WARNING: The use of the these fields is discouraged, as it relies on the unportable behaviour of common C/C++ compiler implementations. Several compilers, including some versions of GCC, may apply program transformations which may render values accessed or modified through these fields incorrect. Any use of these fields in conjunction with such compilers will render the state of the current address space undefined.

See also: the **SpaceId** data type on page 64 and the `L4_Word_t` type on page 234.

D-14.5.2 `L4_nilspace`

```
#define L4_nilspace L4_SpaceId(-1UL)
```

The `L4_nilthread` macro provides a C representation of the OKL4 NilThread constructor of the **ThreadId** data type. It expands to a $-1UL$ encoding of `L4_SpaceId_t`.

See also: the **SpaceId** data type on page 64, the `L4_SpaceId_t` type on page 241 and
the NilSpace data constructor on page 64.

D-14.5.3 `L4_rootspace`

```
#define L4_nilspace L4_SpaceId(0)
```

The `L4_nilthread` macro provides a C representation of the OKL4 NilThread constructor of the **ThreadId** data type. It expands to a 0 encoding of `L4_SpaceId_t`.

See also: the **SpaceId** data type on page 64, the `L4_SpaceId_t` type on page 241 and
the NilSpace data constructor on page 64.

D-14.6 The L4_ThreadId_t Type and Related Definitions

The <l4/types.h> header provides some of the definitions that are related to the **ThreadId** type.

D-14.6.1 L4_ThreadId_t

Field:	Type:	Description:
raw	L4_Word_t	A raw binary representation of the fpage object.
cap.X.type	L4_Word_t :*n*	An alternative means of access to the *ThreadNo* field of the ThreadId object. The width of this field (*n*) is 14 or 32 on 32-bit and 64-bit architectures, respectively.
cap.X.index	L4_Word_t :*n*	An alternative means of access to the *Version* field of the ThreadId object. The width of this field (*n*) is 18 or 32 on 32-bit and 64-bit architectures, respectively.

The L4_ThreadId_t provides a C representation of the OKL4 **ThreadId** constructor. It is implemented as a C union type containing at least the members listed in the above table. A particular implementation may extend the definition of the L4_ThreadId_t type with additional members, provided that the total size of the L4_ThreadId_t type is less than the size of L4_Word_t, as returned by the sizeof operator.

It is recommended that objects of the L4_ThreadId_t type are created using the functions provided by the language bindings library. The user may choose to modify the L4_ThreadId_t member directly, assuming knowledge of the binary representation of the **ThreadId** constructor defined in Part B. Compatibility language bindings provide direct access to the individual fields of the OKL4 ThreadId constructor through the cap.X.thread_no and cap.X.index fields.

WARNING: The use of the these fields is discouraged, as it relies on the unportable behaviour of common C/C++ compiler implementations. Several compilers, including some versions of GCC, may apply program transformations which may render values accessed or modified through these fields incorrect. Any use of these fields in conjunction with such compilers will render the state of the current address space undefined.

See also: the **CapId** data type on page 52 and the L4_Word_t type on page 234.

D-14.6.2 L4_nilthread

```
#define L4_nilthread implementation-defined value
```

The L4_nilthread macro provides a C representation of the OKL4 NilThread constructor of the **CapId** data type. It expands to an implementation-defined value of type L4_ThreadId_t.

See also: the **CapId** data type on page 52, the L4_ThreadId_t type on page 242 and the NilThread data constructor on page 52.

D-14.6.3 L4_anythread

```
#define L4_anythread implementation-defined value
```

The L4_anythread macro provides a C representation of the OKL4 AnyThread constructor of the **CapId** data type. It expands to an implementation-defined value of type L4_ThreadId_t.

See also: the **CapId** data type on page 52, the L4_ThreadId_t type on page 242 and the AnyThread data constructor on page 52.

D-14.6.4 L4_waitnotify

```
#define L4_waitnotify implementation-defined value
```

The L4_waitnotify macro provides a C representation of the OKL4 WaitNotify constructor of the data type **CapId**. It expands to an implementation-defined value of type L4_ThreadId_t.

See also: the **CapId** data type on page 52, the L4_ThreadId_t type on page 242 and
the WaitNotify data constructor on page 53.

D-14.6.5 L4_ThreadNo()

```
L4_Word_t L4_ThreadNo(
        L4_ThreadId_t id);
```

The L4_ThreadNo(*id*) function returns the value of the *ThreadNo* field in the ThreadId constructor represented by the *id* argument. If the *id* argument does not represent a valid ThreadId, the returned value is undefined.

See also: the **CapId** data type on page 52, the L4_ThreadId_t type on page 242,
the ThreadId data constructor on page 52, the L4_Word_t type on page 234 and section A-2.2 on page 28.

D-14.6.6 L4_IsNilThread()

```
L4_Bool_t L4_IsNilThread(
        L4_ThreadId_t id);
```

The L4_IsNilThread(*id*) function returns 1 when the supplied *id* represents the NilThread constructor. In all other cases, L4_IsNilThread(*id*) returns 0.

See also: the **CapId** data type on page 52, the L4_ThreadId_t type on page 242,
the NilThread data constructor on page 52, the L4_nilthread macro on page 242 and
the L4_Bool_t type on page 234.

D-14.6.7 L4_IsThreadEqual()

```
L4_Bool_t L4_IsThreadEqual(
        L4_ThreadId_t id1,
        L4_ThreadId_t id2);
```

The L4_IsThreadEqual(textitid1, textitid2) function returns 1 when the supplied *id1* and *id2* arguments have identical values. Otherwise, L4_IsThreadEqual(*id1*, *id2*) returns 0.

See also: the **CapId** data type on page 52, the L4_ThreadId_t type on page 242,
the L4_Bool_t type on page 234 and section A-2.2 on page 28.

D-14.6.8 `L4_IsThreadNotEqual()`

```
        L4_Bool_t L4_IsThreadNotEqual(
                L4_ThreadId_t id1,
                L4_ThreadId_t id2);
```

The `L4_IsThreadNotEqual(textitid1, textitid2)` function returns `1` when the supplied *id1* and *id2* arguments have different values. Otherwise, `L4_IsThreadNotEqual(`*id1*`, `*id2*`)` returns `0`.

See also: the **CapId** data type on page 52, the `L4_ThreadId_t` type on page 242,
the `L4_Bool_t` type on page 234 and section A-2.2 on page 28.

D-14.6.9 Additional C++ Operators

```
        L4_Bool_t operator == (
                const L4_ThreadId_t& id1,
                const L4_ThreadId_t& id2);
        L4_Bool_t operator != (
                const L4_ThreadId_t& id1,
                const L4_ThreadId_t& id2);
```

When used with a C++ compiler, the `<l4/types.h>` header overloads two C++ operators with the prototypes specified above. Each of these operators has identical semantics to the following functions:

C++ Expression:	**Equivalent C Expression:**
`x == y`	`L4_IsThreadEqual(x, y)`
`x != y`	`L4_IsThreadNotEqual(x, y)`

D-14.7 IPC Error Codes

The `<l4/types.h>` header defines a set of macros providing symbolic names for the values of the individual constructors of the **IpcErrorCode** data type:

`L4_ErrTimeout`	No asynchronous notification partner.
`L4_ErrNonExist`	The target thread does not exist.
`L4_ErrCanceled`	The IPC operation has been cancelled.
`L4_ErrMsgOverflow`	Insufficient number of MessageData registers.
`L4_ErrNotAccepted`	Asynchronous notification rejected by the recipient.
`L4_ErrAborted`	IPC operation has been aborted.

See also: the **IpcErrorCode** data type on page 58, the IpcError data constructor on page 55 and
the `L4_IpcError()` function on page 245.

D-14.7.1 L4_IpcError()

```
L4_Bool_t L4_IpcError(
        L4_Word_t  code,
        L4_Word_t* cause_ptr);

#define L4_ErrSendPhase    implementation-defined value
#define L4_ErrRecvPhase    implementation-defined value
```

The `L4_IpcError(`*code*`,` *cause_ptr*`)` function can be used to analyse the content of the ErrorCode virtual register following an invocation of the Ipc system call. The *code* argument must be set to the value obtained from the `L4_ErrorCode()` function following an unsuccessful invocation of the Ipc system call. The ErrorCode virtual register contains a value of the IPCERROR data constructor. The behaviour of `L4_IpcError()` is undefined for all other values of the *code* argument.

`L4_IpcError(`*code*`,` *cause_ptr*`)` returns the value of the *P* flag in the IPCERROR constructor. The `<l4/types.h>` header defines two macros providing symbolic names to the possible values of this flag:

L4_ErrSendPhase The error occurred during an IPC *send* or *reply* operation.

L4_ErrRecvPhase The error occurred during an IPC *wait* or *receive* operation.

See also: the ErrorCode virtual register on page 72, the IPCERROR data constructor on page 55 and the **IPCERRORCODE** data type on page 58.

D-15 The `<l4/utcb.h>` Header

The `<l4/utcb.h>` header defines functions which provide access to the UTCB.

D-15.1 `L4_LoadMR()`

```
void L4_LoadMR(
        L4_Word_t i,
        L4_Word_t data);
```

The `L4_LoadMR(`*i*`,` *data*`)` function sets the value of the MessageData$_i$ virtual register to the value specified by the *data* argument. If *i* greater than the MaxMessageData system parameter, the behaviour of this function is undefined.

See also: the MessageData virtual register on page 73, the `L4_StoreMR()` function on page 247 and the `L4_Word_t` type on page 234.

D-15.2 `L4_StoreMR()`

```
void L4_StoreMR(
        L4_Word_t  i,
        L4_Word_t* data_ptr);
```

The `L4_StoreMR(`*i*`,` *data_ptr*`)` function stores the value of the MessageData$_i$ virtual register in the variable pointed to by the *data_ptr* argument. If *i* is greater than the MaxMessageData system parameter, the behaviour of this function is undefined.

See also: the MessageData virtual register on page 73, the `L4_LoadMR()` function on page 247 and the `L4_Word_t` type on page 234.

D-15.3 `L4_LoadMRs()`

```
void L4_LoadMRs(
        L4_Word_t       i,
        L4_Word_t       n,
        const L4_Word_t data[]);
```

`L4_LoadMRs(`*i*`,` *n*`,` *data*`)` sets the virtual registers, MessageData$_i$ to MessageData$_{i+n-1}$, inclusive, to the values of the corresponding members of the array specified by the *data* argument. If the expression $i+n-1$ is greater than the MaxMessageData system parameter, the behaviour of this function is undefined.

See also: the MessageData virtual register on page 73, the `L4_StoreMRs()` function on page 248 and the `L4_Word_t` type on page 234.

D-15.4 L4_StoreMRs()

```
        void L4_StoreMRs(
                L4_Word_t i,
                L4_Word_t n,
                L4_Word_t data_ptr[]);
```

The L4_StoreMRs(*i*, *n*, *data_ptr*) function delivers the values of the virtual registers MessageData$_i$ to MessageData$_{i+n-1}$ inclusive in the array specified by the *data_ptr* argument. If the expression $i+n-1$ is greater than the MaxMessageData system parameter, the behaviour of this function is undefined.

See also: the MessageData virtual register on page 73, the L4_LoadMRs() function on page 247 and the L4_Word_t type on page 234.

D-16 The <l4/misc.h> Header

The <l4/misc.h> header provides a number of miscellaneous C definitions that do not belong in any other header of the language bindings library. It automatically includes the <l4/types.h> header, making available all types and constants defined in Chapter D-14.

D-16.1 Memory Attributes

The <l4/misc.h> header defines a set of macros representing the individual constructors of the OKL4 **CACHINGPOLICY** data type:

L4_DefaultMemory	Represents the DEFAULTPOLICY data constructor.
L4_CachedMemory	Represents the CACHEDPOLICY data constructor.
L4_UncachedMemory	Represents the UNCACHEDPOLICY data constructor.
L4_WriteBackMemory	Represents the WRITEBACKPOLICY data constructor.
L4_WriteThroughMemory	Represents the WRITETHROUGHPOLICY data constructor.
L4_CoherentMemory	Represents the COHERENTPOLICY data constructor.
L4_IOMemory	Represents the DEVICEPOLICY data constructor.
L4_IOCombinedMemory	Represents the WRITECOMBININGPOLICY data constructor.

See also: the **CACHINGPOLICY** data type on page 51 and section A-3.3 on page 32.

D-17 Summary of Compatibility Bindings

D-17.1 List of C Macros

D-17.2 List of C Types

D-17.3 List of C Functions