NICTA L4-embedded Kernel Reference Manual

Version NICTA N2

National ICT Australia
Embedded Real-Time and Operating Systems Program (ERTOS)
Kensington Research Laboratory, Sydney
l4spec@ertos.nicta.com.au

Based on Reference Manual for L4 X.2
System Architecture Group
Dept. of Computer Science
Universität Karlsruhe
(L4Ka Team)
l4spec@l4ka.org

Document Revision 2
May 15, 2006

Contents

About This Manual **v**
- Introductory Remarks . . . v
- Understanding This Document . . . vi
- Notation . . . vii
- Using the API . . . viii
- Revision History . . . ix

1 Basic Kernel Interface **1**
- 1.1 Kernel Interface Page . . . 2
- 1.2 KernelInterface . . . 7
- 1.3 Virtual Registers . . . 11

2 Threads **13**
- 2.1 ThreadId . . . 14
- 2.2 Thread Control Registers (TCRs) . . . 16
- 2.3 ExchangeRegisters . . . 19
- 2.4 ThreadControl . . . 23

3 Scheduling **27**
- 3.1 ThreadSwitch . . . 28
- 3.2 Schedule . . . 29
- 3.3 Preempt Flags . . . 32

4 Address Spaces and Mapping **33**
- 4.1 Fpage . . . 34
- 4.2 PhysDesc . . . 36
- 4.3 MapItem . . . 37
- 4.4 MapControl . . . 38
- 4.5 SpaceControl . . . 41

5 IPC **45**
- 5.1 Messages And Message Registers (MRs) . . . 46
- 5.2 IPC Control Registers (TCRs) . . . 49
- 5.3 Ipc . . . 51

6 Miscellaneous **59**
- 6.1 ExceptionHandler . . . 60
- 6.2 Cop Flags . . . 61
- 6.3 ProcessorControl . . . 62
- 6.4 CacheControl . . . 64

7 Protocols **67**
- 7.1 Thread Start Protocol . . . 68
- 7.2 Interrupt Protocol . . . 69
- 7.3 Pagefault Protocol . . . 70
- 7.4 Preemption Protocol . . . 71
- 7.5 Exception Protocol . . . 72
- 7.6 Asynchronous Notification Protocol . . . 73
- 7.7 Generic Booting . . . 74
- 7.8 Tracebuffer . . . 77

A IA-32 Interface **79**
A.1 Virtual Registers . . . 80
A.2 Systemcalls . . . 82
A.3 Kernel Features . . . 85
A.4 IO-Ports . . . 86
A.5 Space Control . . . 87
A.6 Memory Attributes . . . 88
A.7 Exception Message Format . . . 89
A.8 Processor Mirroring . . . 90
A.9 Booting . . . 91

B ARM Interface **93**
B.1 Virtual Registers . . . 94
B.2 Systemcalls . . . 96
B.3 Kernel Features . . . 99
B.4 Memory Attributes . . . 100
B.5 Space Control . . . 101
B.6 Exchange Registers . . . 102
B.7 Exception Message Format . . . 103
B.8 Thumb mode extensions . . . 105
B.9 Booting . . . 106

C MIPS-64 Interface **107**
C.1 Virtual Registers . . . 108
C.2 Systemcalls . . . 110
C.3 Memory Attributes . . . 114
C.4 Exception Message Format . . . 115
C.5 Exchange Registers . . . 117
C.6 Booting . . . 118

D PowerPC Interface **119**
D.1 Virtual Registers . . . 120
D.2 Systemcalls . . . 122
D.3 Memory Attributes . . . 126
D.4 Exception Message Format . . . 127
D.5 Processor Mirroring . . . 129
D.6 Booting . . . 130

E Generic BootInfo **131**
E.1 Generic BootInfo . . . 132
E.2 BootInfo Records . . . 134

F Development Remarks **137**
F.1 Exception Handling . . . 137

Table of Procs, Types, and Constants **139**

Index **145**

About This Manual

Introductory Remarks

Purpose of This Document

This L4 Reference Manual serves as defining document for all L4 APIs and ABIs. Primarily, it addresses L4 microkernel implementors as API/ABI suppliers and code-generator or library implementors as API/ABI users. The reference manual assumes intimate knowledge of basic L4 concepts and hardware architecture. Its key point is precise definition, not explanation and illustration. The

L4 User Manual

is intended to support programmers using L4. It explains and illustrates fundamental concepts and describes in more detail how (and why) to use which function, etc.

Maintainers

The document is maintained by the following members of the NICTA Team:

- Carl van Schaik (Carl.vanSchaik@nicta.com.au)

The document is based on the work of the L4Ka Team:

- Uwe Dannowski (ud3@ira.uka.de)
- Joshua LeVasseur (jtl@ira.uka.de)
- Espen Skoglund (esk@ira.uka.de)
- Volkmar Uhlig (volkmar@ira.uka.de)

Credits

This is subsequently based on a final draft by **Jochen Liedtke**. It reflects his outstanding work on the L4 microkernel and systems research in general. Only his vision of system design made this work possible. Jochen defined the state of the art of microkernel design for nearly a decade. We thank him for his support and try to continue the work in his spirit.

Helpful contributions for improving this reference manual and the L4 interface came from many persons, in particular 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, Harvey Tuch, Marcus Völp, Neal Walfield, Alex Webster, Adam Wiggins, Simon Winwood, and Jean Wolter.

Document History

draft by Jochen Liedtke	??/?? - 06/01
review by L4Ka Team	06/01 - 09/01
L4 developers review	Q4/01
release	01/02
NICTA L4-embedded	09/05

Understanding This Document

This L4 Reference Manual defines the generic API for all 32-bit and 64-bit machines. As such, the generic reference manual is independent of specific processor architectures. It is complemented by processor-specific ABI specifications. Some of them can be found in the appendix of this document.

In this document, we differentiate between *Logical Interface, Generic Binary Interface, Generic Programming Interface, Convenience Programming Interface* and *Processor-specific Binary Interface.*

Logical Interface The logical interface defines all concepts and logical objects such as system-call operations, logical data objects, data types and their semantics. Altogether, they form the logical L4 API.

Generic Binary Interface
Binary representations of most data types and generic data objects are defined independently of specific processors (although there are two different versions, one for 32-bit and a second one for 64-bit processors). Both versions together form the generic binary interface of L4.

From a purist point of view, logical interface plus generic binary interface could be regarded as a complete specification of the hardware-independent L4 microkernel interface. However, for ease-of-use and standardization reasons, the mentioned two fundamental interfaces are complemented by two more interface classes:

Generic Programming Interface
The generic programming interface defines the objects of the logical interface and the generic binary interface as pseudo C++ classes. The language bindings for regular C is for the most part identical to C++. For the cases where the C language causes function naming conflicts, the C version of the function name is given in brackets.
For the time being, only the C and C++ versions of the API are specified. The concrete syntax of other language interfaces will be left open. Later on, all language bindings will be included in the generic programming interface.

Convenience Programming Interface
This interface is not part of the L4 microkernel specification in the strict sense. All of its data types and procedures can be implemented using the generic programming interface. Strictly speaking, it is an interface on top of the microkernel that makes the most common operations more easily usable for the programmer.
It is important to understand that convenience and ease-of-use, not completeness, is the criterion for this interface. The convenience programming interface supports programmers by offering operations that together cover about 95% of the required microkernel functionality. For the remaining 5%, the programmer has to use the basic (not so convenient) operations of the generic programming interface.
Obviously, the convenience programming interface is not mandatory. Consequently, from a minimalist point of view, there is no need to include it in the generic L4 specification.

> *Nevertheless, for reasons of standardization and thus portability of software, every complete L4 language binding has to include the entire convenience programming interface.*

Implementation remark: Although the convenience interface *can* be completely implemented on top of the generic programming interface, i.e., processor independently, the implementor of the convenience interface *may* implement it hardware-dependently and thus incorporate any optimization that becomes possible through a specific processor-specific binary interface.

The last interface class is not part of the generic L4 API specification.

Processor-specific Binary Interface
: Defines the processor-specific binary interface.

Notation

Basic Data Types

This reference manual describes the L4 API and ABI for both 32-bit and 64-bit processors. The data type Word denotes a 32-bit unsigned integer on a 32-bit processor and a 64-bit unsigned integer on a 64-bit processor. Word64, Word32, and Word16 denote 64, 32, and 16-bit words independent of the processor type.

Privileged Threads

Some system calls can only be executed by privileged threads. Any thread belonging to the same address space as one of the initial threads created by the kernel upon boot-time (see page 74) are treated as privileged.

Bit Fields

Bit-field lengths are denoted as subscripts $_{(i/j)}$ where i relates to a 32-bit processor and j to a 64-bit processor. Bit-field subscripts $_{(i)}$ specify bit fields that have the same size for both 32-bit and 64-bit processors. Byte offsets are given as $\pm i \,/\, \pm j$ for 32-bit and 64-bit processors. If all bit-fields of a specified word only adds up to 32 bits, the remaining upper 32 bits on 64-bit processors are *undefined* or *ignored*.

Undefined, Ignored, and *Unchanged*

$\boxed{\sim}$
: Output parameters or bit fields can be *undefined*. Corresponding parameters or fields are denoted by $\sim$. They have no defined value on output, i.e., they may have any value or may even be unaccessible. Any algorithm relying on the value of undefined parameters or bit fields is defined to be incorrect.

$\boxed{-}$
: Input parameters or bit fields can be specified as *ignored*, denoted by –. Such parameters or fields can hold any value without affecting the invoked service. – is also used to define bit fields that are available for additional information. For example, fpage denotations contain some ignored bits that are used for access control bits in some system calls.

$\boxed{\equiv}$
: In processor-specific interfaces, registers are sometimes defined to be unchanged. This is denoted by $\equiv$.

Upward Compatibility

The following holds for future API versions and sub-versions that are specified as *upward-compatible* to the current version.

Output parameters and bit fields.
: Fields currently defined as undefined ($\sim$) may be specified as defined. Such newly defined fields will only deliver additional information. They can be ignored if the system call is used exactly like specified in the current API.

Input parameters and bit fields.

Fields currently defined as ignored (–) may be specified as defined. However, the content of such fields will be only relevant for newly defined features. Such fields will be ignored if a system call is used with the “old” semantics specified in this API.

Using the API

Naming

A programmer can use all function, type, and constant definitions defined in the generic and convenience programming interfaces throughout this manual. All definitions must, however, be prefixed with the string “L4_” and type names must contain the “_t” suffix (e.g., use “L4_Ipc()” and “L4_MsgTag_t” rather than “Ipc()” and “MsgTag”). The interfaces are currently only defined for C++ and C. In some cases the naming used for function names causes conflicts in the C language. These conflicts must be resolved using the alternative name specified in brackets after the function definition.

Include Files

The relevant include files containing the required definitions and declarations are specified in the beginning of the generic and convenience interface sections. In general there is one include file for each chapter in the manual. If only the basic L4 data types are needed they can be included using <l4/types.h>.

Revision History

L4Ka X.2

Revision 1

Initial revision.

Revision 2

- Clarified the specification of the kernel-interface page and kernel configuration page magic.
- UntypedWords and StringItems Acceptor constants collided with function UntypedWords(MsgTag) and StringItems(MsgTag) function declaration. Renamed to UntypedWordsAcceptor and StringItemsAcceptor.
- Changed kernel ids for L4Ka kernels.
- Fixed return types for operators on the Time type.
- Changed wrx access rights in fpages to rwx. Also changed WRX reference bits in fpages returned from *Unmap* system call to RWX.
- Renamed Put functions operating on MsgBuffer to Append.
- Address space deletion is now performed by deleting the last thread of an AS. This makes creation and deletion symmetrical (via ThreadControl). Before, all threads but the last were deleted by ThreadControl, and the last by SpaceControl.
- Added functions for creating ThreadIDs and for retrieving version and thread numbers from them. Fixed size of MyLocalId and MyGlobalId TCRs.
- Specified that the first three thread version numbers available for user threads are dedicated to σ_0, σ_1, and root task respectively.
- Changed the encoding of μ in the magic field of the KIP back to 0xE6 to be compatible with previous versions of the kernel.
- Changed memory descriptors (e.g., dedicated memory) in the kernel-interface page and kernel configuration page to use an array of typed descriptors instead of a static number of predefined ones.
- Added an appendix for the PowerPC interface.
- Added Niltag MsgTag constant.
- Decreased size of MsgBuffer structure to 32.
- Changed single Fpage& argument of Unmap() and Flush() into pass by value.
- Changed the ia32 kernel feature string “small” to “smallspaces”.
- Added appendix for the ia64 interface.
- Changed the ia32 IPC and LIPC ABI to be better suitable for common hardware featuring sysenter/sysexit and gcc.
- Added ProcDesc convenience functions.
- Specified which include files to use for the various parts of the API.
- Allow privileged threads to access ia32 Model-Specific Registers.

- Changed the ia64 ABI for system-call links and the IPC and LIPC system-calls.
- The UTCB location of a new thread is now explicitly specified by a parameter to the THREADCONTROL system-call.
- Added C versions of conflicting function names.
- Added a number of convenience functions for fpages, map items, grant items, string items and kernel interface page fields.
- Added description of the send base in map and grant items.
- Changed subversion numbering for Version X.2 and Version 4 API.
- Renamed the XferTimeout TCR to XferTimeouts and split into separate send and receive timeouts.
- Added two thread specific words to each the architecture specific TCR sections. These words are free to be used by, e.g., IDL compilers.
- Changed name of L4Ka kernels to the official name. Added L4Ka::Strawberry.
- Added appendices for Alpha and MIPS64.

Revision 3

- Clarified description of the *supplier* field in the kernel-interface page.
- Added NumMemoryDescriptors() convenience function.
- Clarified the return value of MemoryDescType() function.
- Fixed faulty specification of Wait_Timeout() and ReplyWait_Timeout().
- Added a new h-flag to *control* parameter in the EXCHANGEREGISTERS system-call. The h-flag controls whether the resume/halt flag should be ignored or not.
- Changed parameter type of TimePeriod() from "int" to "Word64".
- Fixed typo in specification of the MsgTag input/output IPC parameter.
- Added comment to IPC system-call about the read-once semantics of message registers.
- Added member name "raw" to all L4 types declared as structs.
- Renamed start() and stop() functions to Start() and Stop().
- Describe semantics of undefined UTCB memory regions.
- The first 10 message registers on PowerPC are now defined as backed by physical registers.
- The first 9 message registers on Alpha are now defined as backed by physical registers.
- Fixed MR_0 register allocation for IA32 syscalls and adapted syscalls accordingly.

Revision 4

- Added appendix for AMD64.
- Changed MIPS64 IPC ABI to include 9 message registers.
- Added SYSTEMCLOCK syscall for MIPS64.
- Clarified the fact that an interrupt thread may be the originator thread during IPC propagation.
- Added appendix for SPARC v9.
- The *high* field of memory descriptors now specify the last addressable byte in the memory region.

Revision 5

- The ErrorCode TCR is now a generic placeholder for error descriptions of failed system-calls.
- *MemoryControl* now returns a result parameter.
- Defined error codes for various system-calls (EXCHANGEREGISTERS, THREADCONTROL, SCHEDULE, SPACECONTROL, PROCESSORCONTROL and *MemoryControl*).
- Defined convenience definitions for error code values.
- Changed the IA32 SYSTEMCLOCK ABI to clobber the EDI register.
- Specify that the KIP area and the UTCB area of an address space must not overlap.
- For the PowerPC system call trap exception IPC, use a message label of -5, and preserve register LR.
- The EXCHANGEREGISTERS system-call can no longer activate an inactive thread.
- The Fpage argument to Set_Rights() is now passed by reference.
- Fixed inconsistencies about the number of available buffer registers.
- Renamed Void to void, Char to char, and bool to Bool.
- The Start() convenience function now aborts any ongoing IPC operations.
- The Unmap() and Flush() convenience functions operating on a single fpage now deliver the status bits of the modified fpage.
- MIPS64 now uses the k0 ($26) register for holding the UTCB address.
- Added two new memory types for *MemoryControl* on MIPS64.
- Added appendix for generic BootInfo.
- Make it clear that it is not possible to activate a thread in an address space which has not been properly configured with SPACECONTROL.
- Added appendix for ARM.
- If using a 64 bit kernel, define second 32 bit word of kernel interface page to 0.
- Changed the ABI for the PowerPC system calls *Unmap* and *MemoryControl*.

Revision 6

- Removed *control* parameter from PROCESSORCONTROL system call binding and from the PROCESSORCONTROL Alpha system call ABI.
- Added delivery parameter to EXCHANGEREGISTERS controlling whether the syscall should deliver the thread's old values or not. Targeted at MP systems.
- Added operators for adding and subtracting two Clock values.
- Specified that σ_0 also understands the pagefault protocol, and that anonymous σ_0 requests will only regard conventional memory as available.
- Added ARM general exception IPC message format
- Changes MIPS64 syscall exception IPC message format to closer match the general exception message format

NICTA N1

Revision 1

This version of the specification is characterized by the following main changes.

- Removal of Long IPC (string copy).
- Added Async Notification.
- Removed timeouts and SYSTEMCLOCK syscall.
- Provide redirectors on a per thread basis.
- Provide fewer message registers.

Detailed changes.

- Started NICTA N1 version.
- Removed SYSTEMCLOCK syscall.
- Added API Version $0x86$ as NICTA Experimental.
- ReadPrecision of *ClockInfo* field in KIP undefined.
- Defined UTCB and KIP info in KIP to allow non-user controlled areas.
- Added 'NICT' kernel supplier ID.
- Modified ClockInfo to contain only *SchedulePrecision()*.
- Removed *ReadPrecision()* convenience function.
- *SchedulePrecision()* description.
- Added *VirtualRegsInfo* field in KIP.
- Removed *Buffer registers*.
- Added *NotifyMask*, *NotifyBits*, *Acceptor*, *Preempted IP* and *PreemptCallback IP* to TCRs.
- Removed *XferTimeouts* from TCRs.
- Added new access function for new TCR fields, removed *XferTimeouts*.
- Added *from*, *nv* bits to EXCHANGEREGISTERS *control* word.
- Added Copy_XXX_regs convenience functions for EXCHANGEREGISTERS .
- Added *SendRedirector* and *ReceiveRedirector* arguments and descriptions to THREADCONTROL .
- Added remark about *UtcbLocation* for ARM in THREADCONTROL .
- Added error code 9 *ErrInvalidRedirector* for THREADCONTROL .
- Removed sections *Clock*, SYSTEMCLOCK and *Time* from chapter Scheduling.
- Removed argument *time control* from SCHEDULE syscall.
- Change argument *preemption control* to *not used* in SCHEDULE .
- Added *TimeControl* values which are passed for SCHEDULE .
- Modified *Timeslice()* and *SetTimeslice* convenience functions.
- Removed *Id* bits from *PreemptFlags*.
- Changed functionality of *s* bit in *PreemptFlags*.

- Remove *EnablePreemptionFaultException()*, *DisablePreemptionFaultException()*, *DisablePreemption()*, *EnablePreemption()* and *PreemptionPending()* functions.
- Add *EnablePreemprionCallback()*, *DisablePreemptionCallback()*, *PreemeptedIP()* and *Set_PreemptCallbackIP()* functions.
- Removed *Redirector* argument from SPACECONTROL .
- Added comment about ARM *KernelInferfacePageArea* and *UtcbArea* for SPACECONTROL .
- Changed number of Message Registers to be architecture defined and indicated in KIP.
- Updated description of *u* bit in *MsgTag* to cover case where number of untyped word exceeds number of message registers.
- Removed String IPC.
- Reserved typed-items previously describing StringItems.
- Updated message registers convenience functions - removed StringItems.
- Removed *StringItem* and *String Buffers And Buffer Registers* sections.
- Removed 'C' bit from typed messages.
- Added section *IPC Control Registers*.
- Removed *Timeouts* field from IPC syscall.
- Updated description of IPC to include *Asynchronous notification* and to remove *Timeouts*. Timeouts replaced with *blocking / non-blocking* semantics.
- Updated description of LIPC .
- Modified *MsgTag* to include *a* - asynchronous notification, *r* - receive block and *s* send block operation.
- Removed description of *XferTimeouts* TCR from IPC .
- Modified *ErrorCode* in IPC to have a 4-bit error value. Removed *offset* field.
- Removed section on Pagefaults in IPC .
- Added *AsynchIpc()* and *WaitAsynch()* programming interface functions for IPC .
- Updated all Convenience Programming Interface functions for new IPC syscall functionality.
- Remove reference to BR0 from *ExceptionHandler*.
- Change acceptor from BR0 to TCR in *Pagefault Protocol*.
- Remove clock payload from Preemption Protocol and change description.
- Change description of Dedicated memory to "device memory".
- Add Acceptor, NotifyBits, Notify mask to ia32,ARM,mips64 TCRs.
- Remove Buffer Registers from ia32,ARM,mips64 architectures.
- Remove SYSTEMCLOCK syscall from ia32,ARM,mips64 architectures.
- Add SendRedirector and ReceiveRedirector from THREADCONTROL in ia32, ARM, mips64 architectures.
- Remove *time control* argument from SCHEDULE in ia32, ARM, mips64 architectures.
- Remove *Timeouts* argument from IPC and LIPC in ia32, ARM, mips64 architectures.
- Remove *Redirector* argument from SPACECONTROL in ia32, ARM, mips64 architectures.
- Add *ts len / total quantum* arguments to SCHEDULE in ia32, ARM, mips64 architectures.
- Add *Exchange Registers* section to mips64 and ARM architectures.
- Rearrange ARM UTCB layout.
- Fix ARM/MIPS64 utcb location details.

- Add extra fields in ARM section *Memory Attributes*.
- Add *vspace* extension for SPACECONTROL on ARM.
- Rearrange ARM exception message format.
- Add *Thumb mode extensions* section for ARM architecture.

Revision 2

- Fix mips64 IPC and LIPC calls.
- Fix unknown link in tex file.

Revision 3

- Add tracebuffer specification.
- Fixed schedule system call arguments + arm/ia32/mip64 definitions
- Fixed threadcontrol system call arguments for ia32
- Fixed discussion of *FromSpecifier* in Asynchronous notification.
- Fix input parameters *to* field description for Asynchronous notfication.
- Changed IPC input parameter: *MsgTag*: ”ignored if no send phase” to ”ignored if *to* =*nilthread*”
- Fix IPC error codes.
- Fix KIP convienence functions ”VirtualRegisters” should have been ”MessageRegisters”.

Revision 4

- Changed argument *FpageSize* of function *Fpage* from *int* to *word*.
- Update description of IPC to describe what happens when more MRs are specified than supported.
- Changed asynchronous notification IPC to be more general. Allow send and receive phases.
- Added asynchronous notification protocol section.
- Added *Notify* and *WaitNotify* IPC helper functions.
- Changed *Set_Asynch* to *Set_Notify*.
- Reordered ARM and IA32 UTCB layout.
- Added note to ARM that KIP code does not modify the user stack pointer.
- Added note that the reply to exception IPC format is the same as the corresponding exception IPC format if not specified.
- Added tracebuffer protocol.
- Added the *waitnotify* special threadid.
- Added note that pending IPCs to a deleted thread are cancelled/aborted.
- Added asynchronous notification index entries.

Revision 5

- Remove local-id support.
- Modified Exchange registers to allow root threads to call Exregs on any thread.
- Remove local-id from exception messages ARM and MIP64 and update message count.
- Add descriptions for TCRs in thread control registers.

Revision 6

- Fix mips64 message register table.
- Remove local-id references in mips64
- Added powerpc N1 API

NICTA N2

Revision 1

- Cleanup fpage descriptions, add clearer definitions.
- Add *PhysDesc* data type.
- Add *MapItem* data type (redefined for MapControl).
- Added MapControl system call.
- Removed Unmap system call.
- Fix number of bits for KipArea on 64-bit machines.
- Removed *MapItem* and *GrantItem* from IPC.
- Removed *Typed* items from IPC.
- Reordered system calls in KIP for N2 API.
- Removed *MemoryControl* syscall.
- Changed MIPS64 IPC return code from 'result' to 'from'.
- Removed references to sigma0 / sigma1.
- Removed *SharedMemoryType*, added *NUMAMemoryType* and *GlobalMemoryType*.
- Changes AsynchItemAcceptor to NotifyMsgAcceptor.
- Added explicit Tag field to IPC convienence functions.
- A number of typo fixes and clarifications.
- Fix IPC() convienence function argument order. *GlobalMemoryType*.

Revision 1

- Added *CacheControl* system call.
- Clarify IPC descriptions.

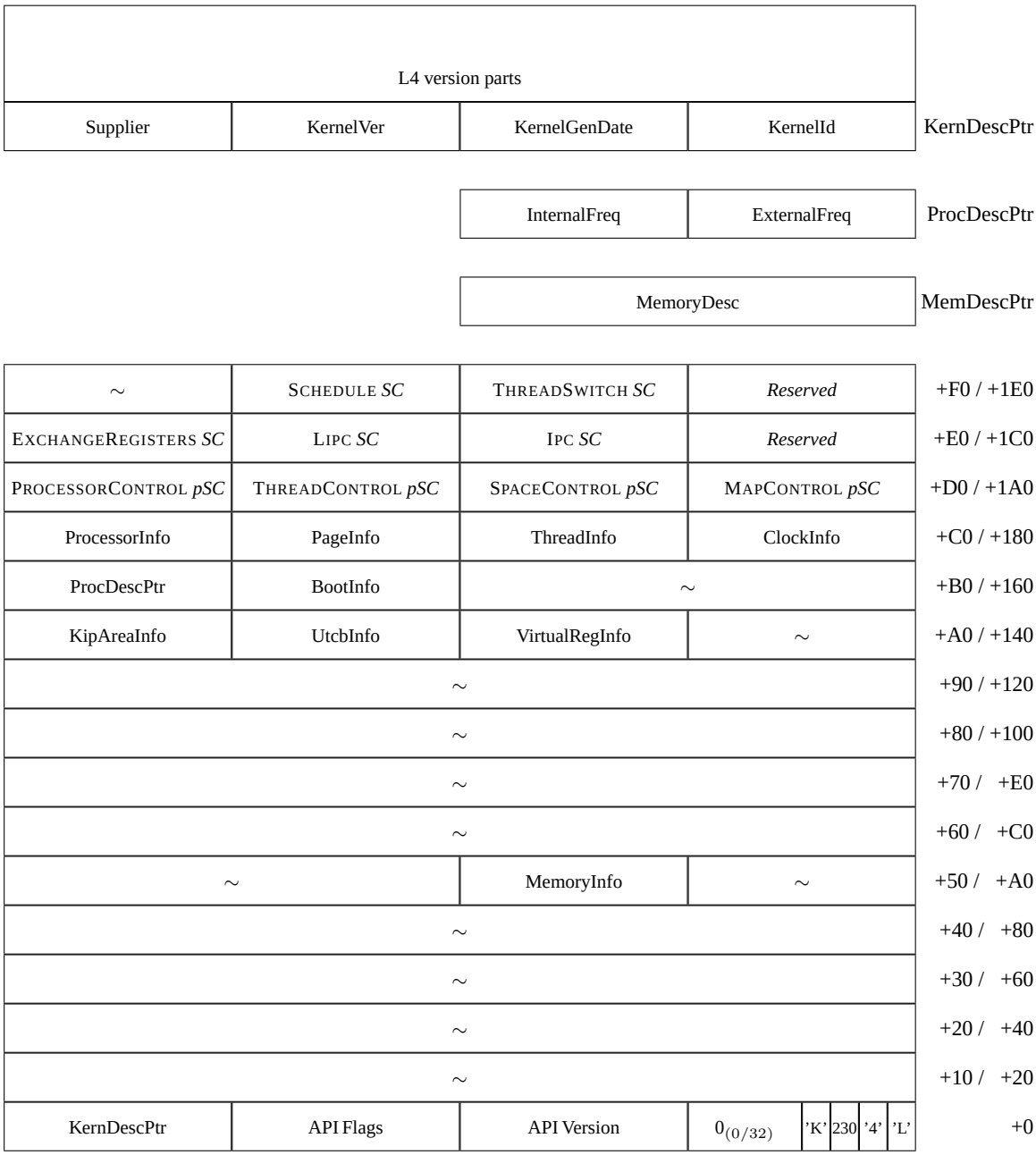
Chapter 1

Basic Kernel Interface

1.1 Kernel Interface Page [Data Structure]

The kernel-interface page contains API and kernel version data, system descriptors including memory descriptors, and system-call links. The remainder of the page is undefined.

The page is a microkernel object. It is directly mapped through the microkernel into each address space upon address-space creation. It is *not* mapped by a pager, can *not* be mapped to another address space and can *not* be unmapped. The creator of a new address space can specify the address where the kernel interface page has to be mapped. This address will remain constant through the lifetime of that address space. Any thread can obtain the address of the kernel interface page through the KERNELINTERFACE system call (see page 7).

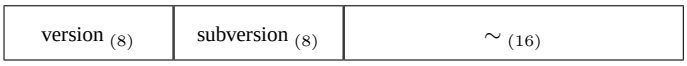


Note that this kernel interface page is basically upward compatible to the *kernel info page* of versions 2 and X.0. The magic byte string "L4μK" at the beginning of the object identifies the kernel interface page.

Version/id number convention: Version/subversion numbers and id/subid numbers with the most significant bit 0 denote official versions/ids and are globally unique through all suppliers. Version/id numbers that have the most significant bit set to 1 denote experimental versions/ids and may be unique only in the context of a supplier.

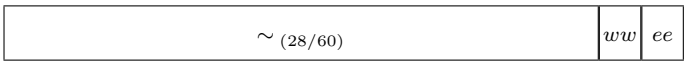
API Description

API Version



version	subversion	
0x02		Version 2
0x83	0x80	Experimental Version X.0
0x83	0x81	Experimental Version X.1
0x84	*rev*	Experimental Version X.2 (Revision *rev*)
0x85	*rev*	Dresden
0x86	*rev*	NICTA N1 (Revision *rev*)
0x04	*rev*	Version 4 (Revision *rev*)

API Flags



ee = 00 : little endian,
= 01 : big endian.

ww = 00 : 32-bit API,
= 01 : 64-bit API.

Note that this field can not be used directly to differentiate between little endian and big endian mode since the ee field resides in different bytes for both modes. Furthermore, the offset address of the API Flags is different for 32-bit and 64-bit modes. In summary, a direct inspection of the kernel interface page is not sufficient to securely differentiate between 32/64-bit modes and little/big endian modes.
Secure mode detection is enabled through the KERNELINTERFACE system call (see page 7). It delivers the API Flags in a register.

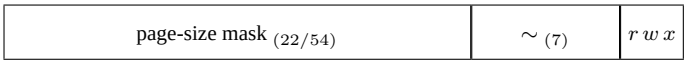
System Description

ProcessorInfo

s The size of the area occupied by a single processor description is 2^s. Location of description fields for the first processor is denoted by *ProcDescPtr*. Description fields for subsequent processors are located directly following the previous one.

$processors$
Number of available system processors.

PageInfo



page-size mask
If bit $k-10$ of the page-size mask field (bit k of the entire word) is set to 1 hardware and kernel support pages of size 2^k. If the bit is 0 hardware and/or kernel do not support pages of size 2^k. Note that fpages of size 2^k *can* be used, even if 2^k is no supported hardware page size. Information about supported hardware page sizes is only a performance hint.

$r\,w\,x$ Identifies the supported access rights ($r$ead, $w$rite, e$x$ecute) that can be set independently of other access rights. A 1-bit signals that the right can be set and reset on a mapped page. For $rwx = 010$, only write permission could be controlled orthogonally. The processor would implicitly permit read and execute access on any mapped page. For $rwx = 111$, all three rights could be set and reset independently.

ThreadInfo

$UserBase$ (12)	$SystemBase$ (12)	t (8)

t Number of valid thread-number bits. The thread number field may be larger but only bits $0 \ldots t-1$ are significant for this kernel. Higher bits must all be 0.

$UserBase$
Lowest thread number available for user threads (see page 14). The first thread number will be used for the initial thread of root task (see page 74). The version numbers (see page 14) for any initial threads will be equal to one.

$SystemBase$
Lowest thread number used for system threads (see page 14). Thread numbers below this value denote hardware interrupts.

ClockInfo

$\sim$ (0/32)	SchedulePrecision (32)

SchedulePrecision
Specifies the maximal jitter ($\pm$) for a scheduled thread activation based on a wakeup time (provided that no thread of higher or equal priority is active and timer interrupts are enabled). Precisions are given in microseconds.

UtcbInfo

$\sim$ (10/42)	s (6)	a (6)	m (10)

s The minimal *area size* for an address space's UTCB area is 2^s. The size of the UTCB area limits the total number of threads k to $2^a mk \leq 2^s$. A size of 0 indicates that the UTCB is not part of the user address space and cannot be controlled (see page 41).

m UTCB size multiplier.

a The UTCB location must be aligned to 2^a. The total size required for one UTCB is $2^a m$.

VirtualRegInfo

$\sim$ (26/58)	$n-1$ (6)

n The number of message registers supported by the kernel.

KipAreaInfo

$\sim$ (26/58)	s (6)

s The size of the kernel interface page area for an address space is 2^s. A size of 0 indicates that the KIP is not part of the user address space and cannot be controlled (see page 41).

BootInfo Prior to kernel initialization a boot loader can write an arbitrary value into the BootInfo field of the kernel configuration page (see page 74). Post-initialization code, e.g., a root server can later read the field from the kernel interface page. Its value is neither changed nor interpreted by the kernel. This is a generic method for passing system information across kernel initialization.

Processor Description

ProcDescPtr Points to an array containing a description for each system processor. The *ProcessorInfo* field contains the dimension of the array. *ProcDescPtr* is given as an address relative to the kernel interface page's base address.

ExternalFreq External Bus frequency in kHz.

InternalFreq Internal processor frequency in kHz.

Kernel Description

KernDescPtr Points to a region that contains 4 kernel-version words (see below) followed by a number of 0-terminated plain-text strings. The first plain-text string identifies the current kernel followed by further optional kernel-specific versioning information. The remaining plain-text strings identify architecture dependent kernel features (see architecture specific *Kernel Features* section). A zero length string (i.e., a string containing only a NUL-character ('\0')) terminates the list of feature descriptions.
KernelDescPtr is given as an address relative to the kernel interface page's base address.

KernelId

id $_{(8)}$	subid $_{(8)}$	$\sim_{(16)}$

Can be used to identify the microkernel.

id	subid	kernel	supplier
0	1	L4/486	GMD
0	2	L4/Pentium	IBM
0	3	L4/x86	UKa
1	1	L4/Mips	UNSW
2	1	L4/Alpha	TUD, UNSW
3	1	Fiasco	TUD
4	1	L4Ka::Hazelnut	UKa
4	2	L4Ka::Pistachio	UKa, UNSW, NICT
4	3	L4Ka::Strawberry	UKa
5	1	NICTA::Pistachio-embedded N1	NICT
5	2	NICTA::Pistachio-embedded N2	NICT

KernelGenDate

$\sim_{(16/48)}$	year-2000 $_{(7)}$	month $_{(4)}$	day $_{(5)}$

Kernel generation date.

KernelVer

ver $_{(8)}$	subver $_{(8)}$	subsubver $_{(16)}$

Can be used to identify the microkernel version. Note that this kernel version is not necessarily related to the API version.

Supplier The four least significant bytes of the *supplier* field specify a character string identifying the kernel supplier:

"GMD␣"	GMD
"IBM␣"	IBM Research
"UNSW"	University of New South Wales, Sydney
"TUD␣"	Technische Universität Dresden
"UKa␣"	Universität Karlsruhe (TH)
"NICT"	National ICT Australia (NICTA)

System-Call Links

SC Link for normal system call.

pSC Link for privileged system call, i.e., a system call that can only be performed by a privileged thread.

The system-call links specify how the application can invoke system-calls for the current microkernel. The interpretation of the system-call links is ABI specific, but will typically be addresses relative to the kernel interface page's base address where kernel provided system-call stubs are located.

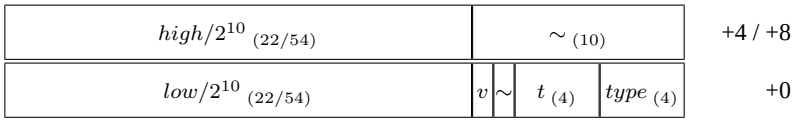
Memory Description

MemoryInfo

$MemDescPtr$
Location of first memory descriptor (as an offset relative to the kernel-interface page's base address). Subsequent memory descriptors are located directly following the first one. For memory descriptors that specify overlapping memory regions, later descriptors take precedence over earlier ones.

n Number of memory descriptors.

MemoryDesc



$high$ Address of last byte in memory region. The ten least significant address bits are all hardwired to 1.

low Address of first byte in memory region. The ten least significant address bits are all hardwired to 0.

v Indicates whether memory descriptor refers to physical memory ($v = 0$) or virtual memory ($v = 1$).

$type$ Identifies the type of the memory descriptor.

Type	Description
0x0	Undefined
0x1	Conventional memory
0x2	Reserved memory (i.e., reserved by kernel)
0x3	Dedicated memory (i.e., device memory)
0x4	NUMA memory (i.e., shared across a NUMA machine)
0x5	Global memory (i.e., always accessible)
0xB	Tracebuffer memory
0xE	Defined by boot loader
0xF	Architecture dependent

$t,\ type = 0xE$
The type of the memory descriptor is dependent on the bootloader. The t field specifies the exact semantics. Refer to boot loader specification for more info.

$t,\ type = 0xF$
The type of the memory descriptor is architecture dependent. The t field specifies the exact semantics. Refer to architecture specific part for more info.

$t,\ type \neq 0xE,\ type \neq 0xF$
The type of the memory descriptor is solely defined by the $type$ field. The content of the t field is undefined.

1.2 KERNELINTERFACE [Slow Systemcall]

⟶	*void**	*kernel interface page*
	Word	*API Version*
	Word	*API Flags*
	Word	*KernelId*

Delivers base address of the *kernel interface page*, *API version*, and *API flags*. The latter two values are copies of the corresponding fields in the kernel interface page. The API information is delivered in registers through this system call (a) to enable unrestricted structural changes of the kernel interface page in future versions, and (b) to enable secure detection of the kernel's endian mode (little/big) and word width (32/64).

The structure of the *kernel interface page* is described on page 2. The page is a microkernel object. It is directly mapped through the microkernel into each address space upon address-space creation. It is *not* mapped by a pager, can *not* be mapped to another address space and can *not* be unmapped. The creator of a new address space can specify the address where the kernel interface page has to be mapped. This address will remain constant through the lifetime of that address space.

Any thread can determine the address of the kernel interface page through this system call. Since the system call may be slow it is highly recommended to store the address in a static variable for further use.

It is also possible to use a unique address for the kernel interface page in all address spaces of a (sub)system. Then, the kernel interface page can be accessed by fixed absolute addresses without using the current system call.

Besides other things, the page describes the current API, ABI, and microkernel version so that a server or an application can find out whether and how it can run on the current microkernel. Since the kernel interface page also contains API- and ABI-specific data for most other system calls the page's base address is typically required before any other system call can be used.

To enable version detection independently of the API and ABI, the current system call is guaranteed to work in all L4 versions. The systemcall code will never change and will be the same on compatible processors. (If a processor is upward compatible to multiple incompatible processors the kernel should offer multiple systemcall codes for this function.)

Output Parameters

---

kernel interface page

Ver X.1 and above

base address $_{(32/64)}$

Kernel interface page address, always page aligned. 0 is no valid address.

Ver X.0 and below

0 $_{(32/64)}$

Older versions (2, X.0, etc.) do not include the kernel interface page as a kernel mapped page. No address is delivered.

API Version

version $_{(8)}$	subversion $_{(8)}$	$\sim$ $_{(16)}$

see page 3, "Kernel Interface Page"

API Flags

$\sim$ $_{(28/60)}$	ww	ee

see page 3, "Kernel Interface Page"

KernelId

id (8)	subid (8)	~ (16)

see page 5, “Kernel Interface Page”

Pagefaults

No pagefaults will happen.

Generic Programming Interface

System-Call Function:

#include <l4/kip.h>

*void ** ***KernelInterface*** (*Word& ApiVersion, ApiFlags, KernelId*)

Convenience Programming Interface

Derived Functions:

#include <l4/kip.h>

struct **MEMORYDESC** { Word raw[2] }

struct **PROCDESC** { Word raw[4] }

*void** ***KernelInterface*** () [*GetKernelInterface*]
Delivers a pointer to the kernel interface page.

Word ***ApiVersion*** ()

Word ***ApiFlags*** ()

Word ***KernelId*** ()

void ***KernelGenDate*** (*void* KernelInterface, Word& year, month, day*)

Word ***KernelVersion*** (*void* KernelInterface*)

Word ***KernelSupplier*** (*void* KernelInterface*)
Delivers the API Version/API Flags/Kernel Id/kernel generation date/kernel version/kernel supplier.

Word ***NumProcessors*** (*void* KernelInterface*)

Word ***NumMemoryDescriptors*** (*void* KernelInterface*)
Delivers number of processors in the system/number of memory descriptors in the kernel-interface page.

Word ***PageSizeMask*** (*void* KernelInterface*)

Word ***PageRights*** (*void* KernelInterface*)
Delivers supported page sizes/page rights for the current kernel/hardware architecture.

Word ***ThreadIdBits*** (*void* KernelInterface*)

Word ***ThreadIdSystemBase*** (*void* KernelInterface*)

Word ***ThreadIdUserBase*** (*void* KernelInterface*)
Delivers number of valid bits for thread numbers/lowest thread number for system threads/lowest thread number for user threads.

Word ***SchedulePrecision*** (*void* KernelInterface*)
Delivers the maximal jitter for wakeups (in μs).

Word ***MessageRegisters*** (*void* KernelInterface*)
Delivers the number of message registers supported by the kernel.

Word ***UtcbAreaSizeLog2*** (*void* KernelInterface*)

Word ***UtcbAlignmentLog2*** (*void* KernelInterface*)

Word ***UtcbSize*** (*void* KernelInterface*)
Delivers required minimum size of UTCB area/alignment requirement for UTCBs/size of a single UTCB.

Word ***KipAreaSizeLog2*** (*void* KernelInterface*)
Delivers size of kernel interface page area.

Word ***BootInfo*** (*void* KernelInterface*)
Delivers the contents of the boot info field.

*char** ***KernelVersionString*** (*void* KernelInterface*)
Delivers the kernel version string.

*char** ***Feature*** (*void* KernelInterface, Word num*)
Delivers the numth kernel feature string, or a null pointer if num exceeds the number of available feature strings.

*MemoryDesc** ***MemoryDesc*** (*void* KernelInterface, Word num*)
Delivers the numth memory descriptor, or a null pointer if num exceeds the number of available descriptors.

*ProcDesc** ***ProcDesc*** (*void* KernelInterface, Word num*)
Delivers the numth processor descriptor, or a null pointer if num exceeds the number of processors of the system (see ProcessorInfo).

Support Functions:

#include <l4/kip.h>

Word ***UndefinedMemoryType***

Word ***ConventionalMemoryType***

Word ***ReservedMemoryType***

Word ***DedicatedMemoryType***

Word ***NUMAMemoryType***

Word ***GlobalMemoryType***

Word ***TracebufferMemoryType***

Word ***BootLoaderSpecificMemoryType***

Word ***ArchitectureSpecificMemoryType***

Bool ***IsVirtual*** (*MemoryDesc& m*) [*IsMemoryDescVirtual*]
Delivers true if memory descriptor specifies a virtual memory region.

Word ***Type*** (*MemoryDesc& m*) [*MemoryDescType*]

Word ***Low*** (*MemoryDesc& m*) [*MemoryDescLow*]

Word ***High*** (*MemoryDesc& m*) [*MemoryDescHigh*]
Delivers type ($t*16+type$), low limit, and high limit of memory region.

Word ***ExternalFreq*** (*ProcDesc& p*) [*ProcDescExternalFreq*]

Word ***InternalFreq*** (*ProcDesc& p*) [*ProcDescInternalFreq*]
Delivers external frequency/internal frequency of processor.

1.3 Virtual Registers [Virtual Registers]

Virtual registers are implemented by the microkernel. They offer a fast interface to exchange data between the microkernel and user threads. Virtual registers are *registers* in the sense that they are static per-thread objects. Dependent on the specific processor type, they can be mapped to hardware registers or to memory locations. Mixtures, some virtual registers to hardware registers, some to memory are also possible. The ABI for virtual-register access depends on the specific processor type and on the virtual-register type, see architecture specific *Virtual Registers* section for specific hardware details.

There are two classes of virtual registers:

- *Thread Control Registers (TCRs),* see page 16
- *Message Registers (MRs),* see page 46

Loading illegal values into virtual registers, overwriting read-only virtual registers, or accessing virtual registers of other threads in the same address space (which may be physically possible if some are mapped to memory locations) is illegal and can have undefined effects on all threads of the current address space. However, since virtual registers can *not* be accessed across address spaces, they are safe from the kernel's point of view: Illegal accesses can like any other programming bug only compromise the originator's address space.

Remark: In general, virtual registers can only be addressed directly, not indirectly through pointers. The generic API therefore offers no operations for indirect virtual-register access. However, processor-specific code generators might use indirect access techniques if the ABI permits it.

VirtualRegInfo ***[KernelInterfacePage Field]***

Defines information relating to the kernel virtual register implementation.

$\sim_{(26/58)}$	$n-1_{(6)}$

n The number of message registers supported by the kernel.

Remark: This kernel specification is designed for embedded systems that are normally very configurable and inherently application specific. Thus it is a valid assumption for the application to halt if it detects insufficient message registers supported by the kernel.

Generic Programming Interface

#include <l4/message.h>

void ***StoreMR*** (*int* i, *Word&* w)

void ***LoadMR*** (*int* i, *Word* w)

Delivers/sets MR $_i$.

void ***StoreMRs*** (*int* i, k, *Word& [*k*]* w)

void ***LoadMRs*** (*int* i, k, *Word& [*k*]* w)

Stores/loads MR $_{i \ldots i+k-1}$ to/from memory.

Chapter 2

Threads

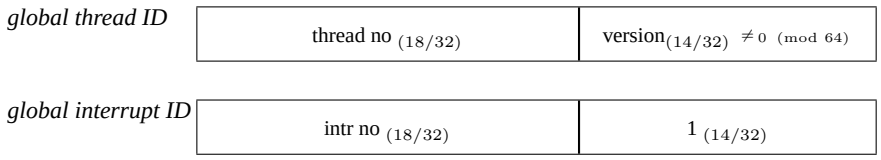
2.1 ThreadId [Data Type]

Thread IDs identify threads and hardware interrupts. All thread IDs are *global* and are unique through the entire system. They identify threads independently of the address space in which they are used.

Global Thread ID

A global thread ID consists of a word, where 18 bits (32-bit processor) or 32 bits (64-bit processor) determine the thread number and 14 bits (32-bit processor) or 32 bits (64-bit processor) are available for a version number. The thread number may not be all ones (unsigned -1).

User-thread numbers can be freely allocated within the interval $[\mathit{UserBase}\,, 2^t)$, where t denotes the upper limit of thread IDs. The thread-number interval $[\mathit{SystemBase}\,, \mathit{UserBase})$ is reserved for L4-internal threads. Hardware interrupts are regarded as hardware-implemented threads. Consequently, they are identified by thread IDs. Their corresponding thread numbers are within the interval $[0\,, \mathrm{SystemBase})$. The values *SystemBase*, *UserBase*, and t are published in the kernel interface page (see page 4).

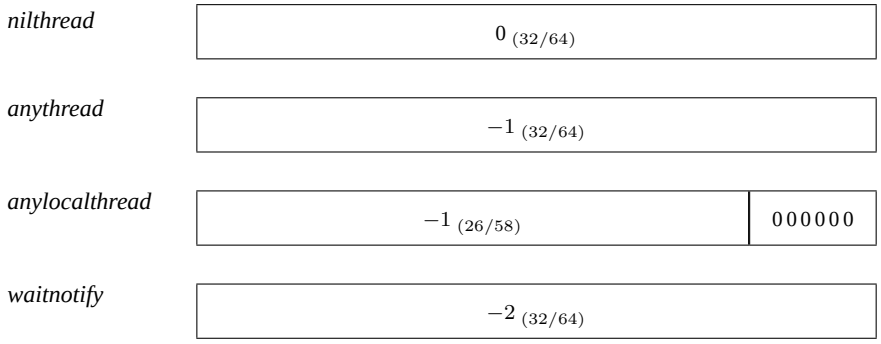


Global thread IDs have a version field whose content can be freely set by those threads that can create and delete threads.

The microkernel checks version fields whenever a thread is accessed through its global thread ID. However, the semantics of the version field are not defined by the microkernel. OS personalities are free to use this field for any purpose. For example, they may use it to make thread IDs unique in time.

Special Thread IDs

Special IDs exist for *nilthread* and three wild cards. The thread ID *anythread* matches with any given thread ID, including all interrupt IDs. The ID *anylocalthread* matches all threads that reside in the same address space. The ID *waitnotify* specifies an endpoint for waiting on asynchronous notifications only.



Generic Programming Interface

```
#include <l4/thread.h>

struct THREADID { Word raw }
```

ThreadId ***nilthread***

ThreadId ***anythread***

ThreadId ***anylocalthread***

ThreadId ***waitnotify***

ThreadId ***GlobalId*** (*Word threadno, version*)
Delivers a thread ID with indicated thread and version number.

Word ***Version*** (*ThreadId t*)

Word ***ThreadNo*** (*ThreadId t*)
Delivers version/thread number of indicated global thread ID.

Convenience Programming Interface

#include <l4/thread.h>

Bool == (*ThreadId l, r*) [*IsThreadEqual*]

Bool != (*ThreadId l, r*) [*IsThreadNotEqual*]
Check if thread IDs match or differ.

Bool ***IsNilThread*** (*ThreadId t*)
{ t == *nilthread* }

ThreadId ***MyGlobalId*** ()
Delivers the global ID of the currently running thread (see TCRs, page 16).

ThreadId ***Myself*** ()
{ MyGlobalId () }

2.2 Thread Control Registers (TCRs) [Virtual Registers]

TCRs are a fast mechanism to exchange relatively static control information between user thread and microkernel. TCRs are static non-transient per-thread registers.

MyGlobalId Global ID of the thread.

ProcessorNo The processor number on which the thread currently executes.

UserDefinedHandle
This field can be freely set and read by user threads. It can, e.g., be used for storing a thread number, a pointer to an additional user thread control block, etc.

Pager Pager of the thread.

ExceptionHandler The thread's exception handler.

Cop Flags Coprocessor flags.

Preempt Flags Preemption flags.

ErrorCode System call error code.

IntendedReceiver The intended destination of a redirected IPC.

VirtualSender/ActualSender
The actual sender of a propegated IPC or the thread to send as in deceiving IPC.

PreemptCallbackIP
Address to which a thread's program counter should be reset when preempted with Preemption-Callback enabled.

PreemptedIP Address of the program counter at which a thread was preempted when Preemption-Callback is enabled.

Acceptor Restricts or allows certain types of IPC messages to the thread.

NotifyBits Bitmask of received notification bits.

NotifyMask Mask of which notification bits are accepted in an IPC asynchronous notification receive operation.

Generic Programming Interface

The listed generic functions permit user code to access TCRs independently of the processor-specific TCR model. All functions are user-level functions; the microkernel is not involved.

```
#include <l4/thread.h>
```

ThreadId ***MyGlobalId*** ()
Delivers the global ID of the currently running thread (see TCRs, page 16).

ThreadId ***Myself*** ()
{ MyGlobalId () }

Word ***ProcessorNo*** ()
Delivers the processor number the current thread is running on. Delivered value is a valid index into the processor description array (see Kernel Interface Page, page 4).

Word ***UserDefinedHandle*** ()

void ***Set_UserDefinedHandle*** (*Word NewValue*)
Delivers/sets the user defined handle of the currently running thread.

ThreadId ***Pager*** ()

void ***Set_Pager*** (*ThreadId NewPager*)
Delivers/sets the pager for the currently running thread.

ThreadId ***ExceptionHandler*** ()

void ***Set_ExceptionHandler*** (*ThreadId NewHandler*)
Delivers/sets the exception handler for the currently running thread.

void ***Set_CopFlag*** (*Word n*)

void ***Clr_CopFlag*** (*Word n*)
Sets/clears coprocessor flag c_n.

Word ***ErrorCode*** ()
Delivers the error code of the last system-call.

ThreadId ***IntendedReceiver*** ()
Delivers the intended receiver of last received IPC (see IPC, page 54).

ThreadId ***ActualSender*** ()
Delivers the actual sender of the last propagated IPC (see IPC, page 54).

void ***Set_VirtualSender*** (*ThreadId t*)
Sets the virtual sender for the next deceiving IPC (see IPC, page 54).

Word ***PreemptedIP*** ()
Delivers the IP of the thread at the last signalled preemption.

void ***Set_PreemptCallbackIP*** (*Word ip*)
Sets the address for preemption callback.

Word ***NotifyMask*** ()
Delivers the current NotifyMask of the thread.

void ***Set_NotifyMask*** (*Word mask*)
Sets the NotifyMask.

Word ***NotifyBits*** ()
Delivers the current NotifyBits of the thread.

void ***Set_NotifyBits*** (*Word bits*)
Sets the NotifyBits field.

Code generators of IDL and other compilers are not restricted to the generic interface. They can use any processor-specific methods and optimizations to access TCRs.

2.3 EXCHANGEREGISTERS [Systemcall]

ThreadId	*dest*	$\longrightarrow$	*ThreadId*	*result*
Word	*control*		*Word*	*control*
Word	*SP*		*Word*	*SP*
Word	*IP*		*Word*	*IP*
Word	*FLAGS*		*Word*	*FLAGS*
ThreadId	*pager*		*ThreadId*	*pager*
Word	*UserDefinedHandle*		*Word*	*UserDefinedHandle*

Exchanges or reads a thread's *FLAGS*, *SP*, and *IP* hardware registers as well as *pager* and *UserDefinedHandle* TCRs. Furthermore, thread execution can be suspended or resumed. The destination thread must be an *active* thread (see page 23) residing in the invoker's address space unless the caller is a root server or the thread's pager.

Any *IP*, *SP*, or *FLAGS* modification changes the corresponding *user-level* registers of the addressed thread. In general, ongoing kernel activities are not influenced. However, a currently active IPC operation can be canceled or aborted. For details see the SR-bit specification below.

Modifications of the *pager* TCR and the *UserDefinedHandle* TCR become immediately effective, whether the destination thread executes in user mode or in kernel mode.

Input Parameters

---

dest Thread ID of the addressed thread. However, the addressed thread must reside in the current address space or the calling thread must be a root server or the thread's pager.

---

control

from $_{(18/32)}$	0 $_{(3/19)}$	$r\,d\,h\,p\,u\,f\,i\,s\,S\,R\,H$

$h\,p\,u\,f\,i\,s$ The s-flag refers to the *SP* register, i to *IP*, f to *FLAGS*, u to the *UserDefinedHandle* TCR, p to the *pager* TCR, and h to the H-flag. If a flag is set to 1, the register/state is overwritten by the corresponding input parameter. Otherwise, the corresponding input parameter is ignored and the register/state is not modified.

$S\,R$ Controls whether the addressed thread's ongoing IPC operation should be canceled/aborted through the system call or not.

- $S = 0$ An IPC operation of the addressed thread that is currently waiting to send a message or is sending a message will continue as usual. *SP*, *IP* or *FLAGS* modifications are deferred until the IPC operation terminates.
- $S = 1$ An IPC operation of the addressed thread that is currently waiting to send a message will be *canceled*.
- $R = 0$ An IPC operation of the addressed thread that is currently waiting to receive a message or is receiving a message will continue as usual. *SP*, *IP* or *FLAGS* modifications are deferred until the IPC operation terminates.
- $R = 1$ An IPC operation of the addressed thread that is currently waiting to receive a message will be *canceled*. An IPC operation that is currently receiving a message will be *aborted*.

H Halts/resumes the thread if $h = 1$. Ignored for $h = 0$.

- $H = 0$ No effect if the thread was not halted. Otherwise, thread execution is resumed.
- $H = 1$ User-level thread execution is halted. Note that ongoing IPCs and other kernel operations are not affected by H. (See SR for also aborting active IPC.)

d	If $d = 1$ the result parameters (*IP*, *SP*, *FLAGS*, UserDefinedHandle, pager, control) are delivered. If $d = 0$ the return values are undefined.
from	Specifies the thread number of the source-thread when $r = 1$.
r	If $r = 1$, user registers are copied from *from* to *dest*. The user's *IP*, *SP* are not copied. This is useful for implementing fork semantics.

SP	The current user-level stack pointer is set to *SP* if $s = 1$. Ignored for $s = 0$.

IP	The current user-level instruction pointer is set to *IP* if $i = 1$. Ignored for $i = 0$.

FLAGS	Sets the user-level processor flags of the thread if $f = 1$. Ignored for $f = 0$. The semantics of the *FLAGS* word depends on the processor type.

UserDefinedHandle	Sets the thread's *UserDefinedHandle* TCR if $u = 1$. Ignored for $u = 0$.

---

pager	Sets the thread's *pager* TCR if $p = 1$. Ignored for $p = 0$.

Output Parameters

result $\neq$ *nilthread*	*global* thread ID of the addressed thread. EXCHANGEREGISTERS succeeded.
result $=$ *nilthread*	Operation failed. The ErrorCode TCR indicates the reason for the failure.

ErrorCode **[TCR]**	Set if *result* $=$ *nilthread*. Undefined if *result* $\neq$ *nilthread*.
$= 2$	Invalid thread. The *dest* parameter specified an invalid thread ID, an inactive thread, or a thread within a different address space.

---

control

$0_{\,(29/61)}$	S	R	H

The control parameter is only valid if $d = 1$ and undefined otherwise.

H	Reports whether the addressed thread was halted ($H = 1$) or not ($H = 0$) when EXCHANGEREGISTERS was invoked. Note that this output *control* bit is independent of the input parameter *control*.
SR	Reports whether the addressed thread was within an IPC operation when EXCHANGEREGISTERS was invoked. A value of 0 reports that the addressed thread was not within a send phase ($S = 0$) or not within a receive phase ($R = 0$), respectively. Note that these output *control* bits are independent of the input parameter *control*.
$R = 1$	Operation was executed while the addressed thread was within the receive phase of an IPC operation. Iff the input control word had $R = 1$ the IPC operation was canceled or aborted.
$S = 1$	Operation was executed while the addressed thread was within the send phase of an IPC operation. Iff the input control word had $S = 1$ the IPC operation was canceled or aborted.

SP Old user-level stack pointer of the thread, if $d = 1$ and undefined for $d = 0$.

IP Old user-level instruction pointer of the thread, if $d = 1$ and undefined for $d = 0$.

FLAGS Old user-level flags of the thread, if $d = 1$ and undefined for $d = 0$. The semantics of this word is processor specific.

UserDefinedHandle
Old content of thread's *UserDefinedHandle* TCR, if $d = 1$ and undefined for $d = 0$.

---

pager Old content of thread's *pager* TCR, if $d = 1$ and undefined for $d = 0$.

Pagefaults

No pagefaults will happen.

Generic Programming Interface

System-Call Function:

#include <l4/thread.h>

ThreadId ***ExchangeRegisters*** (*ThreadId dest, Word control, sp, ip, flags, UserDefinedHandle, ThreadId pager, Word& old_control, old_sp, old_ip, old_flags, old_UserDefinedHandle, ThreadId& old_pager*)

Convenience Programming Interface

Derived Functions:

#include <l4/thread.h>

Word ***UserDefinedHandle*** (*ThreadId t*) [*UserDefinedHandleOf*]

void ***Set_UserDefinedHandle*** (*ThreadId t, Word handle*) [*Set_UserDefinedHandleOf*]
Delivers/sets the user defined handle of specified thread.

ThreadId ***Pager*** (*ThreadId t*) [*PagerOf*]

void ***Set_Pager*** (*ThreadId t, p*) [*Set_PagerOf*]
Delivers/sets the pager for specified thread.

void ***Start*** (*ThreadId t*)

void ***Start*** (*ThreadId t, Word sp, ip*) [*Start_SpIp*]

void ***Start*** (*ThreadId t, Word sp, ip, flags*) [*Start_SpIpFlags*]
Resume execution of specified thread (if halted). Abort any ongoing IPC operations. Optionally modify stack pointer, instruction pointer, and processor flags according to function parameters.

ThreadState ***Stop*** (*ThreadId t*)

ThreadState ***Stop*** (*ThreadId t, Word& sp, ip, flags*) [*Stop_SpIpFlags*]
Halt execution of specified thread and return its current thread state. Do not abort any ongoing IPC operation. Optionally return thread's stack pointer, instruction pointer, and processor flags in output parameters.

ThreadState ***AbortReceive_and_stop*** (*ThreadId t*)

ThreadState ***AbortReceive_and_stop*** (*ThreadId t, Word& sp, ip, flags*) [*AbortReceive_and_stop_SpIpFlags*]
As *stop ()*, except any ongoing IPC receive operation is immediately aborted.

ThreadState ***AbortSend_and_stop*** (*ThreadId t*)

ThreadState ***AbortSend_and_stop*** (*ThreadId t, Word& sp, ip, flags*) [*AbortSend_and_stop_SpIpFlags*]
As *stop ()*, except any ongoing IPC send operation is immediately aborted.

ThreadState ***AbortIpc_and_stop*** (*ThreadId t*)

ThreadState ***AbortIpc_and_stop*** (*ThreadId t, Word& sp, ip, flags*) [*AbortIpc_and_stop_SpIpFlags*]
As *stop ()*, except any ongoing IPC send or receive operations are immediately aborted.

void ***Copy_regs*** (*ThreadId src, ThreadId dest*)

void ***Copy_regs*** (*ThreadId src, ThreadId dest, Word sp, ip*) [*Copy_regs_SpIp*]

Support Functions:

#include <l4/thread.h>

struct **THREADSTATE** { Word raw }

Bool ***ThreadWasHalted*** (*ThreadState s*)

Bool ***ThreadWasSending*** (*ThreadState s*)

Bool ***ThreadWasReceiving*** (*ThreadState s*)

Bool ***ThreadWasIpcing*** (*ThreadState s*)
Query the thread state returned from one of the *stop ()* functions.

Word ***ErrorCode*** ()

Word ***ErrInvalidThread***

2.4 THREADCONTROL [Privileged Systemcall]

ThreadId	*dest*	⟶	*Word*	*result*
ThreadId	*SpaceSpecifier*			
ThreadId	*scheduler*			
ThreadId	*pager*			
ThreadId	*SendRedirector*			
ThreadId	*ReceiveRedirector*			
*void**	*UtcbLocation*			

A privileged thread, e.g., the root server, can delete and create threads through this function. It can also modify the global thread ID (version field only) of an existing thread.

Threads can be created as *active* or *inactive* threads. Inactive threads do not execute but can be activated by active threads that execute in the same address space.

An actively created thread starts immediately by executing a short receive operation from its pager. (An active thread must have a pager.) The actively started thread expects a start message (MsgTag and two untyped words) from its pager. Once it receives the start message, it takes the value of MR_1 as its new *IP*, the value of MR_2 as its new *SP*, and then starts execution at user level with the received *IP* and *SP*.

Interrupt threads are treated as normal threads. They are active at system startup and can *not* be deleted or migrated into a different address space (i.e., SpaceSpecifier must be equal to the interrupt thread ID). When an interrupt occurs the interrupt thread sends an IPC to its pager and waits for an empty end-of-interrupt acknowledgment message (MR_0=0). Interrupt threads never raise pagefaults. To deactivate interrupt message delivery the pager is set to the interrupt thread's own ID.

Input Parameters

---

dest Addressed thread. Only the thread number is effectively used to address the thread. If a thread with the specified thread number exists, its version bits are overwritten by the version bits of *dest id* and any ongoing IPC operations are aborted. Otherwise, the specified version bits are used for thread creations, i.e., a thread creation generates a thread with ID *dest*.

SpaceSpecifier ≠ *nilthread, dest not existing*

Creation. The space specifier specifies in which address space the thread will reside. Since address space do not have own IDs, a thread ID is used as *SpaceSpecifier*. Its meaning is: the new thread should execute in the same address space as the thread *SpaceSpecifier*.
The first thread in a new address space is created with *SpaceSpecifier = dest*. This operation implicitly creates a new empty address space. Note that the new address space is created with an empty UTCB and KIP area. The space creation *must* therefore be completed by a SPACECONTROL operation before the thread(s) can execute.

SpaceSpecifier ≠ *nilthread, dest exists*

Modification Only. The addressed thread *dest* is neither deleted nor created. Modifications can change the version bits of the thread ID, the associated scheduler, the pager, the send/receive redirector or the associated address space, i.e., migrate the thread to a new address space.

SpaceSpecifier = *nilthread, dest exists*

Deletion. The addressed thread *dest* is deleted. Deleting the last thread of an address space implicitly also deletes the address space. Any pending IPCs to the thread will be cancelled/aborted.

---

scheduler ≠ *nilthread*

Defines the scheduler thread that is permitted to schedule the addressed thread. Note that the scheduler thread must exist when the addressed thread starts executing.

scheduler = *nilthread*
The current scheduler association is not modified . This variant is illegal for a creating THREADCONTROL operation.

---

pager ≠ *nilthread* The pager of *dest* is set to the specified thread. If *dest* was inactive before, it is *activated.*

pager = *nilthread* The current pager association is not modified.
If used with a creating THREADCONTROL operation, *dest* is created as an *inactive* thread.

SendRedirector = *nilthread*
The current send-redirector setting for the specified thread is not modified.

SendRedirector = *anythread*
The specified thread is allowed to send an IPC to any thread in the system.

SendRedirector ≠ *anythread*, ≠ *nilthread*
The specified thread is only allowed to send an IPC to a local thread or to a thread in the same address space as the specified send-redirector. All other send operations will be deflected to the redirector, the *redirected bit* (see page 54) in the received message will be set, and the *IntendedReceiver* TCR will indicate the intended receiver of the message.

ReceiveRedirector = *nilthread*
The current receive-redirector setting for the specified thread is not modified.

ReceiveRedirector = *anythread*
The specified thread is allowed to receive an IPC from any thread in the system.

ReceiveRedirector ≠ *anythread*, ≠ *nilthread*
The specified thread is only allowed to receive an IPC from a local thread or a thread in the same address space as the specified receive-redirector. All other send operations to the thread will be deflected to the redirector, the *redirected bit* (see page 54) in the received message will be set, and the *IntendedReceiver* TCR will indicate the intended receiver of the message.

UtcbLocation ≠ *-1* The start address of the UTCB of the thread is set to UtcbLocation. Upon thread activation, the UTCB must fit entirely into the UTCB area of the configured address space, and must be properly aligned according to the UtcbInfo field of the kernel interface page.
It is the application's responsibility to ensure that UTCBs of multiple threads do not overlap. Changing the UtcbLocation of an already active thread is an illegal operation. Note that since a newly created space has an empty UTCB area, it is not possible to activate a thread in an address space which has not been properly configured with SPACECONTROL.
Note that if the s field of the UtcbInfo field is 0, then the location of the UTCB cannot be specified and is controlled by the kernel. In this case, a value of 0 for UtcbLocation must be provided to THREADCONTROL in order to activate a thread (see page 41).

UtcbLocation = *-1* The UTCB location is not modified.

UtcbInfo ***[KernelInterfacePage Field]***
Permits to calculate the appropriate page size of the UTCB area fpage and specifies the size and alignment of UTCBs. Note that the size restricts the total number of threads that can reside in an address space.

$\sim$ (10/42)	s (6)	a (6)	m (10)

s The minimal *area size* for an address space's UTCB area is 2^s. The size of the UTCB area limits the total number of threads k to $2^a mk \leq 2^s$.

m	UTCB size multiplier.
a	The UTCB location must be aligned to 2^a. The total size required for one UTCB is $2^a m$.

Output Parameters

result	The result is 1 if the operation succeeded, otherwise the result is 0 and the ErrorCode TCR indicates the failure reason.

ErrorCode **[TCR]**	Set if *result* $= 0$. Undefined if *result* $\neq 0$.
$= 1$	No privilege. Current thread does not have have privilege to perform the operation.
$= 2$	Unavailable thread. The *dest* parameter specified a kernel thread or an unavailable interrupt thread.
$= 3$	Invalid space. The *SpaceSpecifier* parameter specified an invalid thread ID, or activation of a thread in a not yet initialized space.
$= 4$	Invalid scheduler. The *scheduler* parameter specified an invalid thread ID, or was set to *nilthread* for a creating THREADCONTROL operation.
$= 6$	Invalid UTCB location. *UtcbLocation* lies outside of UTCB area, or attempt to change the *UtcbLocation* for an already active thread.
$= 8$	Out of memory. Kernel was not able to allocate the resources required to perform the operation.
$= 9$	An invalid redirector thread ID was specified, or a redirection-loop was detected.

Pagefaults

No pagefaults will happen.

Generic Programming Interface

System-Call Function:

#include <l4/thread.h>

Word ***ThreadControl*** (*ThreadId dest, SpaceSpecifier, Scheduler, Pager, SendRedirector, ReceiveRedirector, void* UtcbLocation*)

Convenience Programming Interface

Derived Functions:

#include <l4/thread.h>

Word ***AssociateInterrupt*** (*ThreadId InterruptThread, InterruptHandler*)
{ ThreadControl (InterruptThread, InterruptThread, nilthread, InterruptHandler, nilthread, nilthread, -1) }

Associate a handler thread with the specified interrupt source.

Word ***DeassociateInterrupt*** (*ThreadId InterruptThread*)
{ ThreadControl (InterruptThread, InterruptThread, nilthread, InterruptThread, nilthread, nilthread, -1) }

Remove association between the specified interrupt source and any potential handler thread.

void ***Set_SendRedirector*** (*ThreadId Thread, ThreadId Redirector*)
{ ThreadControl (Thread, Thread, nilthread, nilthread, Redirector, nilthread, -1) }

Set the send-redirector of the specified thread.

void ***Set_ReceiveRedirector*** (*ThreadId Thread, ThreadId Redirector*)
{ ThreadControl (Thread, Thread, nilthread, nilthread, nilthread, Redirector, -1) }

Set the receive-redirector of the specified thread.

Support Functions:

Word ***ErrorCode*** ()

Word ***ErrNoPrivilege***

Word ***ErrInvalidThread***

Word ***ErrInvalidSpace***

Word ***ErrInvalidScheduler***

Word ***ErrUtcbArea***

Word ***ErrNoMem***

Word ***ErrInvalidRedirector***

Chapter 3

Scheduling

3.1 THREADSWITCH [Systemcall]

ThreadId dest $\longrightarrow$ *void*

The invoking thread releases the processor (non-preemptively) so that another ready thread can be processed.

Input Parameter

---

dest $=$ *nilthread*	Processing switches to an undefined ready thread which is selected by the scheduler. (It might be the invoking thread.) Since this is "ordinary" scheduling, the thread gets a new timeslice.
dest $\neq$ *nilthread*	If *dest* is ready, processing switches to this thread. In this "extraordinary" scheduling, the invoking thread donates its remaining timeslice to the destination thread. (This one gets the donation in addition to its ordinarily scheduled timeslices, if any.) If the destination thread is not ready or resides on a different processor, the system call operates as described for *dest* = *nilthread*.

Pagefaults

No pagefaults will happen.

Generic Programming Interface

System-Call Function:

```
#include <l4/schedule.h>

void ThreadSwitch (ThreadId dest)
```

Convenience Programming Interface

Derived Functions:

```
#include <l4/schedule.h>

void Yield ()
        { ThreadSwitch (nilthread) }
```

Switch processing to a thread selected by the scheduler.

3.2 SCHEDULE [Systemcall]

ThreadId	*dest*	⟶	*Word*	*result*
Word	*ts len*		*Word*	*rem ts*
Word	*total quantum*		*Word*	*rem quantum*
Word	*processor control*			
Word	*prio*			

The system call can be used by schedulers to define the *priority*, *timeslice length*, and other scheduling parameters of threads. Furthermore, it delivers thread states.

The system call is only effective if the calling thread is defined as the destination thread's scheduler (see *thread control*, page 23).

Input Parameters

---

dest Destination thread ID. The destination thread must be existent (but can be inactive) and the current thread must be defined as the destination thread's scheduler (see *thread control*). Otherwise, the destination thread is not affected.

All further input parameters have no effect if the supplied value is -1, ensuring that the corresponding internal thread variable is *not* modified. The following description always refers to values $\neq -1$.

---

prio

New priority for destination thread. Must be less than or equal to current thread's priority.

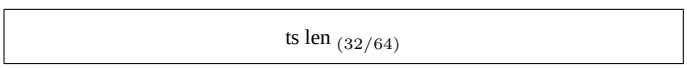
---

processor control

processor number Specifies the processor number to which the thread should be migrated. The processor number must be valid, i.e., smaller than the total number of processors (see kernel interface page at page 3). Otherwise, the parameter is ignored. The first processor number is denoted as 0.

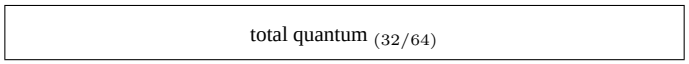
Time controls Time values are specified as values measured in microseconds. The size of the values matches the word-size of the machine architecture. Thus on a 32-bit system, a maximal time of 71 minutes is allowed, and 64-bit systems have practically no limit.

ts len



New timeslice length for the destination thread. A timeslice length of ∞, can be specified, encoded as 0. In that case, the thread never experiences a preemption due to exhausted time slice. The specified value is always rounded up to the nearest possible timeslice length. In particular, a time period of 1 μs results in the shortest possible timeslice. Specifying -1 means that the timeslice length is not modified.

total quantum

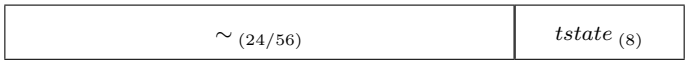


Defines the total quantum for the thread. Exhaustion of the total quantum results in an RPC to the thread's scheduler (i.e., the current thread). (Re)writing the total quantum re-initializes the quantum, independent of the already consumed total quantum. A total quantum of ∞ can be specified, encoded as 0. Specifying -1 means that the total quantum is not modified.
Writing the total quantum reinitializes the current timeslice. After the quantum is exhausted, the thread is preempted while the quantum is reloaded with *ts len* for the next timeslice.

Output Parameters

---

result



$tstate =$ Thread state:

0 *Error.* The operation failed completely. The ErrorCode TCR indicates the reason for the failure.

1 *Dead.* The thread is unable to execute or does not exist.

2 *Inactive.* The thread is inactive/stopped.

3 *Running.* The thread is ready to execute at user-level.

4 *Pending* send. A user-invoked IPC send operation currently waits for the destination (recipient) to become ready to receive.

5 *Sending.* A user-invoked IPC send operation currently transfers an outgoing message.

6 *Waiting* to receive. A user-invoked IPC receive operation currently waits for an incoming message.

7 *Receiving.* A user-invoked IPC receive operation currently receives an incoming message.

8 *WaitingNotify.* The thread is currently waiting for an asynchronous notification.

9 *XcpuWaiting.* The thread is waiting for inter-cpu communication.

ErrorCode **[TCR]** Set if lower 8 bits of *result* $= 0$. Undefined if lower 8 bits of *result* $\neq 0$.

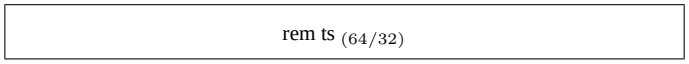
$= 1$ No privilege. Current thread is not the scheduler of the destination thread.

$= 2$ The *dest* parameter specified an invalid thread ID.

$= 5$ Invalid parameter. The specified time-slice length, total quantum, priority, or processor number was invalid.

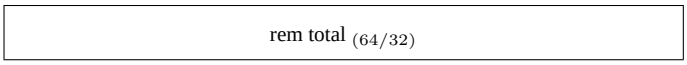
Time controls Time values are specified in microseconds.

rem ts



Remainder of the current timeslice.

rem total



Remaining total quantum of the thread.

Pagefaults

No pagefaults will happen.

Generic Programming Interface

System-Call Function:

#include <l4/schedule.h>

Word ***Schedule*** (*ThreadId dest, ProcessorControl, prio, PreemptionControl*)

Convenience Programming Interface

Derived Functions:

#include <l4/schedule.h>

Word ***Set_Priority*** (*ThreadId dest, Word prio*)
{ Schedule (dest, -1, -1, prio, -1) }

Word ***Set_ProcessorNo*** (*ThreadId dest, Word ProcessorNo*)
{ Schedule (dest, -1, ProcessorNo, -1, -1) }

Word ***Timeslice*** (*ThreadId dest, Word & ts, Word & tq*)
Delivers the remaining timeslice and total quantum of the given thread.

Word ***Set_Timeslice*** (*ThreadId dest, Word ts, Word tq*)
Sets the timeslice and total quantum of the given thread.

Support Functions:

Word ***ErrorCode*** ()

Word ***ErrNoPrivilege***

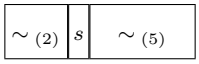
Word ***ErrInvalidThread***

Word ***ErrInvalidParam***

3.3 Preempt Flags [TCR]

The *preemption flags* TCR controls asynchronous preemptions (timeslice exhausted or activation of a higher-priority thread including device interrupts).

Preempt Flags



$s = 0$ Asynchronous preemptions are not signaled.

$s = 1$ Asynchronous preemptions are signaled as a callback by changing the thread's restart instruction pointer to the value specified in the *PreemptCallbackIP* TCR. The thread's instruction pointer at the time of interruption is saved in the *PreemptedIP* TCR.

Generic Programming Interface

#include <l4/schedule.h>

Bool ***EnablePreemptionCallback*** ()

Bool ***DisablePreemptionCallback*** ()
Sets/resets the s-flag and delivers the old s-flag value (true = set).

Word ***PreemptedIP*** ()
Returns the *PreemptedIP* TCR.

void ***Set_PreemptCallbackIP*** (*Word ip*)
Sets the *PreemptCallbackIP* TCR.

Chapter 4

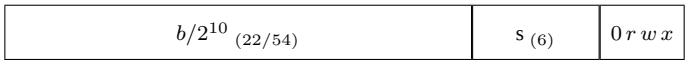
Address Spaces and Mapping

4.1 Fpage [Data Type]

Fpages (Flexpages) are regions of the virtual address space. An fpage consists of all pages mapped actually in this region excluding kernel mapped objects, i.e., kernel interface page and UTCBs. Fpages have a size of at least 1 K. For specific processors, the minimal fpage size may be larger; e.g., a Pentium processor offers a minimal page size of 4 K while the Alpha processor offers smallest pages of 8 K. Fpages smaller than the minimal page size are treated as nilpages. The kernel interface page (see page 3) specifies which page sizes are supported by the hardware/kernel. An fpage of size 2^s has a 2^s-aligned base address b, where $s \geq 10$ for all architectures.

Mapped fpages are considered inseparable objects. That is, if an fpage is mapped, the mapper can not later partially unmap the mapped page; the whole fpage must be unmapped in a single operation. The mappee can, however, separate the fpage and map fpages (objects) of smaller size. Partially unmapping an fpage might or might not work on some systems. The kernel will give no indication as to whether such an operation succeeded or not.

fpage $(b, 2^s)$



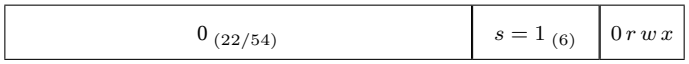
rwx Access rights. See below.

s Encodes the size of the fpage as s^2, where $s \geq 10$.

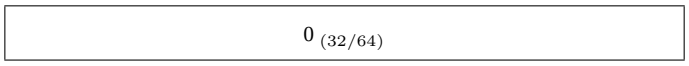
b Encodes the base address of the fpage as $b * 2^{10}$. b must produce a 2^s aligned address.

Special fpage encodings describe the *complete* user address space and the *nilpage*, an fpage which has no base address and a size of 0:

complete



nilpage



Access Rights

rwx The rwx bits define the accessibility of the fpage:

r	readable
w	writable
x	executable

A bit set to one permits the corresponding access to the newly-mapped page *provided that the mapper itself* possesses that access right.
Note that processor architectures may impose restrictions on the access-right combinations. However, *read-only* (including execute), $rwx = 101$, and *read/write/execute*, $rwx = 111$, should be valid for any processor architecture. The kernel interface page (see page 3) specifies which access rights are supported in the processor architecture.

Generic Programming Interface

#include <l4/space.h>

struct **FPAGE** { Word raw }

Word ***Readable***

Word ***Writable***

Word ***eXecutable***

Word ***FullyAccessible***

Word ***ReadeXecOnly***

Word ***NoAccess***

Fpage ***Nilpage***

Fpage ***CompleteAddressSpace***

Bool ***IsNilFpage*** (*Fpage f*)
{ f == *Nilpage* }

Fpage ***Fpage*** (*Word BaseAddress, Word FpageSize* $\geq$ *1K*)

Fpage ***FpageLog2*** (*Word BaseAddress, int Log2FpageSize* $<$ *64*)
Delivers an fpage with the specified location and size.

Word ***Address*** (*Fpage f*)

Word ***Size*** (*Fpage f*)

Word ***SizeLog2*** (*Fpage f*)
Delivers address/size of specified fpage.

Word ***Rights*** (*Fpage f*)

void ***Set_Rights*** (*Fpage& f, Word AccessRights*)
Delivers/sets the access rights for the specified fpage.

Fpage $+$ (*Fpage f, Word AccessRights*) [*FpageAddRights*]

Fpage $+=$ (*Fpage f, Word AccessRights*) [*FpageAddRightsTo*]

Fpage $-$ (*Fpage f, Word AccessRights*) [*FpageRemoveRights*]

Fpage $-=$ (*Fpage f, Word AccessRights*) [*FpageRemoveRightsFrom*]
Adds/removes specified access rights from fpage. Delivers new fpage value.

4.2 PhysDesc [Data Type]

PhysDescs (physical descriptors) describe the physical memory associated with an *fpage* when combined in a *MapItem*. A *physdesc* contains a physical memory address and attributes to be associated with the fpage mapping that address. A *physdesc* is only valid when the *fpage* describes a mapping that covers a contiguous mapped area.

physdesc (p, a)

a Encodes the page attributes to be applied to the associated fpage.

p Encodes the physical base address of the fpage as $p * 2^{10}$. p must produce an address with the same alignment as the associated *fpage*.

Generic attributes

L4 defines eight generic page attributes which all architectures must support. If a particular CPU does not support the specified attribute, a compatible attribute will be chosen such that functional correctness is maintained. Other attributes may be architecture and CPU specific and are described in the architecture specific sections.

Attribute	Description	
0x0	*default_memory*	CPU specific optimal for code and data
0x1	*cached*	CPU specific writethough/writeback
0x2	*uncached*	Caching is disabled for the page
0x3	*writeback*	Cached-writeback
0x4	*writethrough*	Cached-writethrough
0x5	*coherent*	Cached-coherent
0x6	*io_memory*	CPU specific attribute for memory-mapped I/O
0x7	*io_combine*	Uncached with write-combine (eg. frame buffer memory)

Generic Programming Interface

Memory Attributes

Word ***DefaultMemory***

Word ***CachedMemory***

Word ***UncachedMemory***

Word ***WriteBackMemory***

Word ***WriteThroughMemory***

Word ***CoherentMemory***

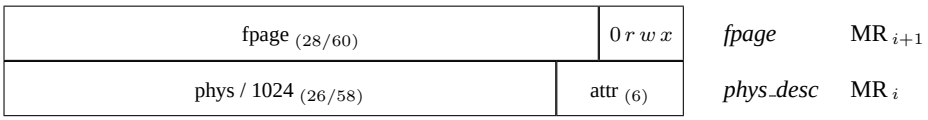
Word ***IOMemory***

Word ***IOCombinedMemory***

PhysDesc ***PhysDesc*** (*Word PhysAddress, Word Attribute*)

4.3 MapItem [Data Type]

An *fpage* (see page 34) or IO fpage that should be mapped is sent to MAPCONTROL as part of the arguments. The fpage is specified by a two-word descriptor, a *MapItem*:



access rights rwx The effective access rights for the newly mapped page.

physical address $phys$

The *phys* field specifies the physical memory address of the page to be mapped.

page attributes $attr$

The memory attributes to be applies to the page. This allows use of cachability and memory ordering semantics supported by the processor.

Pages already mapped in the mappee's address space that would conflict with new mappings are implicitly unmapped before new pages are mapped. For performance reasons, extension of access rights is possible without prior unmapping, iff the very same mapping already exists. This is the case, when

- the mapper maps from the same physical address as the existing mapping; *and*
- the mapper maps to the same virtual destination address as the existing mapping; *and*
- the object (physical address) is the same as the existing mapping.

Access rights can be revoked by mapping. The access rights of the resulting mapping are overwritten with the access rights provided in the *MapItem*.

4.4 MAPCONTROL [Privileged Systemcall]

ThreadId *SpaceSpecifier* *Word* *control* $\longrightarrow$ *Word* *result*

MapControl is used to control mapping of address spaces by a privileged server. This system call is used to insert, modify and query mappings in a target address space.

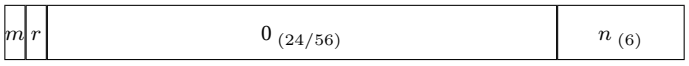
Mappings are specified in MapItems which are located in $\text{MR}_{0\ldots}$. MapControl operations operate on MapItems sequentially. If both query and modify operations are specified, the kernel will perform a query on the first MapItem, then a modify operation on that MapItem before moving to the next MapItem.

Input Parameters

SpaceSpecifier Since address spaces do not have ids, a thread ID is used as *SpaceSpecifier*. It specifies the address space in which to query and/or map fpages into. The *SpaceSpecifier* thread must exist although it may be inactive or not yet started.

---

control



n Specifies the highest numbered MapItem that is located in the message registers. MapItems are in message registers $\text{MR}_{0\ldots 2n+1}$.

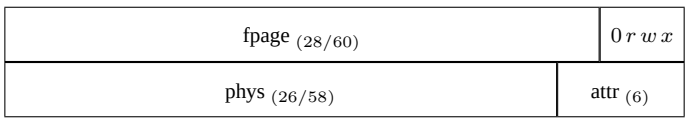
r Specifies that a read operation is to be performed. A read operation operates before a modify operation and returns the state of the mapping before modification. Read operations are performed on the fpage contained in MapItems which are input in the message registers. Results are returned in QueryItems in the message registers.

m Specifies that a modify operation is to be performed. A modify operation operates after a read operation and operates on MapItems passed in message registers.

MapItemList *MR* $_{0\ldots 2n+1}$

MapItems to be processed.

MapItem *MR* $_{2n\ldots 2n+1}$



fpage MR $_{2n+1}$

phys_desc MR $_{2n}$

MapItem to be acted upon if a map or read operation is specified. This object describes a mapping to be inserted, modified and/or queried according to *control*.

fpage Fpage in the address space specified by the *SpaceSpecifier*. A nilpage specifies a no-op.

$0rwx$ Any access bit set to 1 grants the corresponding access right. A cleared bit specifies that the corresponding access right should be revoked. Typical examples:

=0111 Give full access to the fpage. If the fpage existed, its rights are extended.

=0100 Partial map, only allow read-access to the fpage. If the fpage existed, its rights are modified. As a result, the fpage is set to read-only.

=0000	Unmap. All rights are revoked from the fpage and the fpage is unmapped (removed) from the specified address space and will no longer be accessible from that address space.
attr	This specifies the memory attributes to be associated with the fpage. L4 defines eight standard attributes (see page 36), all architectures must implement these attributes. If a particular CPU does not support the specified attribute, the page will be mapped such that functional correctness is maintained. An example: Specifying *io_memory* can be mapped as *uncached* if the CPU does not have specific page attributes for IO device mapping.
phys	Specifies the physical address to use to back the fpage. The physical address is encoded as $base{*}2^{10}$. The physical address must have the same alignment as the fpage. On 32-bit machines, up to 36-bits of physical address can be specified.

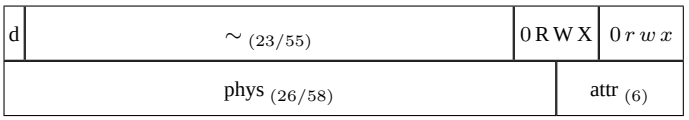
Output Parameters

result The result is 1 if the operation succeeded, otherwise the result is 0 and the ErrorCode TCR indicates the failure reason.

QueryItemList *MR* $_{0\ldots2n+1}$

QueryItems returned.

QueryItem *MR* $_{2n\ldots2n+1}$



If a read operation is specified in the control word, the MapItems are replaced with QueryItems when the system call returns.

rwx	Page access permissions. The bitwise-OR of the page permissions of all pages covered by the fpage in the MapItem. These are the permissions before a modify operation, if specified, took place.
RWX	Page reference bits. The bitwise-OR of the reference bits of all pages covered by the fpage in the MapItem. Reference bits are only valid for processors that support hardware maintained reference bits.
d	Distinct flag. If the corresponding MapItem covers an area of address space that contains exactly one distinct fpage, of the same size, then the distinct (d) flag is set. The $phys$ and $attr$ fields are only valid when this flag is set.
attr	This contains the page attributes of the addressed fpage if the d-flag is set. The value is undefined if d is not set.
phys	This contains the physical address of the addressed fpage if the d-flag is set. The value is undefined if d is not set.

ErrorCode **[TCR]** Set if *result* $= 0$. Undefined if *result* $\neq 0$.

$= 1$	No privilege. Current thread does not have privilege to perform operation.
$= 3$	Invalid space. The *SpaceSpecifier* parameter specified an invalid thread ID.
$= 5$	Invalid parameter. The n field of the control word specifies more MapItems than would fit in available message registers.

Pagefaults

No pagefaults will happen.

Generic Programming Interface

System-Call Function:

#include <l4/space.h>

Word ***MapControl*** (*ThreadId SpaceSpecifier, Word control*)

Convenience Programming Interface

Derived Functions:

Fpage ***Map*** (*ThreadId s, Fpage f, PhysDesc p*) [*MapFpage*]

Fpage ***Map*** (*ThreadId s, Word n, Fpage& [n] fpages PhysDesc& [n] p*) [*MapFpages*]

Maps the specified fpage(s) into the specified address space.
Returns *QueryItems* in *fpages* and *p*.

Fpage ***Unmap*** (*ThreadId s, Fpage f*) [*UnmapFpage*]

Fpage ***Unmap*** (*ThreadId s, Word n, Fpage& [n] fpages*) [*UnmapFpages*]

Unmaps the specified fpage(s) from the specified address space.

4.5 SPACECONTROL [Privileged Systemcall]

ThreadId	*SpaceSpecifier*	$\longrightarrow$	*Word*	*result*
Word	*control*		*Word*	*control*
Fpage	*KernelInterfacePageArea*			
Fpage	*UtcbArea*			

A privileged thread, e.g., the root server, can configure address spaces through this function.

Input Parameters

SpaceSpecifier Since address spaces do not have ids, a thread ID is used as *SpaceSpecifier*. It specifies the address space in which the thread resides. The *SpaceSpecifier* thread must exist although it may be inactive or not yet started. In particular, the thread may reside in an empty address space that is not yet completely created.

KernelInterfacePageArea

Specifies the fpage where the kernel should map the kernel interface page. The supplied fpage must have a size specified in the *KipAreaInfo* field of the kernel interface page, must fit entirely into the user-accessible part of the address space and must not overlap with the UTCB area (see below). Address 0 of the kernel interface page is mapped to the fpage's base address.
The value is ignored if there is at least one active thread in the address space.
Note that when the s field of the *KipAreaInfo* is 0, the KIP area is not part of the user address space and cannot be controlled. In this case, a value of 0 must be passed in *KernelInterfacePageArea*.

KipAreaInfo ***[KernelInterfacePage Field]***

Permits calculation of the appropriate page size of the KernelInterface area fpage.

$\sim$ (26/60)	s (6)

s The size of the kernel interface page area for an address space is 2^s. A size of 0 indicates that the KIP area is not part of the user address space and cannot be controlled.

UtcbArea Specifies the fpage where the kernel should map the UTCBs of all threads executing in the address space. The fpage must fit entirely into the user-accessible part of an address space and must not overlap with the KIP area. The fpage size has to be at least the smallest supported hardware-page size. In fact, the size of the UTCB area restricts the maximum number of threads that can be created in the address space. See the kernel interface page for the space and alignment that is required for UTCBs.
The value is ignored if there is at least one active thread in the address space.
Note that when the s field of the *UtcbInfo* is 0, the UTCB area is outside the user's accessible virtual-address space as defined in the KIP. The UTCB area address is controlled by the kernel and the standard architecture defined method of finding the UTCB address applies. In this case, a value of 0 must be passed in *UtcbArea*.

UtcbInfo ***[KernelInterfacePage Field]***

Permits to calculate the appropriate page size of the UTCB area fpage and specifies the size and alignment of UTCBs. Note that the size restricts the total number of threads that can reside in an address space.

$\sim$ (10/42)	s (6)	a (6)	m (10)

s — The minimal *area size* for an address space's UTCB area is 2^s. The size of the UTCB area limits the total number of threads k to $2^a mk \leq 2^s$. A size of 0 indicates that the UTCB is not part of the user address space and cannot be controlled (see page 41).

m — UTCB size multiplier.

a — The UTCB location must be aligned to 2^a. The total size required for one UTCB is $2^a m$.

---

control — The control field is architecture specific (see architecture specific *Space Control* section). It is undefined for some architectures, but should for reasons of upward compatibility be set to zero.

Output Parameters

---

result — The result is 1 if the operation succeeded, otherwise the result is 0 and the ErrorCode TCR indicates the failure reason.

ErrorCode **[TCR]** Set if *result* $= 0$. Undefined if *result* $\neq 0$.

$= 1$ — No privilege. Current thread does not have privilege to perform operation.

$= 3$ — Invalid space. The *SpaceSpecifier* parameter specified an invalid thread ID.

$= 6$ — Invalid UTCB area. Specified UTCB area too small (see UTCB info on page 4) or not within user accessible virtual memory region (see Memory Descriptors on page 6).

$= 7$ — Invalid KIP area. Specified KIP area too small (see KIP area info on page 4) or not within user accessible virtual memory region (see Memory Descriptors on page 6) or KIP area overlaps with UTCB area.

---

control — Delivers the space control value that was effective for the thread when the operation was invoked. The value is architecture specific.

Pagefaults

No pagefaults will happen.

Generic Programming Interface

System-Call Function:

```
#include <l4/space.h>
```

Word ***SpaceControl*** (*ThreadId SpaceSpecifier, Word control, Fpage KernelInterfacePageArea, UtcbArea, Word& old_Control*)

Convenience Programming Interface

Support Functions:

Word ***ErrorCode*** ()

Word ***ErrNoPrivilege***

Word ***ErrInvalidSpace***

Word ***ErrUtcbArea***

Word ***ErrKipArea***

Chapter 5

IPC

5.1 Messages And Message Registers (MRs) [Virtual Registers]

Messages can be sent and received through the IPC system call (see page 51). Essentially, the sender writes a message into the sender's message registers (MRs) and the receiver reads them from the its MRs. A kernel implementation will always support at least 8 message registers and no more than 64. The actual number of message registers supported is a kernel configuration option and is indicated in the *VirtualRegInfo* field of the kernel interface page. A message can use some or all MRs to transfer untyped words; it can include fpages which are also specified using MRs.

MRs are *virtual registers* (see page 11), but they are more transient than TCRs. *MRs are read-once registers:* once an MR has been read, its value is undefined until the MR is written again. The send phase of an IPC implicitly reads all MRs; the receive phase writes the received message into MRs.

The read-once property permits to implement MRs not only by special registers or memory locations, but also by general registers. Writing to such an MR has to block the corresponding general register for code-generator use; reading the MR can release it. Typically, code generated by an IDL compiler will load MRs just before an IPC system call and store them to user variables just afterwards.

Messages

A message consists of up to 2 sections: the mandatory *message tag*, followed by an optional *untyped-words* section. The message tag is always held in MR_0. It contains message control information and the *message label* which can be freely set by the user. The kernel associates no semantics with it. Often, the message label is used to encode a request key or to define the method that should be invoked by the message.

MsgTag **[MR_0]**

label $_{(16/48)}$	flags $_{(4)}$	$\sim$ $_{(6)}$	u $_{(6)}$

u — Number of untyped words following word 0. $\text{MR}_{1\ldots u}$ hold the untyped words. $u = 0$ denotes a message without untyped words. If u is greater than the number of MRs supported by the kernel (n), only n MRs will be copied.

flags — Message flags, see IPC systemcall, page 51.

label — Freely available, often used to specify the request type or invoked method.

untyped words **[$\text{MR}_{1\ldots u}$]**

The optional untyped-words section holds arbitrary data that is untyped from the kernel's point of view. The data is simply copied to the receiver. The kernel associates no semantics with it.

typed items Typed items are not supported in this version of the L4 API.

Example Messages

struct (*label, *Word [2]* w*)

Word w_2 $_{(32/64)}$	MR_2
Word w_1 $_{(32/64)}$	MR_1
label $_{(16/48)}$ | flags | $\sim$ | $u = 2$	MR_0

Generic Programming Interface

The listed generic functions permit user code to access message registers independently of the processor-specific MR model. All functions are user-level functions; the microkernel is not involved.

MsgTag

#include <l4/ipc.h>

struct **MSGTAG** { Word raw }

MsgTag ***Niltag***
A message tag with no untyped, no label, and no flags.

Bool == (*MsgTag l, r*) [*IsMsgTagEqual*]
Bool != (*MsgTag l, r*) [*IsMsgTagNotEqual*]
Compares all field values of two message tags.

Word ***Label*** (*Msg Tag t*)
Word ***UntypedWords*** (*Msg Tag t*)
Delivers the message label, and number of untyped words, respectively.

MsgTag + (*MsgTag t, Word label*) [*MsgTagAddLabel*]
MsgTag += (*MsgTag t, Word label*) [*MsgTagAddLabelTo*]
Adds a label to a message tag. Old label information is overwritten by the new label.

MsgTag ***MsgTag*** ()
void ***Set_MsgTag*** (*MsgTag t*)
Delivers/sets MR_0.

Convenience Programming Interface

IDL-compiler generated Operations

IDL code generators are not restricted to the generic interface for accessing MRs. Instead, they can use processor-specific methods and thus generate heavily optimized code for MR access.

> *However, such processor-specific MR operations are not generally defined and should be used exclusively by processor-specific IDL code generators. All other programs must use the operations defined in this generic interface.*

Msg

#include <l4/ipc.h>

struct **MSG** { Word raw[64] }

void ***Put*** (*Msg& msg, Word l, int u, Word& [u] ut*) [*MsgPut*]
Loads the specified parameters into the memory object *msg*. The parameter u indicates number of untyped words. It is assumed that the *msg* object is large enough to contain all items.

void ***Get*** (*Msg& msg, Word& ut*) [*MsgGet*]
Stores the *msg* object into the specified parameters.

MsgTag ***MsgTag*** (*Msg& msg*) [*MsgMsgTag*]
void ***Set_MsgTag*** (*Msg& msg, MsgTag t*) [*Set_MsgMsgTag*]
Delivers/sets the message tag of the *msg* object.

Word ***Label*** (*Msg& msg*) [*MsgLabel*]

void ***Set_Label*** (*Msg& msg, Word label*) [*Set_MsgLabel*]
Delivers/sets the label of the *msg* object.

void ***Load*** (*Msg& msg*) [*MsgLoad*]
Loads message registers $\mathrm{MR}_{0\ldots}$ from the *msg* object.

void ***Store*** (*MsgTag t, Msg& msg*) [*MsgStore*]
Stores the message tag t and the current message beginning with MR_1 to the memory object *msg*. The number of message registers to be stored is derived from t.

void ***Clear*** (*Msg& msg*) [*MsgClear*]
Empties the *msg* object (i.e., clears the message tag).

void ***Append*** (*Msg& msg, Word w*) [*MsgAppendWord*]
Appends an untyped item to the *msg* object. It is assumed that there is enough memory in the *msg* object to contain the new item.

void ***Put*** (*Msg& msg, Word u, Word w*) [*MsgPutWord*]
Puts an untyped word at untyped word position u (first untyped word has position 0) in the *msg* object. It is assumed that the object contains at least $u + 1$ untyped words.

Word ***Get*** (*Msg& msg, Word u*) [*MsgWord*]

void ***Get*** (*Msg& msg, Word u, Word& w*) [*MsgGetWord*]
Delivers the untyped words at position u. It is assumed that the object contains at least $u + 1$ untyped words.

Low-Level MR Access

#include <l4/ipc.h>

void ***StoreMR*** (*int i, Word& w*)

void ***LoadMR*** (*int i, Word w*)
Delivers/sets MR_i.

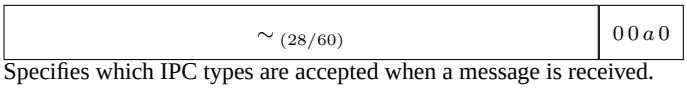
void ***StoreMRs*** (*int i, k, Word& [k] w*)

void ***LoadMRs*** (*int i, k, Word& [k] w*)
Stores/loads $\mathrm{MR}_{i\ldots i+k-1}$ to/from memory.

5.2 IPC Control Registers (TCRs) [Virtual Registers]

IPC control registers are TCRs which are used to control certain IPC operations.

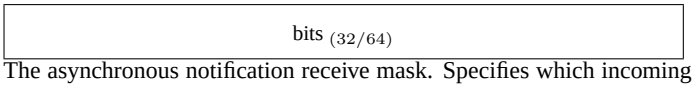
Acceptor **[TCR]**



Specifies which IPC types are accepted when a message is received.

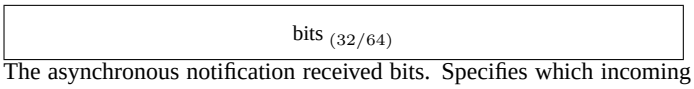
a Asynchronous notifications are accepted (delivered to the thread's $NotifyBits$ TCR) iff $a = 1$. Note that the recipient will only receive the notification if/when it does the appropriate IPC wait operation, see page 51 for details.

NotifyMask **[TCR]**



The asynchronous notification receive mask. Specifies which incoming asynchronous notification bits are waited on (accepted) when a asynchronous notification message is received.

NotifyBits **[TCR]**



The asynchronous notification received bits. Specifies which incoming asynchronous notification bits have been delivered.

Generic Programming Interface

The listed generic functions permit user code to access the IPC control registers. All functions are user-level functions; the microkernel is not involved.

Acceptor

#include <l4/ipc.h>

struct **ACCEPTOR** { Word raw }

Acceptor ***UntypedWordsAcceptor***

Acceptor ***NotifyMsgAcceptor***
Delivers an acceptor which allows untyped words or asynchronous notifications.

Acceptor + (*Acceptor l, r*) [*AddAcceptor*]

Acceptor += (*Acceptor l, r*) [*AddAcceptorTo*]
Adds asynchronous notification items to an acceptor.

Acceptor − (*Acceptor l, r*) [*RemoveAcceptor*]

Acceptor −= (*Acceptor l, r*) [*RemoveAcceptorFrom*]
Removes asynchronous notification items from an acceptor.

void ***Accept*** (*Acceptor a*)
Sets acceptor.

Acceptor ***Accepted*** ()
Returns the current acceptor.

void ***Set_NotifyMask*** (*Word mask*)
Sets the asynchronous notification receive mask.

Word ***Get_NotifyMask*** ()
Returns the asynchronous notification receive mask.

void ***Set_NotifyBits*** (*Word bits*)
Sets the asynchronous notification received bits.

Word ***Get_NotifyBits*** ()
Returns the asynchronous notification received bits.

5.3 IPC [Systemcall]

ThreadId to ThreadId *FromSpecifier* $\longrightarrow$ *ThreadId from*

IPC is the fundamental operation for inter-process communication and synchronization. It can be used for intra- and inter-address-space communication. All communication is unbuffered and, with the exception of *asynchronous notification*, synchronous in nature: a message is transferred from the sender to the recipient if and only if the recipient has invoked a corresponding IPC operation. The sender blocks until this happens or the IPC fails immediately depending on parameters specified by the sender.

IPC can be used to signal asynchronous notifications between threads. Notifications can be delivered only if a recipient is accepting notifications (a is set in *Acceptor*). Notifications are similar to IPC, however it is always possible to notify a thread regardless of whether it has called the corresponding IPC operation to wait for notifications.

IPC can be used to send synchronous messages as well as to send asynchronous notifications from the sender to the recipient. For the description of messages see page 46. A single IPC call combines an optional send phase and an optional receive phase. Which phases are included is determined by the parameters *to* and *FromSpecifier*. Transitions between send phase and receive phase are atomic.

IPC operations are also controlled by MRs, and some TCRs.

Variants

To enable implementation-specific optimizations, there exist two variants of the IPC system call. Functionally, both variants are identical. Transparently to the user, a kernel implementation can unify both variants or implement differently optimized functions.

IPC Default IPC function. Must always be used except if all criteria for using LIPC are fulfilled.

LIPC IPC function that may be optimized for sending messages to local threads. Should be used whenever it is absolutely clear that in the overwhelming majority of all invocations:

- a send phase is included; *and*
- the destination thread is in the same address space as the sender; *and*
- a receive phase is included; *and*
- the destination thread runs on the same processor; *and*
- the ReceiveBlock flag is set.

Asynchronous notification

Asynchronous notification is a mechanism for asynchronously notifying other threads of events. The kernel does not specify what events may be signalled, it is left to the application to define. Notification operations are designed to be fast and non-blocking for the sender.

Asynchronous notification involves delivering notify-bits to a recipient thread. Notify bits are delivered regardless of whether the recipient has invoked the corresponding IPC receive operation. Notify bits are accumulated in the thread's *NotifyBits* TCR. The accumulation process performs a bitwise OR of the signalled notify-bits to the recipient's *NotifyBits* TCR. The *NotifyBits* TCR is in the UTCB and may be checked by the recipient without invoking an IPC call. Clearing bits in the NotifyBits TCR must be done atomically.

A recipient can choose to allow or deny notifications by using the a flag in the *Acceptor* TCR. Sending a notification to a thread that is not accepting notifications causes the sender's IPC operation to fail with the *NotAccepted* error code.

An asynchronous notification send is specified by setting then n flag in the IPC message tag. When this is the case, the kernel interprets MR_1 to be the notify_word; the set of notification bits to sent to the recipient. The kernel delivers notify bits by performing a bitwise OR operation; $NotifyBits \mid= notify\_word$.

A thread can wait on set of notification bits in an IPC call by specifying an appropriate IPC receive phase. The *NotifyMask* TCR is used to control an asynchronous IPC receive operation and specifies the set of requested notification bits. The kernel determines whether any requested notification bits are present by testing for non zero result of $NotifyBits \ \& \ NotifyMask$.

There are two methods for waiting for notification bits via IPC: An IPC that specifies either $from = waitnotify$ or $from = anythread$. Note that a must be set in the *Acceptor* for both cases.

When $from = waitnotify$, the thread will block until any of the requested notification bits are received. If one or more requested bits are already in the *NotifyBits* TCR, the IPC operation will return immediately.

When $from = anythread$, the thread will block until either a message is received from another thread or when one or more requested notification bits are received. Note that pending synchronous messages have higher priority than pending notification bits and will be received first.

When an IPC operation receives notification bits, the set of requested bits (*NotifyMask*) that were received are returned in MR $_1$. The kernel atomically clears the these bits from the *NotifyBits* TCR. That is: $NotifyBits \ \& \ NotifyMask$ is returned and then $NotifyBits \ \&= \sim NotifyMask$.

Note, it is not possible to determine the origin of notification bits. Thus a thread may only specify $from = waitnotify$ or $from = anythread$ in an IPC operation to receive notifications. This also means that asynchronous notifications can not be sent from a thread with a send redirector or receive redirector set. The IPC will fail with the *NotAccepted* error code.

The kernel associates no semantics with any notification bits, this is left to the application to define. As an additional optimization, the user may read the *NotifyMask* TCR to determine if any notification bits are pending instead of invoking an IPC. *NotifyMask* can only be modified by usert with an atomic read-modify-write operation.

See page 73 for more details on receiving notifications.

Input Parameters

---

to = *nilthread* IPC includes no send phase.

to ≠ *nilthread* Destination thread; IPC includes a send phase

---

FromSpecifier = *nilthread*

IPC includes no receive phase.

FromSpecifier = *waitnotify*

IPC includes a receive phase. Incoming asynchronous notifications are accepted from any thread in the system. If the a-bit of the *Acceptor* is not set, the operation is *undefined*.

FromSpecifier = *anythread*

IPC includes a receive phase. Incoming messages are accepted from any thread (including hardware interrupts). Asynchronous notifications are received if the a flag is set in the *Acceptor*.

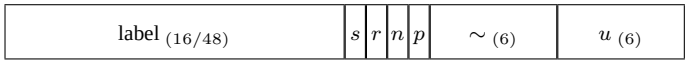
FromSpecifier = *anylocalthread*

IPC includes a receive phase. Incoming messages are accepted from any thread that resides in the current address space. Asynchronous notifications will not be received.

FromSpecifier ≠ *any of above*

IPC includes a receive phase. Incoming messages are accepted only from the specified thread. (Note that hardware interrupts can be specified.) Asynchronous notifications are not received from any thread including the specified thread.

MsgTag **[MR$_0$]**



Message head of the message to be sent. Only the upper 16/48 bits are freely available. The lower 16 bits contain IPC control information. They describe the message to be sent and specify whether the send and receive phases block or not.

u — Number of untyped words following word 0. MR $_{1...u}$ hold the untyped words. $u = 0$ denotes a message with no untyped words. If u is greater than the number of MRs supported by the kernel (x), the IPC operation will fail with the *Message Overflow* error code.

p=0 — Normal (unpropagated) send operation. The recipient gets the original sender's id.

p=1 — Propagating send operation. The *VirtualSender* TCR specifies the id of the originator thread. (i.e., the thread to send the message on behalf of). If originator thread and current sender, or current sender and receiver reside in the same address space, propagation is always permitted. Otherwise, IPC occurs unpropagated. Propagation is also allowed if the originator thread is an interrupt thread waiting (closed) for the current thread, or if the current sender is a redirector for the originator thread (or there exists a chain of redirectors from the originator to the current sender).
If propagation is permitted, the receiver receives the originator's id instead of the current sender's id, the *p* bit in the receiver's MsgTag is set, and the current sender's id is stored in the receiver's *ActualSender* TCR. If the originator thread is waiting (closed) for a reply from the current sender, the originator's state is additionally modified so that it now waits for the new receiver instead of the current sender.

n — An asynchronous notification send operation is requested.
If n is set, the s and u fields are ignored.

r — ReceiveBlock operation. When the IPC operation contains a receive phase and r is set, the receive phase will block if no valid incoming messages are pending. If r is clear, the receive phase does not block if no incoming messages are pending and the IPC fails with *No-partner*.

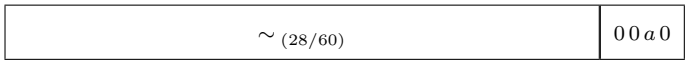
s — SendBlock operation. When the IPC operation contains a send phase, the send phase will block if the destination thread is not ready to accept messages from the sending thread. When this bit is clear and the destination thread is not ready, the IPC fails immediately with *No-partner*.

label — Freely available, often used to specify the request type or invoked method. This field is ignored by the kernel and transferred to the destination unmodified.

[MR$_1$] — When n is set, this contains the notify-bits to sent. Ignored if no send phase.

[MR$_{1...u}$] — Untyped words to be sent when n is clear. Ignored if no send phase.

Acceptor **[TCR]**



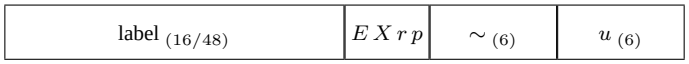
The acceptor specifies which IPC types are accepted when a message is received.

a — Asynchronous notifications are accepted iff $a = 1$.

Output Parameters

from Thread ID of the sender from which the IPC was received or the originator thread in the case of a propagated send operation.
Reception of asynchronous notifications is encoded as receiving a message from *nilthread* with the output *MsgTag E* error indicator cleared.
Only defined for IPC operations that include a receive phase.

MsgTag **[MR$_0$]**



If the IPC operation included a receive phase, MR$_0$ contains the message tag of the received message. The upper 16/48 bits contain the user-specified label. The lower bits describe the received message, contain the error indicator, and the cross-processor IPC indicator.
MR$_0$ is defined even if the IPC operation did not include a receive phase. In the send-only case, MR$_0$ returns the error indicator.

u Number of untyped words following word 0. $u = 0$ means no untyped words. For IPC operations without receive phase, $u = 0$ is delivered.

p Propagated IPC. If clear ($p = 0$) the IPC was not propagated. If set ($p = 1$) the IPC was propagated and the *FromSpecifier* indicates the originator thread's id. The *ActualSender* TCR specifies the id of the thread which performed the propagation.

r Redirected IPC. If clear ($r = 0$) the IPC was not a redirected one. If set ($r = 1$) the IPC was redirected to the current thread, and the *IntendedReceiver* TCR specifies the id of the thread supposed to receive the message. See page 24 for redirector details.

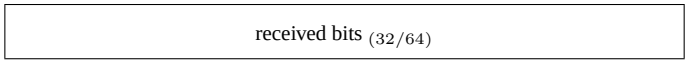
X Cross-processor IPC. If clear ($X = 0$) the received IPC came from a thread running on the same processor as the receiver. If set ($X = 1$) the received IPC was cross-processor. For IPC operations without receive phase, $X = 0$ is delivered.

E Error indicator. If clear ($E = 0$) the IPC operation completed successfully.
If set ($E = 1$) IPC failed. If the send phase was successful but an error occurred in the receive phase, the entire IPC fails. The error code and additional information can be retrieved from the ErrorCode TCR. The fields *label*, and u are valid if the error code signals a partially received message.

label Label of the received message. For IPC operations without receive phase, the label is 0.

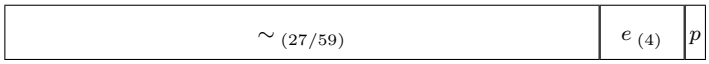
[MR$_{1 \ldots u}$] Untyped words that have been received. Undefined if no receive phase.

Received Bits **[MR$_1$]**



When an asynchronous notification is received via IPC, this field contains the set of returned bits ($NotifyBits \,\&\, NotifyMask$).

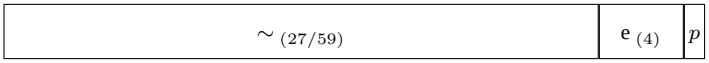
ErrorCode **[TCR]**



Only defined if the error indicator E in MR$_0$ is set. The p field specifies whether the error occurred during send or receive phase. If the error occurred during the receive phase the send phase (if any) was completed successfully before. If the error occurred during the send phase, the receive phase (if any) was skipped.

p Specifies whether the error occurred during the send phase ($p = 0$) or the receive phase ($p = 1$).

errors 1,2,3,5



Error happened before a partner thread was involved in the message transfer. Therefore, the error is signalled only to the thread that invoked the failing IPC operation.

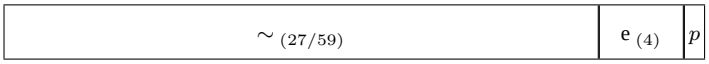
$e = 1$ *No-partner*.
From is undefined in this case. This occurs on (1) a non-blocking send operation to a thread not ready to receive a message from the caller, and (2) a non-blocking receive operation where no send operation is pending.

$e = 2$ *Non-existing* partner. If the error occurred in the send phase, *to* does not exist. (*Anythread* or *waitnotify* as a destination is illegal and will also raise this error.) If the error occurred in the receive phase, *FromSpecifier* does not exist. (*FromSpecifier* = *anythread* is legal, and thus will never raise this error.)

$e = 3$ *Cancelled* by another thread (system call *exchange registers*).

$e = 5$ *NotAccepted* an asynchronous notification was not accepted by another thread.

errors 4,7



A partner thread is already involved in the IPC operation, and the error is therefore signalled to both threads.

$e = 4$ *Message Overflow*.
A message overflow can occur if more MRs are required to send the IPC than are available.

$e = 7$ *Aborted* by another thread (see system call *exchange registers*).

Generic Programming Interface

System-Call Function:

#include <l4/ipc.h>

MsgTag ***Ipc*** (*ThreadId to, FromSpecifier, MsgTag tag, ThreadId& from*)

MsgTag ***Lipc*** (*ThreadId to, FromSpecifier, MsgTag tag, ThreadId& from*)

Note that message registers have read-once semantics and that returning the message tag implies reading MR_0. The contents of the message tag is therefore lost if the application does not implicitly store the return value of IPC or LIPC .

Convenience Programming Interface

Derived Functions:

#include <l4/ipc.h>

MsgTag ***Call*** (*ThreadId to*)
{ tag = L4_MsgTag (); Set_ReceiveBlock (&tag); Set_SendBlock (&tag); Ipc (to, to, tag, –); }

MsgTag ***Send*** (*ThreadId to*)
{ tag = L4_MsgTag (); Set_SendBlock (&tag); Ipc (to, nilthread, tag, –); }

MsgTag ***Reply*** (*ThreadId to*)
{ tag = L4_MsgTag (); Clear_SendBlock (&tag); Ipc (to, nilthread, tag, –); }

MsgTag ***Receive*** (*ThreadId from*)
{ tag = L4_MsgTag (); Set_ReceiveBlock (&tag); Ipc (nilthread, from, tag, –); }

MsgTag ***Wait*** (*ThreadId& from*)
{ tag = L4_MsgTag (); Set_ReceiveBlock (&tag); Ipc (nilthread, anythread, tag, from); }

MsgTag ***ReplyWait*** (*ThreadId to, ThreadId& from*)
{ tag = L4_MsgTag (); Set_ReceiveBlock (&tag); Clear_SendBlock (&tag); Ipc (to, anythread, tag, from); }

MsgTag ***Lcall*** (*ThreadId to*)
{ tag = L4_MsgTag (); Set_ReceiveBlock (&tag); Set_SendBlock (&tag); Lipc (to, to, tag, –); }

MsgTag ***LreplyWait*** (*ThreadId to, ThreadId& from*)
{ tag = L4_MsgTag (); Set_ReceiveBlock (&tag); Clear_SendBlock (&tag); Lipc (to, anylocalthread, tag, from); }

MsgTag ***Notify*** (*ThreadId to, Word& mask*)
{ tag = L4_MsgTag (); Clear_SendBlock (&tag); Set_Notify(&tag); Ipc (to, nilthread, tag, –); }

MsgTag ***WaitNotify*** (*Word& mask, ThreadId& from*)
{ tag = L4_MsgTag (); Set_ReceiveBlock (&tag); Ipc (nilthread, waitnotify, tag, from); }

Support Functions:

#include <l4/ipc.h>

Bool ***IpcSucceeded*** (*MsgTag t*)

Bool ***IpcFailed*** (*MsgTag t*)
Delivers the state of the error indicator (the E bit of MR_0).

Bool ***IpcPropagated*** (*MsgTag t*)

Bool ***IpcRedirected*** (*MsgTag t*)

Bool ***IpcXcpu*** (*MsgTag t*)
Checks if the IPC was propagated/redirected/cross CPU.

Word ***ErrorCode*** ()

ThreadId ***IntendedReceiver*** ()

ThreadId ***ActualSender*** ()
Delivers the error code/intended receiver TCR/actual sender.

void ***Set_Propagation*** (*MsgTag& t*)
Sets the propagation bit.

void ***Set_Notify*** (*MsgTag& t*)
Sets the asynchronous notification bit.

void ***Set_ReceiveBlock*** (*MsgTag& t*)
Sets the receive block bit.

void ***Clear_ReceiveBlock*** (*MsgTag& t*)
Clears the receive block bit.

void ***Set_SendBlock*** (*MsgTag& t*)
Sets the send block bit.

void ***Clear_SendBlock*** (*MsgTag& t*)
Clears the send block bit.

void ***Set_VirtualSender*** (*ThreadId t*)
Sets the virtual sender TCR.

Chapter 6

Miscellaneous

6.1 ExceptionHandler [TCR]

An exception handler thread can be installed to receive exception IPCs.

ExceptionHandler

≠nilthread	Specifies the exception handler thread. When a thread raises an exception the kernel sends an exception IPC message on the thread's behalf to the thread's exception handler thread and waits for a response from the exception handler containing the instruction pointer where the thread should continue execution in MR_1. The format of the exception IPC message is architecture specific. The architectural registers of the faulting thread, TCRs, and the MRs containing the exception message are preserved.
=nilthread	No exception handler is specified. If an exception is raised the thread is halted and not scheduled anymore. *nilthread is the default value for newly created threads.*

Generic Programming Interface

```
#include <l4/thread.h>
```

ThreadId ***ExceptionHandler*** ()

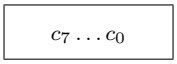
void ***Set_ExceptionHandler*** (*ThreadId new*)

Delivers/sets the exception handler TCR.

6.2 Cop Flags [TCR]

The *coprocessor flags* TCR helps the kernel to optimize thread switching for some hardware architectures.

Cop Flags



By resetting a c_i-bit to 0, a thread tells the system that it no longer needs coprocessor i. If the kernel finds $c_i = 0$, it concludes that registers and state of coprocessor i do not have to be saved. However, the kernel ensures that the coprocessor can not be used as a covert channel between different address spaces.
Once a thread has reset bit c_i it *must* set c_i to 1 *before* it issues the next operation on coprocessor i. Otherwise, coprocessor registers and state might be arbitrarily modified while using it.
Note that the c_i-bits are *write-only*. Reading them results in an undefined value. Upon thread creation, all c_i-bits are set to 1.

Generic Programming Interface

#include <l4/thread.h>

void ***Set_CopFlag*** (*Word n*)

void ***Clr_CopFlag*** (*Word n*)

Sets/clears coprocessor flag c_n.

6.3 PROCESSORCONTROL [Privileged Systemcall]

Word	*ProcessorNo*	⟶	*Word*	*result*
Word	*InternalFrequency*			
Word	*ExternalFrequency*			
Word	*voltage*			

Control the internal frequency, external frequency, or voltage for a system processor.

Input Parameters

ProcessorNo Specifies the processor to control. Number must be a valid index into the processor descriptor array (see Kernel Interface Page, page 4).

All further input parameters have no effect if the supplied value is -1, ensuring that the corresponding value is *not* modified. The following description always refers to values $\neq -1$.

InternalFrequency Sets internal frequency for processor to the given value (in kHz).

ExternalFrequency
Sets external frequency for processor to the given value (in kHz).

voltage Sets voltage for processor to the given value (in mV). A value of 0 shuts down the processor.

Output Parameters

result The result is 1 if the operation succeeded, otherwise the result is 0 and the ErrorCode TCR indicates the failure reason.

ErrorCode **[TCR]** Set if *result* $= 0$. Undefined if *result* $\neq 0$.

$= 1$ No privilege. Current thread does not have privilege to perform operation.

Note that the active internal and external frequency of all processors are available to all threads via the kernel interface page.

Pagefaults

No pagefaults will happen.

Generic Programming Interface

System-Call Function:

#include <l4/misc.h>

Word ***ProcessorControl*** (*Word ProcessorNo, InternalFrequency, ExternalFrequency, voltage*)

Convenience Programming Interface

Support Functions:

Word ***ErrorCode*** ()

Word ***ErrNoPrivilege***

6.4 CACHECONTROL [Systemcall]

ThreadId *SpaceSpecifier* *Word* ⟶ *Word* *result*
Word *Control*

This system call is used to perform operations on processor caches. Typically, *CacheControl* is used to perform cache-flushing operations, cache locking/unlocking or cache configuration. Additional arguments in the form of address ranges are passed in message registers (MRs).

Input Parameters

SpaceSpecifier specifies the target address space to operate on, by specifying a thread in the address space. The *SpaceSpecifier* thread must exist and reside in a completely initialized address space.

Control specifies the operations which are required.

$0_{(14/46)}$	$lx_{(6)}$	$op_{(6)}$	$n_{(6)}$

n Specifies the highest numbered address range that is located in the message registers. Address ranges occupy 2 MRs each and are located in registers $\mathrm{MR}_{0 \ldots 2n+1}$.

op Specifies the type of cache operation to be performed.

lx The cache level mask to operate on. Where bit 0 corresponds to the L1 cache, bit 1 to the L2 cache and so on.

Cache operations

0 Architecture specific. Architecture specific data is passed in message registers.

1 Flush address ranges specified in the MRs.

4 Flush the entire instruction cache. (Clean + Invalidate). Any address ranges are ignored.

5 Flush the entire data cache. (Clean + Invalidate). Any address ranges are ignored.

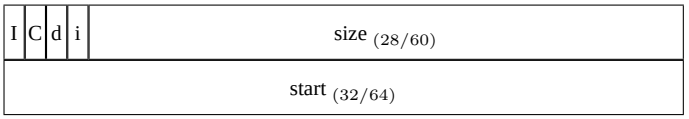
6 Flush the entire cache. (Clean + Invalidate). Any address ranges are ignored.

8 Lock the specified address ranges.

9 Unlock the specified address ranges.

AddressRangeList *MR* $_{0\ldots2n+1}$
Address ranges to be processed.

AddressRange *MR* $_{2n\ldots2n+1}$



$start$	Start address in the target address space.
$size$	Size of the range in bytes. Note that it is only possible to specify a range up to 256MB on a 32-bit processor.
i	If set, the range includes the instruction cache.
d	If set, the range includes the data cache.
C	If set, the cache range will be cleaned. Ignored for *lock*, *unlock* operations.
I	If set, the cache range will be invalidated. If C is also set, the range will be cleaned before invalidatation. Ignored for *lock*, *unlock* operations.

Output Parameters

result	The result is 1 if the operation succeeded, otherwise the result is 0 and the ErrorCode TCR indicates the failure reason.

ErrorCode **[TCR]** Set if *result* $= 0$. Undefined if *result* $\neq 0$.

$= 1$	No privilege. Current thread does not have privilege to perform operation.
$= 5$	Invalid parameter. One or more of the arguments provided was invalid.

Pagefaults

No pagefaults will happen.

Generic Programming Interface

System-Call Function:

#include <l4/cache.h>

Word ***CacheControl*** (*ThreadId SpaceSpecifier, Word control*)

Convenience Programming Interface

Derived Functions:

Word ***CacheFlushAll*** ()
Flushes the entire processor cache. (Clean + Invalidate).

Word ***CacheFlushRange*** (*ThreadId s, Word start, end*)
Flushes the cache over the specified address range in address space s. (Clean + Invalidate).

Word ***CacheFlushIRange*** (*ThreadId s, Word start, end*)
Flushes the I-cache over the specified address range in address space s. (Clean + Invalidate).

Word ***CacheFlushDRange*** (*ThreadId s, Word start, end*)
Flushes the D-cache over the specified address range in address space s. (Clean + Invalidate).

Word ***CacheFlushRangeInvalidate*** (*ThreadId s, Word start, end*)
Invalidates the cache over the specified address range in address space s.

Support Functions:

Word ***ErrorCode*** ()

Word ***ErrNoPrivilege***

Word ***ErrInvalidParam***

Chapter 7

Protocols

7.1 Thread Start Protocol [Protocol]

Newly created active threads start immediately by receiving a message from its pager. The received message contains the initial instruction-pointer and stack-pointer for the thread.

From Pager

Initial SP $_{(32/64)}$				MR $_2$
Initial IP $_{(32/64)}$				MR $_1$
0 $_{(16/48)}$	0 $_{(4)}$	$\sim$ $_{(6)}$	$u = 2$ $_{(6)}$	MR $_0$

7.2 Interrupt Protocol [Protocol]

Interrupts are delivered as an IPC call to the interrupt handler thread (i.e., the pager of the interrupt thread). The interrupt is disabled until the interrupt handler sends a re-enable message.

From Interrupt Thread

-1 $_{(12/44)}$	0 $_{(4)}$	0 $_{(4)}$	$\sim$ $_{(6)}$	$u = 0$ $_{(6)}$	MR$_0$

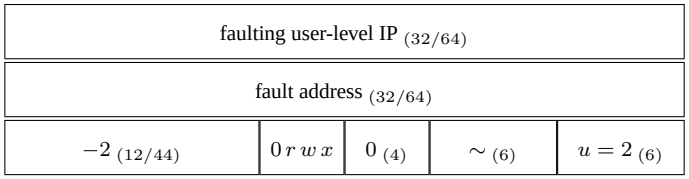
To Interrupt Thread

0 $_{(16/48)}$	0 $_{(4)}$	$\sim$ $_{(6)}$	$u = 0$ $_{(6)}$	MR$_0$

7.3 Pagefault Protocol [Protocol]

A thread generating a pagefault will cause the kernel to transparently generate a pagefault IPC to the faulting thread's pager. The behavior of the faulting thread is undefined if the pager does not exactly follow this protocol.

To Pager



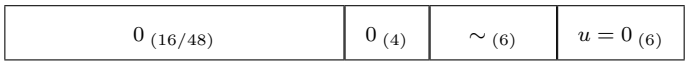
rwx

The rwx bits specify the fault reason:

r	read fault
w	write fault
x	execute fault

A bit set to one reports the type of the attempted access. On processors that do not differentiate between read and execute accesses, x is never set. Read and execute accesses will both be reported by the r bit.

From Pager



7.4 Preemption Protocol [Protocol]

From Preempted Thread

$-3_{\,(12/44)}$	$0_{\,(4)}$	$0_{\,(4)}$	$\sim_{\,(6)}$	$u = 0_{\,(6)}$	$\mathrm{MR}_{\,0}$

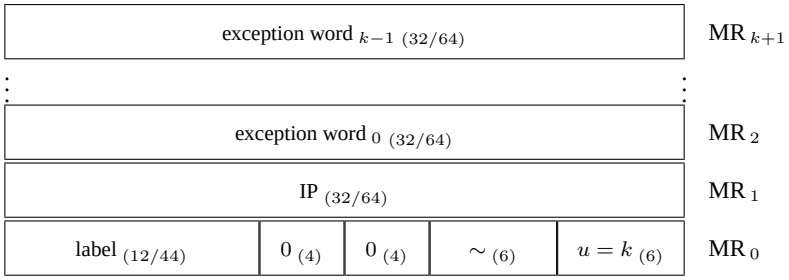
If the message can not be delivered the thread blocks until the receiver is ready.

7.5 Exception Protocol [Protocol]

The exception IPC contains a label, the faulting instruction pointer, and additional architecture specific exception words. The reply from the exception handler contains a label, an instruction pointer where the faulting thread is resumed, and an optional number of additional architecture specific words.

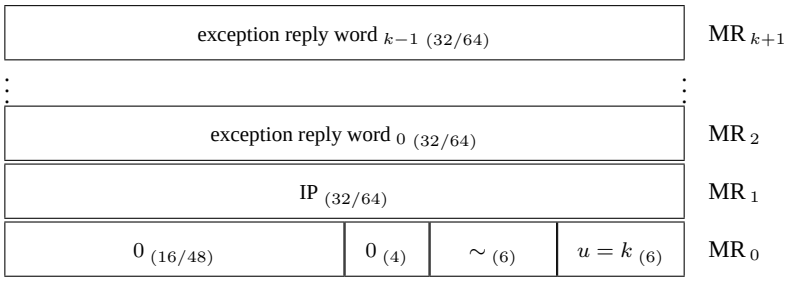
Note that the stack pointer is not explicitly specified to allow architecture specific optimizations.

To Exception Handler



k	Number of exception words.
label	specifies the exception type.
$= -4$	System exceptions are defined for all architectures.
$= -5$	Architecture specific exceptions.

From Exception Handler



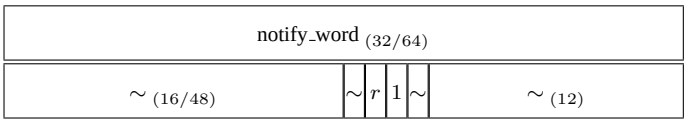
k	Number of exception reply words.
IP	Location where execution is resumed in the faulting thread.

If the reply from exception handler message is not defined by the architecture, it has the same format as the corresponding exception IPC message.

7.6 Asynchronous Notification Protocol [Protocol]

Asynchronous notifications are sent to a thread via a special type of IPC as described below and on page 51.

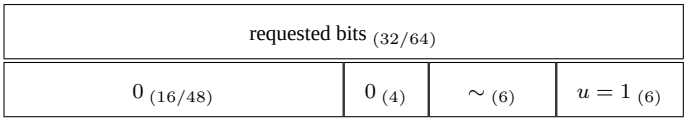
Sending notification to a thread



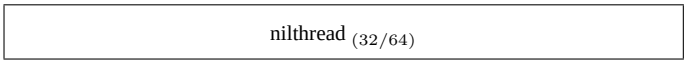
r If the IPC contains a receive phase, r specifies a ReceiveBlock operation.

Receiving an asynchronous notification is indicated and detected by receiving a valid message from *nilthread*.

Receiving a notification



from



7.7 Generic Booting [Protocol]

Machine-specific boot procedures are described on pages 91 ff.

After booting, L4 initializes itself. It generates the basic *root server* which is intended to boot the higher-level system.

The *root server* is a user-level server and not part of the pure kernel. The predefined one can be replaced by modifying the following table in the L4 image before starting L4. The kernel debugger *kdebug* is also not part of the kernel and can accordingly be replaced by modifying the table.

		MemoryDesc		MemDescPtr
∼	BootInfo	∼		+B0 / +160
∼				+A0 / +140
∼				+90 / +120
∼				+80 / +100
∼				+70 / +E0
∼				+60 / +C0
Kdebug.config1	Kdebug.config0	MemoryInfo	∼	+50 / +A0
root server.high	root server.low	root server.IP	root server.SP	+40 / +80
∼.high	∼.low	∼.IP	∼.SP	+30 / +60
∼.high	∼.low	∼.IP	∼.SP	+20 / +40
Kdebug.high	Kdebug.low	Kdebug.entry	Kdebug.init	+10 / +20
∼		API Version	$\sim_{(0/32)}$ 'K' 230 '4' 'L'	+0
+C / +18	+8 / +10	+4 / +8	+0	

The addresses are offsets relative to the configuration page's base address. The configuration page is located at a page boundary and can be found by searching for the magic "L4μK" starting at the load address. The IP and SP values however, are absolute addresses. The appropriate code must be loaded at these addresses before L4 is started.

IP Physical address of a server's initial instruction pointer (start).

SP Physical address of a server's initial stack pointer (stack bottom).

Kdebug.init Physical address of *kdebug*'s initialization routine.

Kdebug.entry Physical address of *kdebug*'s exception handler entry point.

Kdebug.low Physical address of first byte of kernel debugger. Must be page aligned.

Kdebug.high Physical address of last byte of kernel debugger. Must be the last byte in page.

Kdebug.config Configuration fields which can be freely interpreted by the kernel debugger. The specific semantics of these fields are provided with the specific kernel debuggers.

BootInfo Prior to kernel initialization a boot loader can write an arbitrary value into this field. Post-initialization code, e.g., a root server can later read the field. Its value is neither changed nor interpreted by the kernel. This is the generic method for passing system information across kernel initialization.

MemoryInfo

MemDescPtr $_{(16/32)}$	n $_{(16/32)}$

$MemDescPtr$ Location of first memory descriptor (as an offset relative to the configuration page's base address). Subsequent memory descriptors are located directly following the first one. For memory descriptors that specify overlapping memory regions, later descriptors take precedence over earlier ones.

n Initially equals the number of available memory descriptors in the configuration page. Before starting L4 this number must be initialized to the number of inserted memory descriptors.

MemoryDesc

$high/2^{10}$ $_{(22/54)}$		$\sim$ $_{(10)}$			+4 / +8
$low/2^{10}$ $_{(22/54)}$	v	$\sim$	t $_{(4)}$	$type$ $_{(4)}$	+0

Memory descriptors should be initialized before starting L4. The kernel may after startup insert additional memory descriptors or modify existing ones (e.g., for reserved kernel memory).

$high$ Address of last byte in memory region. The ten least significant address bits are all hardwired to 1.

low Address of first byte in memory region. The ten least significant address bits are all hardwired to 0.

v Indicates whether memory descriptor refers to physical memory ($v = 0$) or virtual memory ($v = 1$).

$type$ Identifies the type of the memory descriptor.

Type	Description
0x0	Undefined
0x1	Conventional memory
0x2	Reserved memory (i.e., reserved by kernel)
0x3	Dedicated memory (i.e., device memory)
0x4	NUMA memory (i.e., shared across a NUMA machine)
0x5	Global memory (i.e., always accessible)
0xB	Tracebuffer memory
0xE	Defined by boot loader
0xF	Architecture dependent

t Identifies the precise type for boot loader specific or architecture dependent memory descriptors.

$type = 0xB$

The memory region is occupied by a tracebuffer used by the kernel.

$type = 0xE$
: The type of the memory descriptor is dependent on the bootloader. The t field specifies the exact semantics. Refer to boot loader specification for more info.

$type = 0xF$
: The type of the memory descriptor is architecture dependent. The t field specifies the exact semantics. Refer to architecture specific part for more info.

$type \neq 0xE,\ type \neq 0xF$
: The type of the memory descriptor is solely defined by the $type$ field. The content of the t field is undefined.

7.8 Tracebuffer [Protocol]

Depending on the kernel configuration, a tracebuffer may be provided by the kernel which is useful for debugging, profiling and performance monitoring. The tracebuffer contains a descriptor and one or more flip-buffers.

The kernel contains a number of tracepoints. Each tracepoint has a major and minor number and a trace-id. The tracebuffer implementation allows specific major numbers to be enabled and disabled individually, thus allowing groups of tracepoints to be enabled and disabled.

A user level operating system or trace-server can read the tracebuffer and process statistics or log the data depending on its needs. If the tracebuffer is implemented, the kernel provides a virtual interrupt source to which the user can register in order to receive interrupts whenever the kernel switches flip-buffers.

The user must free the flip-buffers after processing to allow the kernel to reuse them. If the kernel runs out of available flip-buffers, it will not log subsequent tracepoints until a new buffer is available.

The user can determine whether the kernel implements a tracebuffer by looking for a memory descriptor with the *Tracebuffer memory* type. This memory descriptor describes the physical memory address range used by the tracebuffer. See page 6.

Appendix A

IA-32 Interface

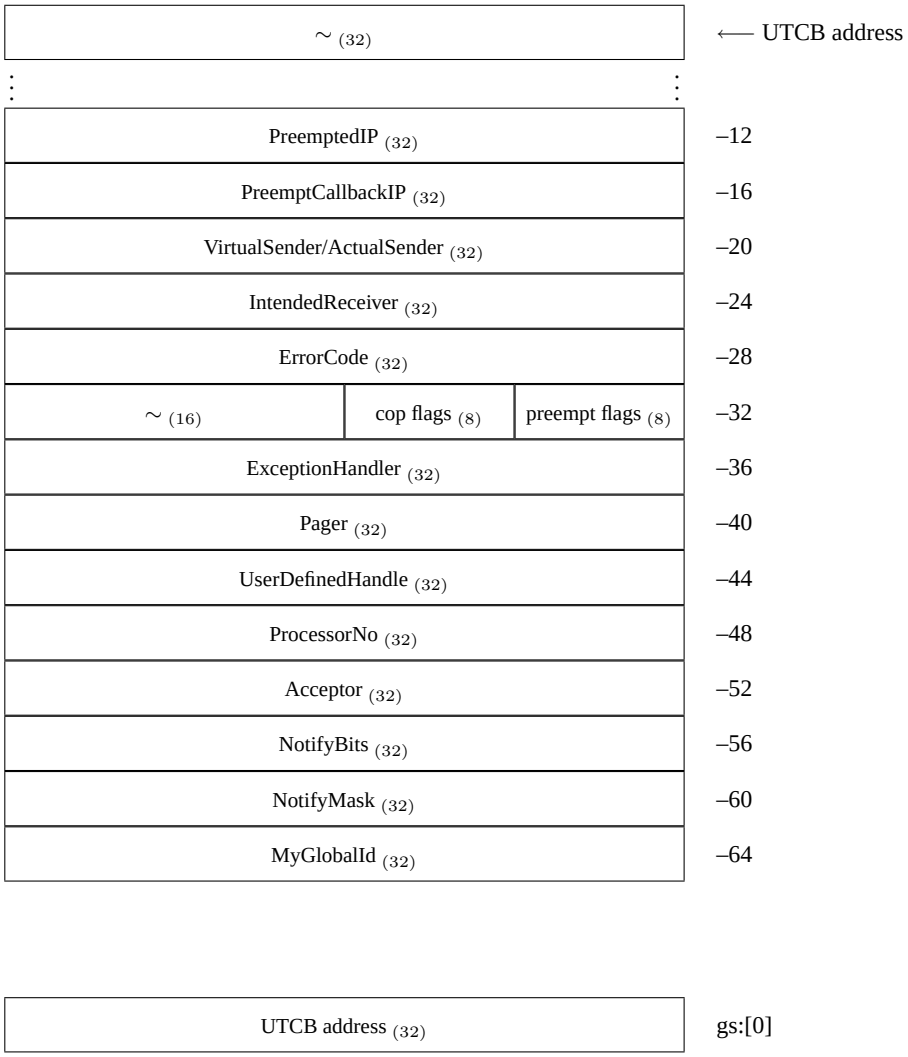
A.1 Virtual Registers [ia32]

Thread Control Registers (TCRs)

TCRs are implemented as part of the ia32-specific user-level thread control block (UTCB). The address of the current thread's UTCB will not change over the lifetime of the thread. Setting the UTCB address of an active thread via THREADCONTROL is similar to deletion and re-creation. There is a fixed correlation between the UtcbLocation parameter when invoking THREADCONTROL and the UTCB address. The UTCB address of the current thread can be loaded through a machine instruction

```
mov    %gs:[0], %r
```

UTCB objects of the current thread can then be accessed as any other memory object. UTCBs of other threads must not be accessed, even if they are physically accessible.



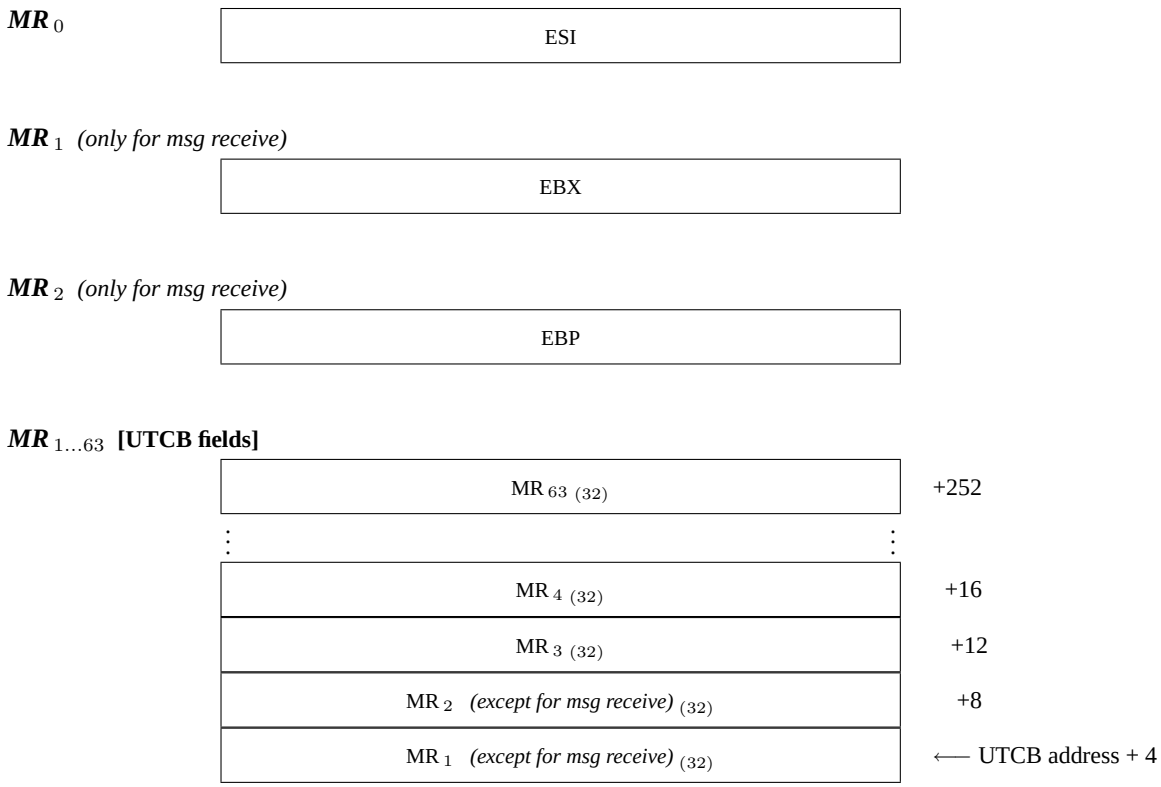
Message Registers (MRs)

Memory-mapped MRs are implemented as part of the ia32-specific user-level thread control block (UTCB). The address of the current thread's UTCB will not change over the lifetime of the thread. Setting the UTCB address of an active thread via THREADCONTROL is similar to deletion and re-creation. There is a fixed correlation between the UtcbLocation parameter when invoking THREADCONTROL and the UTCB address. The UTCB address of the current thread can be loaded through a machine instruction

```
mov    %gs:[0], %r
```

UTCB objects of the current thread can then be accessed as any other memory object. UTCBs of other threads must not be accessed, even if they are physically accessible.

MR_0 is always mapped to a general register. MR_1 and MR_2 are mapped to general registers when reading a received message; in all other cases, MR_1 and MR_2 are mapped to memory locations. $MR_{3\ldots63}$ are always mapped to memory.



UTCB Memory With Undefined Semantics

The kernel will associate no semantics with memory located at *UTCB address*...*UTCB address* + 3. The application can use this memory as thread local storage, e.g., for implementing the L4 API. Note, however, that the memory contents within this region may be overwritten during a system-call operating on message registers.

Note, depending on kernel configuration, not all 64 message registers may be available. In this case, no semantics are associated with the memory defined for the unused MRs as above. Note also that when fewer message registers are configured, the kernel may reduce the UTCB size such that memory locations beyond the highest usable message register may not be accessible.

All undefined UTCB memory which is not covered by the above mentioned region may have kernel defined semantics.

A.2 Systemcalls [ia32]

The system-calls which are invoked by the call instruction take the target of the calls from the system-call link fields in the kernel interface page (see page 2). Each system-call link specifies an address relative to the kernel interface page's base address. An application may use instructions other than call to invoke the system-calls, but must ensure that a valid return address resides on the stack.

KERNELINTERFACE [Slow Systemcall]

–	EAX	– **KernelInterface** →	EAX	*base address*
–	ECX		ECX	*API Version*
–	EDX		EDX	*API Flags*
–	ESI	lock: nop	ESI	*Kernel ID*
–	EDI		EDI	≡
–	EBX		EBX	≡
–	EBP		EBP	≡
–	ESP		ESP	≡

EXCHANGEREGISTERS [Systemcall]

dest	EAX	– **Exchange Registers** →	EAX	*result*
control	ECX		ECX	*control*
SP	EDX		EDX	*SP*
IP	ESI	call *ExchangeRegisters*	ESI	*IP*
FLAGS	EDI		EDI	*FLAGS*
UserDefinedHandle	EBX		EBX	*UserDefinedHandle*
pager	EBP		EBP	*pager*
–	ESP		ESP	≡

"*FLAGS*" refers to the user-modifiable ia32 processor flags that are held in the EFLAGS register.

THREADCONTROL [Privileged Systemcall]

dest	EAX	– **Thread Control** →	EAX	*result*
Pager	ECX		ECX	∼
Scheduler	EDX		EDX	∼
SpaceSpecifier	ESI	call *ThreadControl*	ESI	∼
SendRedirector	EDI		EDI	∼
RecvRedirector	EBX		EBX	∼
UtcbLocation	EBP		EBP	∼
–	ESP		ESP	≡

THREADSWITCH [Systemcall]

dest	EAX	– **ThreadSwitch** →	EAX	≡
–	ECX		ECX	≡
–	EDX		EDX	≡
–	ESI	call *ThreadSwitch*	ESI	≡
–	EDI		EDI	≡
–	EBX		EBX	≡
–	EBP		EBP	≡
–	ESP		ESP	≡

SCHEDULE [Systemcall]

dest	EAX	– **Schedule** →	EAX	*result*
ts len	ECX		ECX	*rem ts*
total quantum	EDX		EDX	*tem total*
processor control	ESI	call *Schedule*	ESI	∼
prio	EDI		EDI	∼
–	EBX		EBX	∼
–	EBP		EBP	∼
–	ESP		ESP	≡

IPC [Systemcall]

to	EAX	– **Ipc** →	EAX	*from*
–	ECX		ECX	∼
FromSpecifier	EDX		EDX	∼
MR_0	ESI	call *Ipc*	ESI	MR_0
MR_1	EDI		EDI	MR_1
–	EBX		EBX	*UTCB*
MR_2	EBP		EBP	MR_2
–	ESP		ESP	≡

LIPC [Systemcall]

to	EAX	– **Lipc** →	EAX	*from*
–	ECX		ECX	∼
FromSpecifier	EDX		EDX	∼
MR_0	ESI	call *Lipc*	ESI	MR_0
MR_1	EDI		EDI	MR_1
–	EBX		EBX	*UTCB*
MR_2	EBP		EBP	MR_2
–	ESP		ESP	≡

MAPCONTROL [Systemcall]

SpaceSpecifier	EAX	– **MapControl** →	EAX	*result*
∼	ECX		ECX	∼
control	EDX		EDX	∼
MR_0	ESI	call *MapControl*	ESI	MR_0
MR_1	EDI		EDI	MR_1
–	EBX		EBX	*UTCB*
MR_2	EBP		EBP	MR_2
–	ESP		ESP	≡

SPACECONTROL [Privileged Systemcall]

SpaceSpecifier	EAX	– **Space Control** →	EAX	*result*
control	ECX		ECX	*control*
KernelInterfacePageArea	EDX		EDX	∼
UtcbArea	ESI	call *SpaceControl*	ESI	∼
–	EDI		EDI	∼
–	EBX		EBX	∼
–	EBP		EBP	∼
–	ESP		ESP	≡

PROCESSORCONTROL [Privileged Systemcall]

ProcessorNo	EAX	– **Processor Control** →	EAX	*result*
InternalFrequency	ECX		ECX	∼
ExternalFrequency	EDX		EDX	∼
voltage	ESI	call *ProcessorControl*	ESI	∼
–	EDI		EDI	∼
–	EBX		EBX	∼
–	EBP		EBP	∼
–	ESP		ESP	≡

A.3 Kernel Features [ia32]

The ia32 architecture supports the following kernel feature descriptors in the kernel interface page (see page 5).

String	Feature
“smallspaces”	Kernel has small address spaces enabled.

A.4 IO-Ports [ia32]

On ia32 processors, IO-ports are handled as fpages. IO fpages can be mapped and unmapped like memory fpages. Their minimal granularity is 1. An IO-fpage of size $2^{s'}$ has a $2^{s'}$-aligned base address p, i.e. $p \bmod 2^{s'}=0$. An fpage with base port address p and size $2^{s'}$ is denoted as described below.

IO fpage $(p, 2^{s'})$

p (16/48)	s' (6)	$s=2$ (6)	$0\,r\,w\,x$

IO-ports can only be mapped idempotently, i.e., physical port x is either mapped at IO address x in the task's IO address space, or it is not mapped at all.

Generic Programming Interface

#include <l4/space.h>

Fpage ***IoFpage*** (*Word BaseAddress, int FpageSize*)

Fpage ***IoFpageLog2*** (*Word BaseAddress, int Log2FpageSize* < 64)

Delivers an IO fpage with the specified location and size.

A.5 Space Control [ia32]

The SPACECONTROL system call has an architecture dependent *control* parameter to specify various address space characteristics. For ia32, the *control* parameter has the following semantics.

Input Parameter

control

s A value of 1 indicates the intention to change the *small address space number* for the specified address space. The small space number will remain unchanged if $s = 0$.

small If $s = 1$, sets the small address space number for the specified address space. Small address space numbers from 1 to 255 are available. A value of 0 indicates a regular large address space. An assigned small space number is effective on *all* CPUs in an SMP system.
The position (pos) of the least significant bit of *small* indicates the size of the small space by the following formula: $size = 2^{pos} * 4$ MB. After removing the least significant bit, the remaining bits of *small* indicate the location of the space within a 512 MB region using the following formula: $location = small * 2$ MB. Setting the small space number fails if the specified region overlaps with an already existing one.
The *small* field is ignored if $s = 0$, or if the kernel does not support small spaces (see Kernel Features, page 85).

Output Parameter

control

e Indicates if the change of small space number was effective ($e = 1$). Undefined if $s = 0$ in the input parameter.

small The old value for the small space number. A value of 0 is possible even if the space has previously been put into a small address space. An implicit change to small space number 0 can happen if a thread within the space accesses memory beyond the specified small space size.

Generic Programming Interface

```
#include <l4/space.h>

Word LargeSpace

Word SmallSpace (Word location, size)
```

Delivers a small space number with the specified *location* and *size* (both in MB). It is assumed that $size = 2^p * 4$ for some value $p < 8$.

A.6 Memory Attributes [ia32]

The IA-32 architecture implements or emulates all generic attributes see 36. Note that some attributes are only supported on certain processors. See the “IA-32 Intel Architecture Software Developer’s Manual, Volume 3: System Programming Guide” for the semantics of the memory attributes and which processors they are supported on.

Generic Programming Interface

```
#include <l4/misc.h>

Word DefaultMemory
Word UncacheableMemory
Word WriteCombiningMemory
Word WriteThroughMemory
Word WriteProtectedMemory
Word WriteBackMemory
```

A.7 Exception Message Format [ia32]

To Exception Handler

EAX $_{(32)}$					MR $_{12}$
ECX $_{(32)}$					MR $_{11}$
EDX $_{(32)}$					MR $_{10}$
EBX $_{(32)}$					MR $_{9}$
ESP $_{(32)}$					MR $_{8}$
EBP $_{(32)}$					MR $_{7}$
ESI $_{(32)}$					MR $_{6}$
EDI $_{(32)}$					MR $_{5}$
ErrorCode $_{(32)}$					MR $_{4}$
ExceptionNo $_{(32)}$					MR $_{3}$
EFLAGS $_{(32)}$					MR $_{2}$
EIP $_{(32)}$					MR $_{1}$
$-4/-5$ $_{(12/44)}$	0 $_{(4)}$	0 $_{(4)}$	$t = 0$ $_{(6)}$	$u = 12$ $_{(6)}$	MR $_{0}$

#PF (page fault), #MC (machine check exception), and some #GP (general protection), #SS (stack segment fault), and #NM (no math coprocessor) exceptions are handled by the kernel and therefore do not generate exception messages.

Note that executing an INT n instructions in 32-bit mode will always raise a #GP (general protection). The exception handler may interpret the error code ($8n + 2$, see processor manual) and emulate the INT n accordingly.

A.8 Processor Mirroring [ia32]

Segments

L4 uses a flat (unsegmented) memory model. There are only three segments available: *user_space*, a read/write segment, *user_space_exec*, an executable segment, and *utcb_address*, a read-only segment. Both *user_space* and *user_space_exec* cover (at least) the complete user-level address space. *Utcb_address* covers only enough memory to hold the UTCB address.

The values of segment selectors *are undefined*. When a thread is created, its segment registers SS, DS, ES and FS are initialized with *user_space*, GS with *utcb_address*, and CS with *user_space_exec*. Whenever the kernel detects a general protection exception and the segment registers are not loaded properly, it reloads them with the above mentioned selectors. From the user's point of view, the segment registers cannot be modified.

However, the binary representation of *user_space* and *user_space_exec* may change at any point during program execution. Never rely on any particular value.

Furthermore, the LSL (load segment limit) machine instruction may deliver wrong segment limits, even floating ones. The result of this instruction is always *undefined*.

Debug Registers

User-level debug registers exist per thread. DR0...3, DR6 and DR7 can be accessed by the machine instructions mov n,DRx and mov DRx,r. However, only task-local breakpoints can be activated, i.e., bits G0...3 in DR7 cannot be set. Breakpoints operate per thread. Breakpoints are signaled as #DB exception (INT 1).

Note that user-level breakpoints are suspended when kernel breakpoints are set by the kernel debugger.

Model-Specific Registers

All privileged threads in the system have read and write access to all the Model-Specific Registers (MSRs) of the CPU. Modification of some MSRs may lead to undefined system behavior. Any access to an MSR by an unprivileged thread will raise an exception.

A.9 Booting [ia32]

PC-compatible Machines

L4 can be loaded at any 16-byte-aligned location beyond 0x1000 in physical memory. It can be started in real mode or in 32-bit protected mode at address 0x100 or 0x1000 relative to its load address. The protected-mode conditions are compliant to the Multiboot Standard Version 0.6.

Start Preconditions		
	Real Mode	32-bit Protected Mode
load base (L)	$L \geq$ 0x1000, 16-byte aligned	$L \geq$ 0x1000
load offset (X)	$X =$ 0x100 or $X =$ 0x1000	$X =$ 0x100 or $X =$ 0x1000
Interrupts	disabled	disabled
Gate A20	$\sim$	open
EFLAGS	I=0	I=0, VM=0
CR0	PE=0	PE=1, PG=0
(E)IP	X	$L + X$
CS	$L/16$	0, 4GB, 32-bit exec
SS,DS,ES	$\sim$	0, 4GB, read/write
EAX	$\sim$	0x2BADB002
EBX	$\sim$	$^*P$
$\langle P + 0 \rangle$	n/a	$\sim$ OR 1
$\langle P + 4 \rangle$		below 640 K mem in K
$\langle P + 8 \rangle$		beyond 1M mem in K
all remaining registers & flags (general, floating point, ESP, xDT, TR, CRx, DRx)	$\sim$	$\sim$

L4 relocates itself to 0x1000, enters protected mode if started in real mode, enables paging and initializes itself.

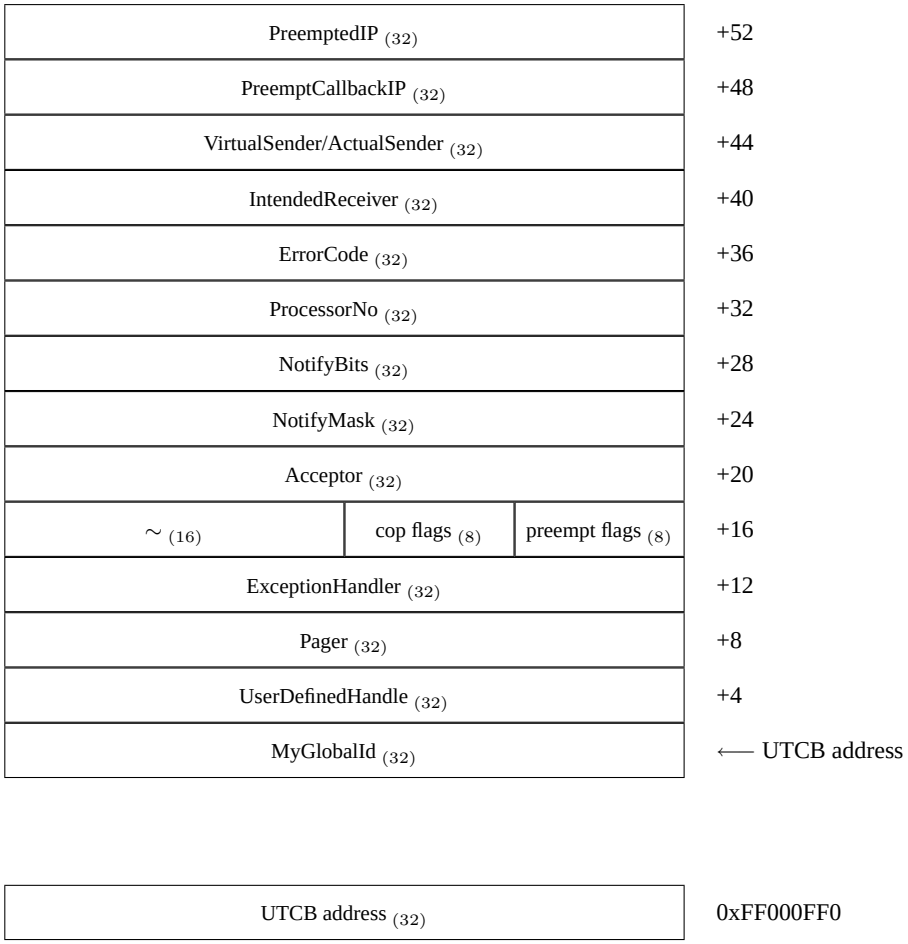
Appendix B

ARM Interface

B.1 Virtual Registers [ARM]

Thread Control Registers (TCRs)

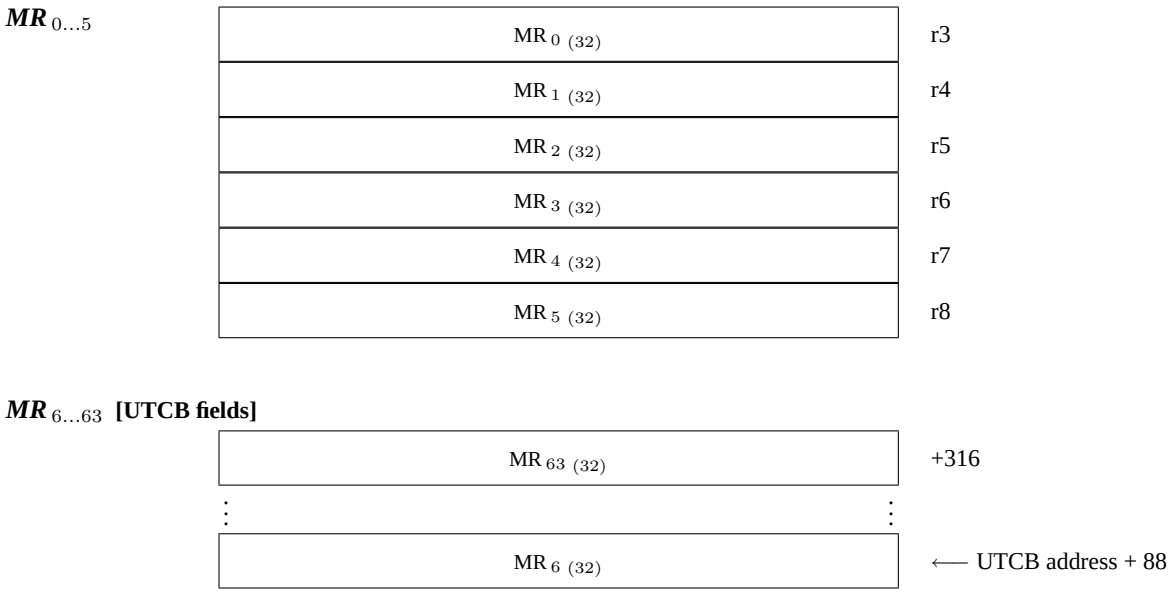
TCRs are mapped to memory locations. They are implemented as part of the ARM-specific user-level thread control block (UTCB). The address of the current thread's UTCB will not change over the lifetime of the thread. The UTCB address of the current thread can be read from the memory location 0xFF000FF0. UTCB objects of the current thread can then be accessed as any other memory object. UTCBs of other threads must not be accessed, even if they are physically accessible.



Message Registers (MRs)

Message registers MR_0 through MR_5 map to the processor's general purpose register file for IPC, LIPC and unmap calls. The remaining message registers map to memory locations in the UTCB. MR_5 starts at byte offset 84 in the UTCB, and successive message registers follow in memory.

The first six message registers are mapped to the registers r3 to r8. $MR_{6...63}$ are mapped to memory in the UTCB.



UTCB Memory With Undefined Semantics

The kernel will associate no semantics with memory located at *UTCB address* + 64...*UTCB address* + 87. The application can use this memory as thread local storage, e.g., for implementing the L4 API. Note, however, that the memory contents within this region may be overwritten during a system-call operating on message registers.

Note, that depending on kernel configuration, not all 64 message registers may be available. In this case, no semantics are associated with the memory defined for the unused MRs as above. Note also that when fewer message registers are configured, the kernel may reduce the UTCB size such that memory locations beyond the highest usable message register may not be accessible.

All undefined UTCB memory which is not covered by the above mentioned region may have kernel defined semantics.

B.2 Systemcalls [ARM]

The system-calls, which are invoked by the *bl* instruction, take the target of the calls from the system call link fields in the kernel interface page (see page 2). Each system-call link value specifies an address relative to the kernel interface page's base address. One may invoke the system calls with any instruction that branches to the appropriate target, as long as the return-address is contained in *r14*.

The locations of the system-calls are fixed during the life of an application, although they may change outside of the life of an application. It is not valid to prelink an application against a set of system call locations. The official locations are always provided in the KIP.

Note: For ease of debugging, it has been defined that KIP code will not modify the user stack pointer (*r13*) even if it needs to push data onto the stack.

The *s*p and *l*r registers are always preserved across system calls. Unless defined below, registers *r*8...r12 have undefined values following system calls other than KernelInterface.

KERNELINTERFACE [Slow Systemcall]

–	*r0*	– **KernelInterface** →	*r0*	*KIP base address*
–	*r1*		*r1*	*API Version*
–	*r2*		*r2*	*API Flags*
–	*r3*	mov sp, #-76; swi	*r3*	*Kernel ID*
–	*r4*		*r4*	≡
–	*r5*		*r5*	≡
–	*r6*		*r6*	≡
–	*r7*		*r7*	≡

For this system-call all registers other than the output registers are preserved.

EXCHANGEREGISTERS [Systemcall]

dest	*r0*	– **Exchange Registers** →	*r0*	*result*
control	*r1*		*r1*	*control*
SP	*r2*		*r2*	*SP*
IP	*r3*	bl *ExchangeRegisters*	*r3*	*IP*
FLAGS	*r4*		*r4*	*FLAGS*
UserDefinedHandle	*r5*		*r5*	*UserDefinedHandle*
pager	*r6*		*r6*	*pager*
–	*r7*		*r7*	∼

The *FLAGS* field corresponds to the ARM *CPSR* register.

THREADCONTROL [Privileged Systemcall]

dest	*r0*	– **Thread Control** →	*r0*	*result*
space	*r1*		*r1*	∼
scheduler	*r2*		*r2*	∼
pager	*r3*	bl *ThreadControl*	*r3*	∼
SendRedirector	*r4*		*r4*	∼
ReceiveRedirector	*r5*		*r5*	∼
UTCB	*r6*		*r6*	∼
–	*r7*		*r7*	∼

THREADSWITCH [Systemcall]

dest	*r0*	– **ThreadSwitch** →	*r0*	∼
–	*r1*		*r1*	∼
–	*r2*		*r2*	∼
–	*r3*	bl *ThreadSwitch*	*r3*	∼
–	*r4*		*r4*	∼
–	*r5*		*r5*	∼
–	*r6*		*r6*	∼
–	*r7*		*r7*	∼

SCHEDULE [Systemcall]

dest	*r0*	– **Schedule** →	*r0*	*result*
ts len	*r1*		*r1*	*rem ts*
total quantum	*r2*		*r2*	*rem total*
processor control	*r3*	bl *Schedule*	*r3*	∼
prio	*r4*		*r4*	∼
–	*r5*		*r5*	∼
–	*r6*		*r6*	∼
–	*r7*		*r7*	∼

IPC [Systemcall]

dest	*r0*	– **Ipc** →	*r0*	*result*
FromSpecifier	*r1*		*r1*	∼
–	*r2*		*r2*	∼
MR_0	*r3*	bl *Ipc*	*r3*	MR_0
MR_1	*r4*		*r4*	MR_1
MR_2	*r5*		*r5*	MR_2
MR_3	*r6*		*r6*	MR_3
MR_4	*r7*		*r7*	MR_4
MR_5	*r8*		*r8*	MR_5

LIPC [Systemcall]

dest	*r0*	– **Lipc** →	*r0*	*result*
FromSpecifier	*r1*		*r1*	∼
–	*r2*		*r2*	∼
MR_0	*r3*	bl *Lipc*	*r3*	MR_0
MR_1	*r4*		*r4*	MR_1
MR_2	*r5*		*r5*	MR_2
MR_3	*r6*		*r6*	MR_3
MR_4	*r7*		*r7*	MR_4
MR_5	*r8*		*r8*	MR_5

MAPCONTROL [Systemcall]

SpaceSpecifier	*r0*	– **MapControl** →	*r0*	*result*
control	*r1*		*r1*	∼
–	*r2*		*r2*	∼
MR_0	*r3*	bl *MapControl*	*r3*	MR_0
MR_1	*r4*		*r4*	MR_1
MR_2	*r5*		*r5*	MR_2
MR_3	*r6*		*r6*	MR_3
MR_4	*r7*		*r7*	MR_4
MR_5	*r8*		*r8*	MR_5

SPACECONTROL [Privileged Systemcall]

SpaceSpecifier	*r0*	– **Space Control** →	*r0*	*result*
control	*r1*		*r1*	*control*
KernelInterfacePageArea	*r2*		*r2*	∼
UtcbArea	*r3*	bl *SpaceControl*	*r3*	∼
–	*r4*		*r4*	∼
–	*r5*		*r5*	∼
–	*r6*		*r6*	∼
–	*r7*		*r7*	∼

PROCESSORCONTROL [Privileged Systemcall]

ProcessorNo	*r0*	– **Processor Control** →	*r0*	*result*
InternalFreq	*r1*		*r1*	∼
ExternalFreq	*r2*		*r2*	∼
voltage	*r3*	bl *ProcessorControl*	*r3*	∼
–	*r4*		*r4*	∼
–	*r5*		*r5*	∼
–	*r6*		*r6*	∼
–	*r7*		*r7*	∼

B.3 Kernel Features [ARM]

The ARM architecture supports the following kernel feature descriptors in the kernel interface page (see page 5).

String	Feature
“PIDs”	Kernel has ARM-PID support enabled.
“virtualspaceids”	Kernel has virtual-space identifiers enabled.

B.4 Memory Attributes [ARM]

The ARM architecture implements or emulates all generic attributes see 36.

Before disabling cache for a page, the software must ensure that all memory belonging to the target page is flushed from the cache.

B.5 Space Control [ARM]

The SPACECONTROL system call has an architecture dependent *control* parameter to specify various address space characteristics. For ARM, the *control* parameter has the following semantics.

Input Parameter

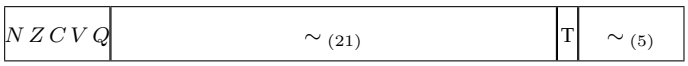
control

PID If the kernel has *ARM-PID* support, this sets the PID register value that will be loaded for threads in this address space. The effect of this is described in the Fast Context Switch Extension section of the ARM Architecture Reference Manual.
All addresses supplied to and returned from kernel syscalls (e.g. UTCB location) correspond to the MVA.

vspace If the kernel has *virtual-space identifiers* support, then the vspace field specifies the VirtualSpaceID of the current address space. Address spaces with the same VirtualSpaceID are defined as having no conflicting aliases of physical pages in their virtual address space. A typical example is a single-address-space operating system.
The L4 kernel can optimize address space switches for ARM virtual caches with knowledge of this address space relationship. It is up to the privileged services to enforce the non-conflicting address space layout. A violation of this rule will corrupt all address spaces with the same VirtualSpaceID and violate security.

B.6 Exchange Registers [ARM]

The EXCHANGEREGISTERS system call has an architecture dependent *FLAGS* parameter to specify various user-level CPU flags that can be controlled. For ARM, the *FLAGS* parameter has the same fields as the ARM *CPSR* register. Not all bits in the *CPSR* are controllable. The following shows which bits are valid.



B.7 Exception Message Format [ARM]

Syscall emulation exception message

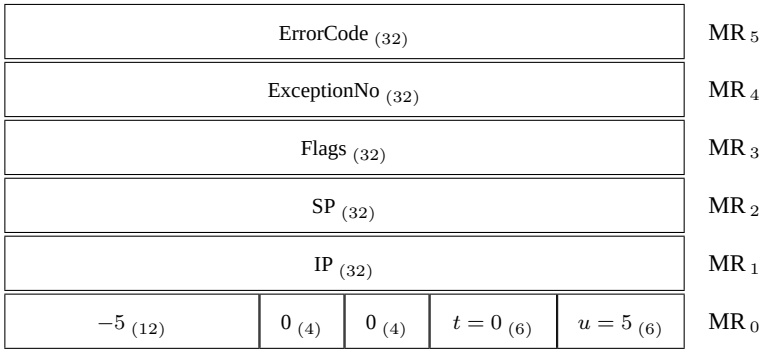
Flags $_{(32)}$					MR $_{13}$
Syscall $_{(32)}$					MR $_{12}$
LR $_{(32)}$					MR $_{11}$
SP $_{(32)}$					MR $_{10}$
PC $_{(32)}$					MR $_{9}$
r3 $_{(32)}$					MR $_{8}$
r2 $_{(32)}$					MR $_{7}$
r1 $_{(32)}$					MR $_{6}$
r0 $_{(32)}$					MR $_{5}$
r7 $_{(32)}$					MR $_{4}$
r6 $_{(32)}$					MR $_{3}$
r5 $_{(32)}$					MR $_{2}$
r4 $_{(32)}$					MR $_{1}$
-5 $_{(12)}$	0 $_{(4)}$	0 $_{(4)}$	$t = 0$ $_{(6)}$	$u = 13$ $_{(6)}$	MR $_{0}$

On execution of an ARM *SWI* instruction, the above message is delivered to the thread's exception handler.

The *Syscall* field contains the encoding of the instruction causing the system call exception. The exception handler can decode the system call number from the lower 24 bits.

Generic Traps

Generic Trap Message To Exception Handler



The kernel synthesizes exception messages in response to architecture specific events. Some traps are handled by the kernel and therefore do not generate exception messages. The kernel preserves all user state.

The following is a table of values for the Generic Trap *ExceptionNo*:

Exception	ExceptionNo	ErrorCode	Delivered
Undefined instruction	1	Instruction	Yes
Data abort	0x100 + (fault status)	Fault address	(external aborts/unhandled)
Reset exception			No
FIQ exception			No

Note, not all of these exceptions will be delivered via exception IPC. Some will be handled by the kernel. Delivered exceptions are indicated in the last column of the table above.

B.8 Thumb mode extensions [ARM]

On CPUs that support thumb mode, certain kernel operations are extended to provide support specifying the mode of operation.

In certain cases, the L4 kernel honors the mode-bit set in the LSB of an instruction-pointer. In these cases, when setting the instruction pointer of a thread, the thread's CPU mode is set to ARM mode if the LSB is clear, otherwise the thread's CPU mode is set to THUMB mode. The following is a list of kernel operations which comply.

- *Asynchronous preemption* see page 32. The LSB of the *PreemptCallbackIP* TCR is honored. The kernel also sets the LSB of the *PreemptedIP* with the thread's thumb state.
- *Exchange Registers*. The *IP* input field is honored. The LSB of the *IP* output is undefined. The *FLAGS* output value contains the correct value of the thumb bit. If the *FLAGS* input is specified, the thumb bit it contains overrides the LSB of the *IP* input.
- *Thread start protocol*.
- *Generic booting protocol*.

The kernel interface page contains additional vectors for making system calls from thumb mode starting at offset 0x110.

~	tSCHEDULE *SC*	tTHREADSWITCH *SC*	*Reserved*	+130
tEXCHANGEREGISTERS *SC*	tLIPC *SC*	tIPC *SC*	*Reserverd*	+120
tPROCESSORCONTROL *pSC*	tTHREADCONTROL *pSC*	tSPACECONTROL *pSC*	tMAPCONTROL *pSC*	+110

B.9 Booting [ARM]

The kernel is provided as an ELF file and must be loaded at the physical load address defined in the ELF header. It must begin execution at the corresponding physically addressed entry point with MMU disabled.

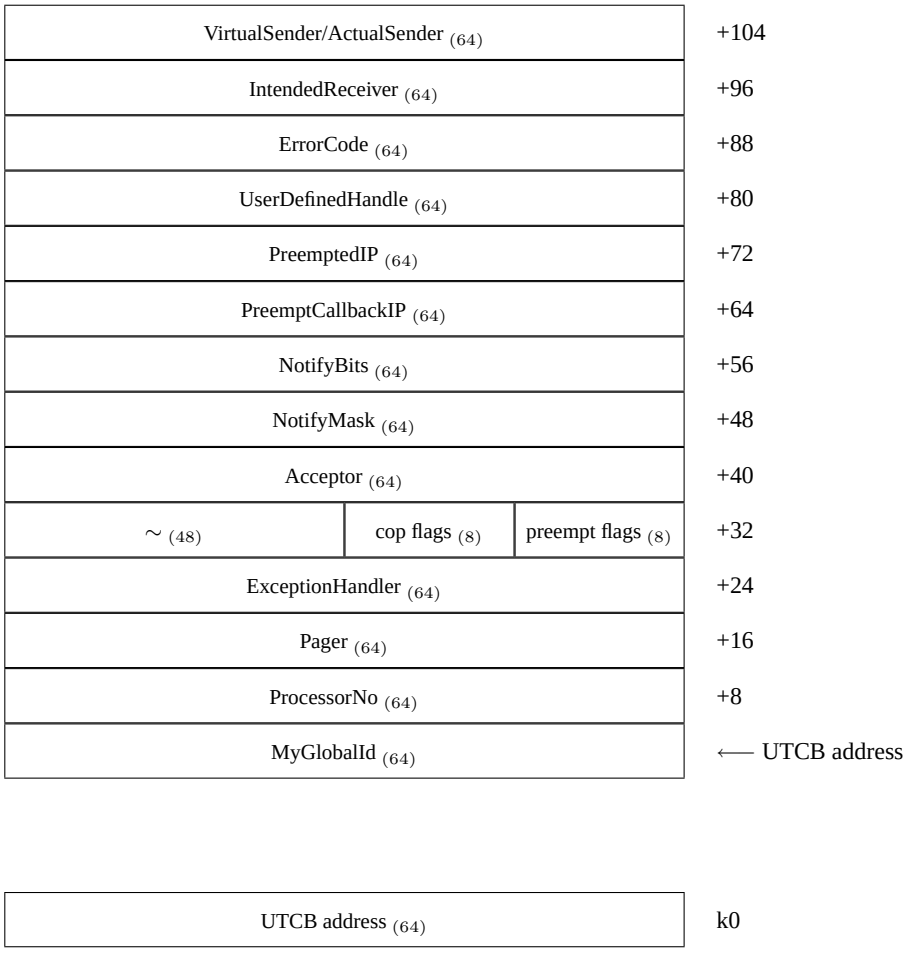
Appendix C

MIPS-64 Interface

C.1 Virtual Registers [MIPS-64]

Thread Control Registers (TCRs)

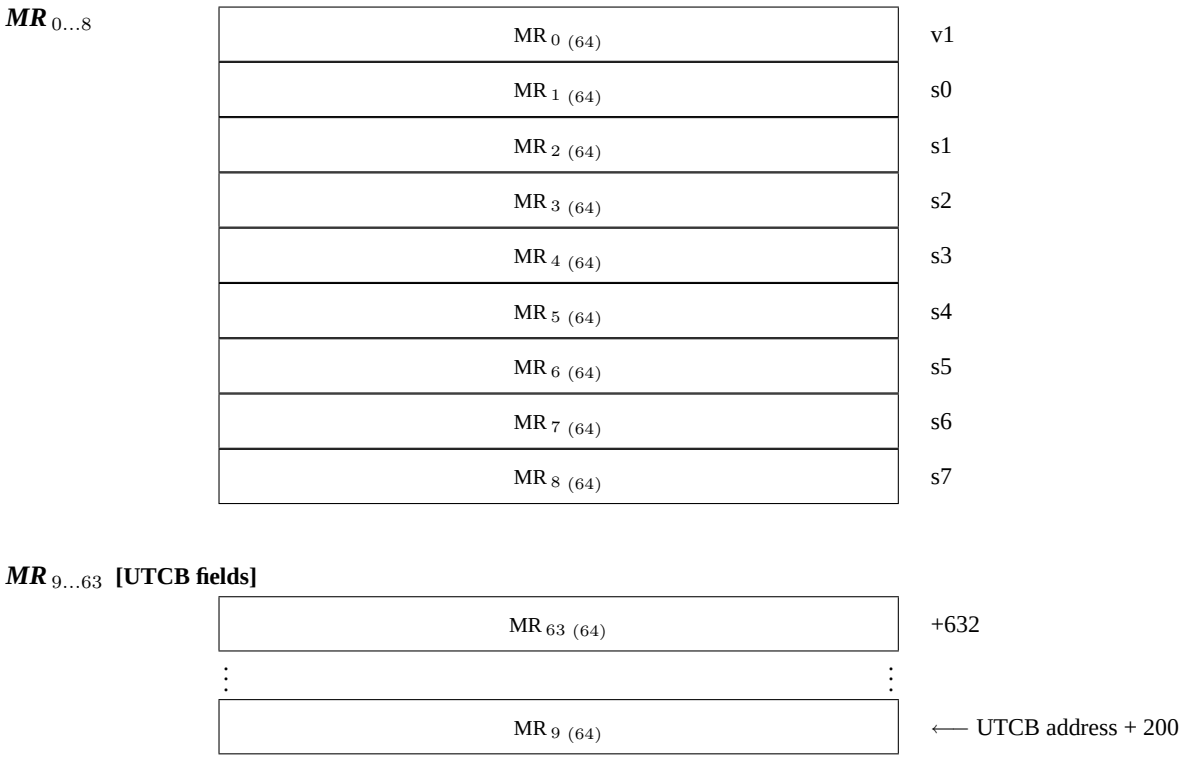
TCRs are mapped to memory locations. They are implemented as part of the mips64-specific user-level thread control block (UTCB). The UTCB is available in the *k0* register. UTCB objects of the current thread can be accessed as any other memory object. UTCBs of other threads must not be accessed, even if they are physically accessible.



Message Registers (MRs)

Message registers MR_0 through MR_8 map to the processor's general purpose register file for IPC and LIPC calls. The remaining message registers map to memory locations in the UTCB. MR_9 starts at byte offset 200 in the UTCB, and successive message registers follow in memory.

The first nine message registers are mapped to the registers v1, s0 to s7. $MR_{9...63}$ are mapped to memory in the UTCB.



UTCB Memory With Undefined Semantics

The kernel will associate no semantics with memory located at *UTCB address* + 128. . . *UTCB address* + 199. The application can use this memory as thread local storage, e.g., for implementing the L4 API. Note, however, that the memory contents within this region may be overwritten during a system-call operating on message registers.

Note, depending on kernel configuration, not all 64 message registers may be available. In this case, no semantics are associated with the memory defined for the unused MRs as above. Note also that when fewer message registers are configured, the kernel may reduce the UTCB size such that memory locations beyond the highest usable message register may not be accessible.

All undefined UTCB memory which is not covered by the above mentioned region may have kernel defined semantics.

C.2 Systemcalls [MIPS-64]

The system-calls invoked via the *jal* instruction are located in the kernel's area of the virtual address space. Their precise locations are stored in the kernel interface page (see page 2). One may invoke the system calls with any instruction that branches to the appropriate target, as long as the return-address register *RA* contains the correct return address.

The locations of the system-calls are fixed during the life of an application, although they may change outside of the life of an application. It is not valid to prelink an application against a set of system call locations. The official locations are always provided in the KIP.

In general, the kernel follows the MIPS ABI64 calling convention for the system call boundary. This means that arguments are passed in the a0...a7 registers (t0...t3 = a4...a7), and the result is placed in the v0 register. All floating point registers are preserved across a system call. All other registers contain return values, are undefined, or may be preserved according to processor specific rules.

KERNELINTERFACE [Slow Systemcall]

0x1FACECA1114E1F64	*at*	− **KernelInterface** →	*at*	≡	
−	*v0,v1*		*v0,v1*	≡	
−	*a0...a3*		*a0...a3*	≡	
−	*a4*	opcode 0x07FFFFFF	*a4*	*KIP base address*	
−	*a5*		*a5*	*API Version*	
−	*a6*		*a6*	*API Flags*	
−	*a7*		*a7*	*Kernel ID*	
−	*t4...t7*		*t4...t7*	≡	
−	*s0...s7*		*s0...s7*	≡	
−	*t8, t9*		*t8, t9*	≡	
−	*gp, sp*		*gp, sp*	≡	
−	*s8*		*s8*	≡	
−	*ra*		*ra*	≡	

For this system-call, all registers other than the output registers are preserved.

EXCHANGEREGISTERS [Systemcall]

−	*at*	− **Exchange Registers** →	*at*	∼	
−	*v0*		*v0*	*result*	
−	*v1*		*v1*	∼	
dest	*a0*	jal *ExchangeRegisters*	*a0*	*control*	
control	*a1*		*a1*	*SP*	
SP	*a2*		*a2*	*IP*	
IP	*a3*		*a3*	*FLAGS*	
FLAGS	*a4*		*a4*	*pager*	
UserDefinedHandle	*a5*		*a5*	*UserDefinedHandle*	
pager	*a6*		*a6*	∼	
−	*a7*		*a7*	∼	
−	*t4...t7*		*t4...t7*	∼	
−	*s0...s7*		*s0...s7*	∼	
−	*t8, t9*		*t8, t9*	∼	
−	*gp*		*gp*	∼	
−	*sp*		*sp*	≡	
−	*s8*		*s8*	≡	
−	*ra*		*ra*	∼	

THREADCONTROL [Privileged Systemcall]

–	*at*	– **Thread Control** →	*at*	∼
–	*v0*		*v0*	*result*
–	*v1*		*v1*	∼
dest	*a0*	jal *ThreadControl*	*a0*	∼
space	*a1*		*a1*	∼
scheduler	*a2*		*a2*	∼
pager	*a3*		*a3*	∼
SendRedirector	*a4*		*a4*	∼
ReceiveRedirector	*a5*		*a5*	∼
UTCB	*a6*		*a6*	∼
–	*a7*		*a7*	∼
–	*t4...t7*		*t4...t7*	∼
–	*s0...s7*		*s0...s7*	∼
–	*t8, t9*		*t8, t9*	∼
–	*gp*		*gp*	∼
–	*sp*		*sp*	≡
–	*s8*		*s8*	≡
–	*ra*		*ra*	∼

THREADSWITCH [Systemcall]

–	*at*	– **ThreadSwitch** →	*at*	∼
–	*v0, v1*		*v0, v1*	∼
dest	*a0*		*a0*	∼
–	*a1...a3*	jal *ThreadSwitch*	*a1...a3*	∼
–	*a4...a7*		*a4...a7*	∼
–	*t4...t7*		*t4...t7*	∼
–	*s0...s7*		*s0...s7*	∼
–	*t8, t9*		*t8, t9*	∼
–	*gp*		*gp*	∼
–	*sp*		*sp*	≡
–	*s8*		*s8*	≡
–	*ra*		*ra*	∼

SCHEDULE [Systemcall]

–	*at*	– **Schedule** →	*at*	∼
–	*v0*		*v0*	*result*
–	*v1*		*v1*	∼
dest	*a0*	jal *Schedule*	*a0*	∼
ts len	*a1*		*a1*	*rem ts*
total quantum	*a2*		*a2*	*rem total*
processor control	*a3*		*a3*	∼
prio	*a4*		*a4*	∼
–	*a5...a7*		*a5...a7*	∼
–	*t4...t7*		*t4...t7*	∼
–	*s0...s7*		*s0...s7*	∼
–	*t8, t9*		*t8, t9*	∼
–	*gp*		*gp*	∼
–	*sp*		*sp*	≡
–	*s8*		*s8*	≡
–	*ra*		*ra*	∼

IPC [Systemcall]

–	*at*			*at*	∼
–	*v0*			*v0*	*from*
MR_0	*v1*			*v1*	MR_0
to	*a0*		jal *Ipc*	*a0*	∼
FromSpecifier	*a1*			*a1*	∼
–	*a2*			*a2*	∼
–	*a3*			*a3*	∼
–	*a4...a7*			*a4...a7*	∼
–	*t4...t7*			*t4...t7*	∼
MR_1	*s0*			*s0*	MR_1
MR_2	*s1*			*s1*	MR_2
MR_3	*s2*			*s2*	MR_3
MR_4	*s3*			*s3*	MR_4
MR_5	*s4*			*s4*	MR_5
MR_6	*s5*			*s5*	MR_6
MR_7	*s6*			*s6*	MR_7
MR_8	*s7*			*s7*	MR_8
–	*t8, t9*			*t8, t9*	∼
–	*gp*			*gp*	∼
–	*sp*			*sp*	≡
–	*s8*			*s8*	≡
–	*ra*			*ra*	∼

– **Ipc** →

LIPC [Systemcall]

–	*at*			*at*	∼
–	*v0*			*v0*	*result*
MR_0	*v1*			*v1*	MR_0
to	*a0*		jal *Lipc*	*a0*	∼
FromSpecifier	*a1*			*a1*	∼
–	*a2*			*a2*	∼
–	*a3*			*a3*	∼
–	*a4...a7*			*a4...a7*	∼
–	*t4...t7*			*t4...t7*	∼
MR_1	*s0*			*s0*	MR_1
MR_2	*s1*			*s1*	MR_2
MR_3	*s2*			*s2*	MR_3
MR_4	*s3*			*s3*	MR_4
MR_5	*s4*			*s4*	MR_5
MR_6	*s5*			*s5*	MR_6
MR_7	*s6*			*s6*	MR_7
MR_8	*s7*			*s7*	MR_8
–	*t8, t9*			*t8, t9*	∼
–	*gp*			*gp*	∼
–	*sp*			*sp*	≡
–	*s8*			*s8*	≡
–	*ra*			*ra*	∼

– **Lipc** →

MAPCONTROL [Systemcall]

–	*at*		– **MapControl** →	*at*	∼
–	*v0*			*v0*	*result*
MR_0	*v1*			*v1*	MR_0
SpaceSpecfier	*a0*		jal *MapControl*	*a0*	∼
control	*a1*			*a1*	∼
–	*a2*			*a2*	∼
–	*a3*			*a3*	∼
–	*a4…a7*			*a4…a7*	∼
–	*t4…t7*			*t4…t7*	∼
MR_1	*s0*			*s0*	MR_1
MR_2	*s1*			*s1*	MR_2
MR_3	*s2*			*s2*	MR_3
MR_4	*s3*			*s3*	MR_4
MR_5	*s4*			*s4*	MR_5
MR_6	*s5*			*s5*	MR_6
MR_7	*s6*			*s6*	MR_7
MR_8	*s7*			*s7*	MR_8
–	*t8, t9*			*t8, t9*	∼
–	*gp*			*gp*	∼
–	*sp*			*sp*	≡
–	*s8*			*s8*	≡
–	*ra*			*ra*	∼

SPACECONTROL [Privileged Systemcall]

–	*at*		– **Space Control** →	*at*	∼
–	*v0*			*v0*	*result*
–	*v1*			*v1*	∼
SpaceSpecifier	*a0*		jal *SpaceControl*	*a0*	*control*
control	*a1*			*a1*	∼
KernelInterfacePageArea	*a2*			*a2*	∼
UtcbArea	*a3*			*a3*	∼
–	*a4*			*a4*	∼
–	*a5…a7*			*a5…a7*	∼
–	*t4…t7*			*t4…t7*	∼
–	*s0…s7*			*s0…s7*	∼
–	*t8, t9*			*t8, t9*	∼
–	*gp*			*gp*	∼
–	*sp*			*sp*	≡
–	*s8*			*s8*	≡
–	*ra*			*ra*	∼

PROCESSORCONTROL [Privileged Systemcall]

–	*at*		– **Processor Control** →	*at*	∼
–	*v0*			*v0*	*result*
–	*v1*			*v1*	∼
processor no	*a0*		jal *ProcessorControl*	*a0*	∼
InternalFreq	*a1*			*a1*	∼
ExternalFreq	*a2*			*a2*	∼
voltage	*a3*			*a3*	∼
–	*a4…a7*			*a4…a7*	∼
–	*t4…t7*			*t4…t7*	∼
–	*s0…s7*			*s0…s7*	∼
–	*t8, t9*			*t8, t9*	∼
–	*gp*			*gp*	∼
–	*sp*			*sp*	≡
–	*s8*			*s8*	≡
–	*ra*			*ra*	∼

C.3 Memory Attributes [MIPS-64]

The MIPS64 architecture implements or emulates all generic attributes see 36. The default attributes depend on the specific MIPS platform.

Before disabling the cache for a page, the software must ensure that all memory belonging to the target page is flushed from the cache.

C.4 Exception Message Format [MIPS-64]

System Call Trap

System Call Trap Message to Exception Handler

a7 $_{(64)}$				MR $_{13}$
a6 $_{(64)}$				MR $_{12}$
a5 $_{(64)}$				MR $_{11}$
a4 $_{(64)}$				MR $_{10}$
a3 $_{(64)}$				MR $_{9}$
a2 $_{(64)}$				MR $_{8}$
a1 $_{(64)}$				MR $_{7}$
a0 $_{(64)}$				MR $_{6}$
v1 $_{(64)}$				MR $_{5}$
v0 $_{(64)}$				MR $_{4}$
Status $_{(64)}$				MR $_{3}$
SP $_{(64)}$				MR $_{2}$
IP $_{(64)}$				MR $_{1}$
-5 $_{(44)}$	0 $_{(4)}$	$t = 0$ $_{(6)}$	$u = 13$ $_{(6)}$	MR $_{0}$

When user code executes the Mips *syscall* instruction, the kernel delivers the system call trap message to the exception handler. The kernel preserves only partial user state when handling a *syscall* instruction. State is preserved similarly for the inclusive set of saved registers according the MIPS ABI 64,n32,o32 for function calls. The *Status* value is described under *Generic Traps*.

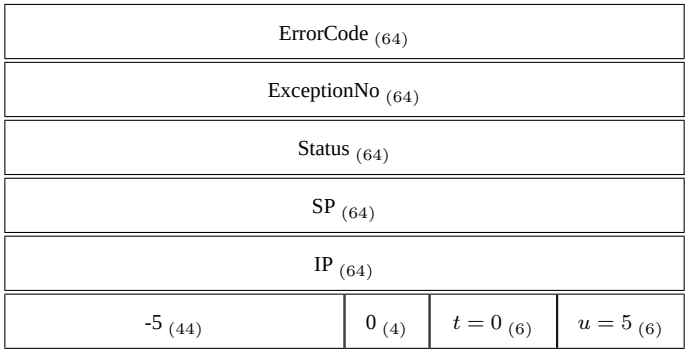
The non-volatile registers are: *s0 . . . s7, gp, sp, fp/s8*

The volatile registers are: *AT, v0, v1, a0 . . . a7, t4 . . . t9, k0, k1, ra, hi, lo*

Thread virtual registers may also be clobbered.

Generic Traps

Generic Trap Message To Exception Handler



The kernel synthesizes exception messages in response to architecture specific events. Some traps are handled by the kernel and therefore do not generate exception messages. The kernel preserves all user state, including thread virtual registers. The *Status* value is encoded as *bits: 31…1 = Flags: 31…1, bit: 0 = Branch*. *Branch* indicates whether the exception took place in a branch delay slot or not.

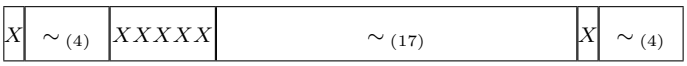
The following is a table of values for the Generic Trap *ExceptionNo*:

Exception	ExceptionNo	ErrorCode	Delivered
Interrupt	0	-	No
TLB Write Denied	1	-	No
TLB Miss Load	2	-	No
TLB Miss Store	3	-	No
Address Error (load/execute)	4	BadVAddress	Yes
Address Error (store)	5	BadVAddress	Yes
Bus Error (instruction)	6	-	Yes
Bus Error (data)	7	-	Yes
System Call	8	-	v0 $\geq$ 0
Break Point	9	-	!(-111 $\geq$ AT $\geq$ -100)
Reserved Instruction	10	Instruction	AT $\neq$ MAGIC_KIP_REQUEST
Coprocessor Unavailable	11	Number	CP0, CP2, CP3
Arithmetic Overflow	12	-	Yes
Trap	13	-	Yes
Virtual Coherency (instruction)	14	-	Yes
Floating Point	15	-	Yes
Watch Point	23	-	Unless used by kdb
Virtual Coherency (data)	31	-	Yes

Note, not all of these exceptions will be delivered via exception IPC. Some will be handled by the kernel. Delivered exceptions are indicated in the last column of the table above.

C.5 Exchange Registers [MIPS-64]

The EXCHANGEREGISTERS system call has an architecture dependent *FLAGS* parameter to specify various user-level CPU flags that can be controlled. For MIPS64, the *FLAGS* parameter has the same fields as the MIPS *status* register. Not all bits in the *status* register are controllable. The following shows which bits are valid.



C.6 Booting [MIPS-64]

The kernel is provided as an ELF file and must be loaded according to the load addresses defined in the ELF header (corresponding to the physical region of the virtual address space). The kernel must be started in 64bit mode.

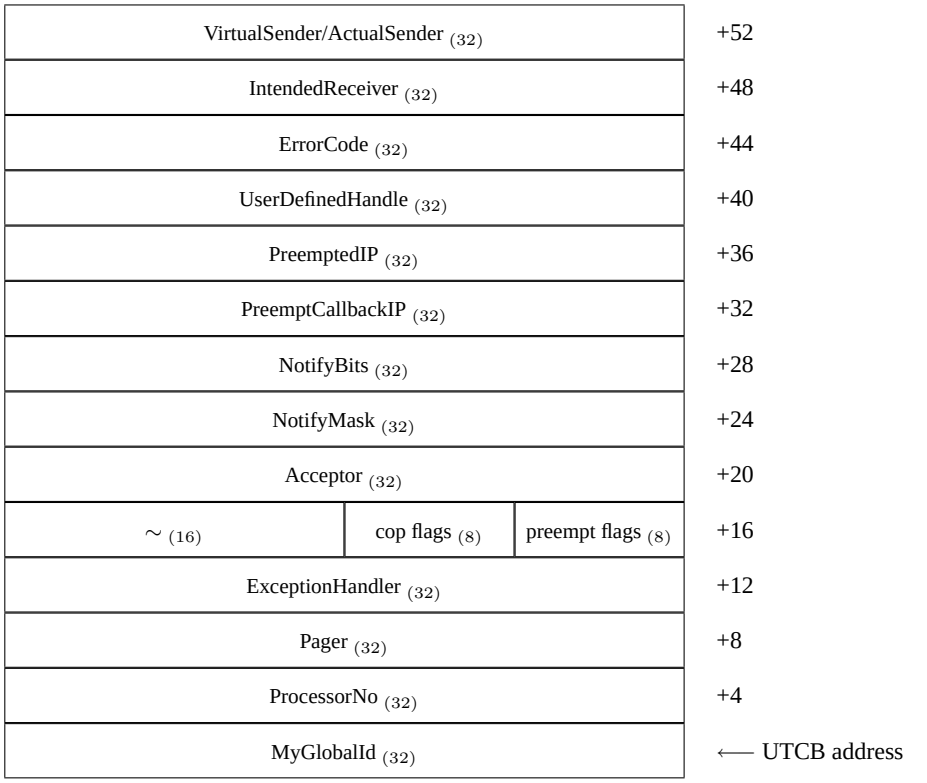
Appendix D

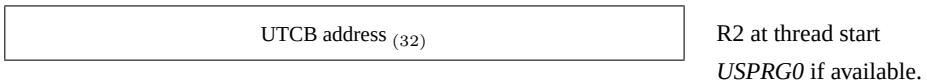
PowerPC Interface

D.1 Virtual Registers [powerpc]

Thread Control Registers (TCRs)

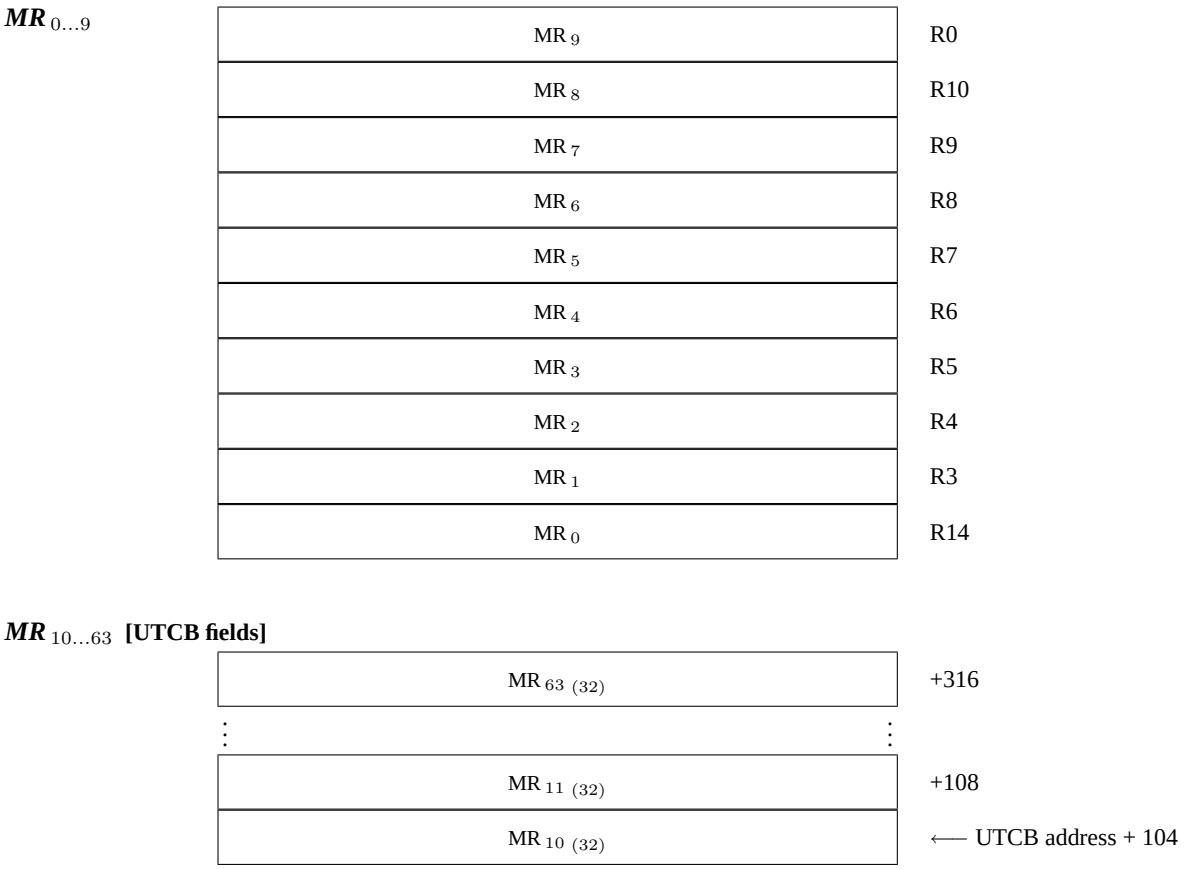
TCRs are mapped to memory locations. They are implemented as part of the PowerPC-specific user-level thread control block (UTCB). The UTCB address is provided in the general purpose register R2 at application start. On certain processors (eg PPC405, PPC440), the UTCB is also available in the *USPRG0* SPR when available. The R2 register is always preserved across system calls, which is useful for keeping the UTCB in R2 on processors without the *USPRG0* SPR. UTCB objects of the current thread can be accessed as any other memory object. UTCBs of other threads must not be accessed, even if they are physically accessible.





Message Registers (MRs)

Message registers MR_0 through MR_9 map to the processor's general purpose register file. The remaining message registers map to memory locations in the UTCB. MR_{10} starts at byte offset 104 in the UTCB, and successive message registers follow in memory.



UTCB Memory With Undefined Semantics

The kernel will associate no semantics with memory located at *UTCB address* + 64. . . *UTCB address* + 107. The application can use this memory as thread local storage, e.g., for implementing the L4 API. Note, however, that the memory contents within this region may be overwritten during a system-call operating on message registers.

Note, depending on kernel configuration, not all 64 message registers may be available. In this case, no semantics are associated with the memory defined for the unused MRs as above. Note also that when fewer message registers are configured, the kernel may reduce the UTCB size such that memory locations beyond the highest usable message register may not be accessible.

All undefined UTCB memory which is not covered by the above mentioned region may have kernel defined semantics.

D.2 Systemcalls [powerpc]

The PowerPC system calls are invoked by changing the location of the instruction pointer to the location of the system call address, with the return address in the link-return (LR) register. The invocation may take place via any mechanism which changes the instruction pointer location. The precise locations of the system calls are stored in the kernel interface page (see page 2).

The locations of the system calls are fixed during the life of an application, although they may change outside of the life of an application. It is not valid to prelink an application against a set of system call locations. The official locations are always provided in the kernel interface page.

The registers defined to survive across system-call invocations (unless otherwise noted) are: R1, R2, R30, R31, and the floating point registers. All other registers contain return values, are undefined, or may be preserved according to processor specific rules.

The R2 register must contain the UTCB pointer when invoking all system calls.

PowerPC uses one alternative system call invocation mechanism, for the KERNELINTERFACE system call. This system call is invoked via the 'tlbia' instruction, and most registers are preserved across the function call.

KERNELINTERFACE [Slow Systemcall]

0x14ca11	*r0*	– **KernelInterface** →	*r0*	≡	
–	*r3*		*r3*	*KIP base address*	
–	*r4*		*r4*	*API Version*	
–	*r5*	tlbia	*r5*	*API Flags*	
–	*r6*		*r6*	*Kernel ID*	
–	*r7*		*r7*	≡	
–	*r8...r29*		*r8...r29*	≡	

For this system-call, all registers other than the output registers are preserved. The tlbia instruction encoding is 0x7c0002e4.

EXCHANGEREGISTERS [Systemcall]

–	*r0*	– **Exchange Registers** →	*r0*	∼	
dest	*r3*		*r3*	*result*	
control	*r4*		*r4*	*control*	
SP	*r5*	jal *ExchangeRegisters*	*r5*	*SP*	
IP	*r6*		*r6*	*IP*	
FLAGS	*r7*		*r7*	*FLAGS*	
UserDefinedHandle	*r8*		*r8*	*UserDefinedHandle*	
pager	*r9*		*r9*	*pager*	
–	*r10*		*r10*	∼	
–	*r11*		*r11*	∼	
–	*r12...r29*		*r12...r29*	∼	

"FLAGS" refers to the user-modifiable PowerPC processor flags that are held in the MSR register. See the PowerPC Processor Mirroring section (page 129).

THREADCONTROL [Privileged Systemcall]

–	*r0*	– **Thread Control** →	*r0*	∼
dest	*r3*		*r3*	*result*
SpaceSpecifier	*r4*		*r4*	∼
Scheduler	*r5*	jal *ThreadControl*	*r5*	∼
Pager	*r6*		*r6*	∼
SendRedirector	*r7*		*r7*	∼
ReceiveRedirector	*r8*		*r8*	∼
UtcbLocation	*r9*		*r9*	∼
–	*r10...r29*		*r10...r29*	∼

THREADSWITCH [Systemcall]

–	*r0*	– **ThreadSwitch** →	*r0*	∼
dest	*r3*		*r3*	∼
–	*r4*		*r4*	∼
–	*r5*	jal *ThreadSwitch*	*r5*	∼
–	*r6*		*r6*	∼
–	*r7*		*r7*	∼
–	*r8*		*r8*	∼
–	*r9*		*r9*	∼
–	*r10...r29*		*r10...r29*	∼

SCHEDULE [Systemcall]

–	*r0*	– **Schedule** →	*r0*	∼
dest	*r3*		*r3*	*result*
ts len	*r4*		*r4*	*rem ts*
total quantum	*r5*	jal *Schedule*	*r5*	*rem total*
processor control	*r6*		*r6*	∼
prio	*r7*		*r7*	∼
–	*r8*		*r8*	∼
–	*r9*		*r9*	∼
–	*r10...r29*		*r10...r29*	∼

IPC [Systemcall]

MR_9	*r0*	– **Ipc** →	*r0*	MR_9
MR_1	*r3*		*r3*	MR_1
MR_2	*r4*		*r4*	MR_2
MR_3	*r5*	jal *Ipc*	*r5*	MR_3
MR_4	*r6*		*r6*	MR_4
MR_5	*r7*		*r7*	MR_5
MR_6	*r8*		*r8*	MR_6
MR_7	*r9*		*r9*	MR_7
MR_8	*r10*		*r10*	MR_8
–	*r11*		*r11*	∼
–	*r12*		*r12*	∼
–	*r13*		*r13*	∼
MR_0	*r14*		*r14*	MR_0
to	*r15*		*r15*	∼
FromSpecifier	*r16*		*r16*	*from*
–	*r17*		*r17*	∼
–	*r18...r29*		*r18...r29*	∼

LIPC [Systemcall]

MR_9	*r0*	– **Lipc** →	*r0*	MR_9
MR_1	*r3*		*r3*	MR_1
MR_2	*r4*		*r4*	MR_2
MR_3	*r5*	jal *Lipc*	*r5*	MR_3
MR_4	*r6*		*r6*	MR_4
MR_5	*r7*		*r7*	MR_5
MR_6	*r8*		*r8*	MR_6
MR_7	*r9*		*r9*	MR_7
MR_8	*r10*		*r10*	MR_8
–	*r11*		*r11*	∼
–	*r12*		*r12*	∼
–	*r13*		*r13*	∼
MR_0	*r14*		*r14*	MR_0
to	*r15*		*r15*	∼
FromSpecifier	*r16*		*r16*	*from*
–	*r17*		*r17*	∼
–	*r18...r29*		*r18...r29*	∼

MAPCONTROL [Systemcall]

MR_9	*r0*	– **MapControl** →	*r0*	MR_9
MR_1	*r3*		*r3*	MR_1
MR_2	*r4*		*r4*	MR_2
MR_3	*r5*	jal *MapControl*	*r5*	MR_3
MR_4	*r6*		*r6*	MR_4
MR_5	*r7*		*r7*	MR_5
MR_6	*r8*		*r8*	MR_6
MR_7	*r9*		*r9*	MR_7
MR_8	*r10*		*r10*	MR_8
–	*r11*		*r11*	∼
–	*r12*		*r12*	∼
–	*r13*		*r13*	∼
MR_0	*r14*		*r14*	MR_0
SpaceSpecifier	*r15*		*r15*	*result*
control	*r16*		*r16*	∼
–	*r17*		*r17*	∼
–	*r18...r29*		*r18...r29*	∼

SPACECONTROL [Privileged Systemcall]

–	*r0*	– **Space Control** →	*r0*	∼
SpaceSpecifier	*r3*		*r3*	*result*
control	*r4*		*r4*	*control*
KernelInterfacePageArea	*r5*	jal *SpaceControl*	*r5*	∼
UtcbArea	*r6*		*r6*	∼
–	*r7*		*r7*	∼
–	*r8*		*r8*	∼
–	*r9...r29*		*r9...r29*	∼

PROCESSORCONTROL [Privileged Systemcall]

–	*r0*	– **Processor Control** →	*r0*	∼
processor no	*r3*		*r3*	*result*
InternalFreq	*r4*		*r4*	∼
ExternalFreq	*r5*	jal *ProcessorControl*	*r5*	∼
voltage	*r6*		*r6*	∼
–	*r7*		*r7*	∼
–	*r8*		*r8*	∼
–	*r9*		*r9*	∼
–	*r10…r29*		*r10…r29*	∼

D.3 Memory Attributes [powerpc]

To be fixed.

The PowerPC architecture places a variety of restrictions on the usage of the memory/cache attributes. Some combinations are meaningless (such as combining write-through with caching-inhibited), or are not permitted and will lead to undefined behavior (for example, instruction fetching is incompatible with some combinations of attributes). The precise semantics of the memory/cache access attributes are described in the "Programming Environments Manual For 32-Bit Implementations of the PowerPC Architecture."

Before disabling the cache for a page, the software must ensure that all memory belonging to the target page is flushed from the cache.

Generic Programming Interface

#include <l4/misc.h>

Word ***DefaultMemory***

Word ***WriteThroughMemory***

Word ***WriteBackMemory***

Word ***CachingInhibitedMemory***

Word ***CachingEnabledMemory***

Word ***GlobalMemory***

Word ***LocalMemory***

Word ***GuardedMemory***

Word ***SpeculativeMemory***

D.4 Exception Message Format [powerpc]

System Call Trap

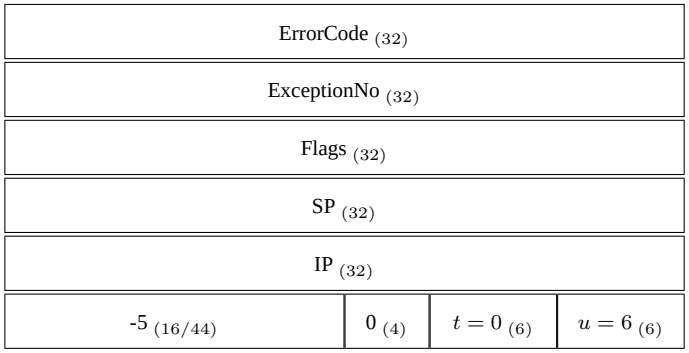
System Call Trap Message to Exception Handler

Flags $_{(32)}$				MR $_{12}$
SP $_{(32)}$				MR $_{11}$
IP $_{(32)}$				MR $_{10}$
r0 $_{(32)}$				MR $_{9}$
r10 $_{(32)}$				MR $_{8}$
r9 $_{(32)}$				MR $_{7}$
r8 $_{(32)}$				MR $_{6}$
r7 $_{(32)}$				MR $_{5}$
r6 $_{(32)}$				MR $_{4}$
r5 $_{(32)}$				MR $_{3}$
r4 $_{(32)}$				MR $_{2}$
r3 $_{(32)}$				MR $_{1}$
-5 $_{(16/48)}$	0 $_{(4)}$	$t = 0$ $_{(6)}$	$u = 12$ $_{(6)}$	MR $_{0}$

When user code executes the PowerPC 'sc' instruction, the kernel delivers the system call trap message to the exception handler. The kernel preserves only partial user state when handling an 'sc' instruction. State is preserved similarly to the SVR4 PowerPC ABI for function calls. The non-volatile registers are R1, R2, R13...R31, CR2, CR3, CR4, LR, and FPSCR. The volatile registers are R0, R3...R12, CR0, CR1, CR5...CR7, CTR, and XER. Thread virtual registers may also be clobbered.

Generic Traps

Generic Trap Message To Exception Handler



The kernel synthesizes exception messages in response to architecture specific events. Some traps are handled by the kernel and therefore do not generate exception messages. The kernel preserves all user state, including thread virtual registers.

D.5 Processor Mirroring [powerpc]

The kernel will expose the following supervisor instructions to all user level programs via emulation: MFSPR for the PVR, and other cpu-specific version infomation.

The EXCHANGEREGISTERS system-call accesses the flags of the processor. The flags map directly to the PowerPC MSR register. The following bits may be read and modified by user applications: LE, BE, SE, FE0, and FE1 if supported by the cpu. The kernel may also expose additional cpu-specific bits.

D.6 Booting [powerpc]

Apple New World Compatible Machines

L4 must be loaded into memory at the physical location defined by the kernel's ELF header. It can be started with virtual addressing enabled or disabled. Execution of L4 must begin at the entry point defined by the kernel's ELF header.

When entering the kernel, the registers which support in-register file parameter passing, R3–R10 according to the SVR4 ABI, must be cleared for upwards compatibility, except as noted below. All other registers in the register file are undefined at kernel entry.

The kernel may use OpenFirmware for debug console I/O. To support OpenFirmware I/O, the OpenFirmware virtual mode client call-back address must be passed to the kernel in register R5, and OpenFirmware must be prepared to handle client call-backs using virtual addressing. In all other cases, register R5 must be zero.

The boot loader must copy the OpenFirmware device tree to memory, and record its physical location in a memory descriptor of the kernel interface page. The copy of the device tree must include the package handles of the device tree nodes

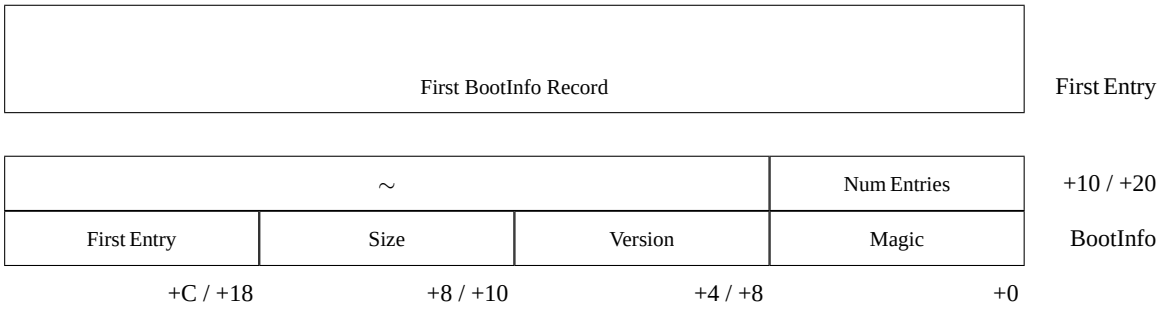
Appendix E

Generic BootInfo

E.1 Generic BootInfo [Data Structure]

The generic BootInfo structure contains boot loader specific data such as loaded modules or files, location of system tables, etc. The data structure can be located anywhere in memory, but must be aligned at a word size.

The BootInfo structure is a pure boot loader specific object. That is, the kernel does not associate any semantics with its contents. A boot loader is free to choose whether to provide a BootInfo structure or not. Starting a system without a generic BootInfo structure is perfectly valid.



The base address of the bootinfo structure is specified by the Bootinfo field in the kernel interface page (see page 4). Note that the base address as specified by the BootInfo field is a physical address. An application running on virtual memory must determine the location of the BootInfo structure within its own address space by other means.

BootInfo Description

Magic The magic number 0x14B0021D. The magic also determines the endianess of the structure (i.e., the value 0x1D02B014 indicates that the endian is wrong). The word size of the BootInfo structure is defined by the word size specified in the kernel interface page (see page 3).

Version API version of the BootInfo structure. This document describes version 1. Note that any changes in the BootInfo records themselves do not influence the version in the main BootInfo structure. This enables BootInfo records to be added or modified without introducing major incompatibilities with a program that parses the BootInfo structure. Only the added/modified BootInfo record types are influenced by the update.

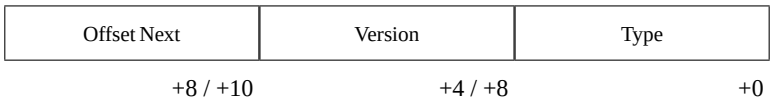
Size The size (in bytes) of the complete BootInfo structure, including all BootInfo records and data referenced by these records.

First Entry Points to the first BootInfo record. *First Entry* is given as an address relative to the base address of the BootInfo structure itself.

Num Entries Number of BootInfo records in the BootInfo structure.

Generic BootInfo Record

The exact structure of a BootInfo record is determined by the type of the record. Only the three first words of the record are defined for all BootInfo record types.



Type Specifies the type of the BootInfo record.

Version Specifies the API version of the BootInfo record type. Increasing the version of a BootInfo record type does not also require an increase in the main BootInfo version. Later versions of a BootInfo record are guaranteed to be backwards compatible with older versions.

Offset Next The offset (in bytes) to the next BootInfo record. Note that the offset may vary from record to record, even for records of the same type. This enables the boot loader to have variable length records, place data in between records, or otherwise align records for ease of implementation. It is wrong to assume that the offset associated with a particular version of a record type is constant.

Convenience Programming Interface

#include <l4/bootinfo.h>

struct **BOOTREC** { Word raw [*] }

Bool ***BootInfo_Valid*** (*void* BootInfo*)
Checks whether specified BootInfo structure is valid or not (i.e., whether the magic number and the version number are correct).

Word ***BootInfo_Size*** (*void* BootInfo*)
Delivers the size (in bytes) of the BootInfo structure. It is assumed that *BootInfo* specifies a valid BootInfo structure.

*BootRec** ***BootInfo_FirstEntry*** (*void* BootInfo*)
Delivers the first BootInfo record of the BootInfo structure. It is assumed that *BootInfo* specifies a valid BootInfo structure.

Word ***BootInfo_Entries*** (*void* BootInfo*)
Delivers the number of BootInfo records in the BootInfo structure. It is assumed that *BootInfo* specifies a valid BootInfo structure.

Word ***Type*** (*BootRec* BootRec*) [*BootRec_Type*]
Delivers the type of the BootInfo record.

*BootRec** ***Next*** (*BootRec* BootRec*) [*BootRec_Next*]
Delivers the next BootInfo record. The value returned by the last BootInfo record in the BootInfo structure is undefined.

E.2 BootInfo Records [BootInfo]

BootInfo records can be listed in any order. This section lists currently defined BootInfo records. A program encountering an unknown BootInfo record can skip past the record using the ubiquitous *Offset Next* field.

Simple Module The *Simple Module* BootInfo record specifies a binary file loaded into main memory by the boot loader.

		Cmdline Off	Size	+10 / +20
Start	Offset Next	version = 1	type = 0x1	
+C / +18	+8 / +10	+4 / +8	+0	

Start Physical address of first byte in loaded module.

Size Size of loaded module (in bytes).

Cmdline Off Address of command line associated with loaded module, or 0 if no command line exists. Address is specified relative to base address of current BootInfo record.

Simple Executable The *Simple Executable* BootInfo record specifies an executable file which has been loaded into main memory and relocated by the boot loader. The record can only specify simple executables with single code, data, and bss sections.

Cmdline Off	Label	Flags	Initial IP	+30 / +60
Bss.Size	Bss.Vstart	Bss.Pstart	Data.Size	+20 / +40
Data.Vstart	Data.Pstart	Text.Size	Text.Vstart	+10 / +20
Text.Pstart	Offset Next	version = 1	type = 0x2	
+C / +18	+8 / +10	+4 / +8	+0	

Pstart Physical address of first byte in code/data/bss section of the loaded executable.

Vstart Virtual address of first byte in code/data/bss section of the loaded executable.

Size Size of code/data/bss section (in bytes).

Initial IP Virtual address of entry point for loaded executable.

Flags Flags for the loaded executable (defined by boot loader or application programs). Note that regular applications may not necessarily have write permissions on the *Flags* field.

Label Freely available word (defined by boot loader or application programs). Note that regular applications may not necessarily have write permissions on the *Label* field.

Cmdline Off Address of command line associated with loaded executable, or 0 if no command line exists. Address is specified relative to base address of current BootInfo record.

EFI Tables

The *EFI Tables* BootInfo record specifies the location and size of the EFI memory map, and the location of the EFI system table.

Memdesc Version	Memdesc Size	Memmap Size	Memmap	+10 / +20
Systab	Offset Next	version = 1	type = 0x101	
+C / +18	+8 / +10	+4 / +8	+0	

Systab Physical address of EFI system table, or 0 if EFI system table is not present.

Memmap Physical address of EFI memory map. Undefined if *Memmap Size* = 0.

Memmap Size Size (in bytes) of the EFI memory map, or 0 if EFI memory map is not present.

Memdesc Size Size (in bytes) of descriptor entries in the EFI memory map. Undefined if *Memmap Size* = 0.

Memdesc Version Version of descriptor entries in the EFI memory map. Undefined if *Memmap Size* = 0.

Multiboot info

The *Multiboot info* BootInfo record specifies the location of the first byte in the multiboot header.

Multiboot Addr	Offset Next	version = 1	type = 0x102
+C / +18	+8 / +10	+4 / +8	+0

Multiboot Addr Physical address of first byte in multiboot header.

Convenience Programming Interface

#include <l4/bootinfo.h>

Word ***BootInfo_Module***

Word ***BootInfo_SimpleExec***

Word ***BootInfo_EFITables***

Word ***BootInfo_Multiboot***

Word ***Module_Start*** (*BootRec* b*)

Word ***Module_Size*** (*BootRec* b*)
Delivers the start and size of the specified boot module.

*char** ***Module_Cmdline*** (*BootRec* b*)
Delivers the command line of the specified boot module, or 0 if command line does not exist.

Word ***SimpleExec_TextPstart*** (*BootRec* b*)

Word ***SimpleExec_TextVstart*** (*BootRec* b*)

Word ***SimpleExec_TextSize*** (*BootRec* b*)

Word ***SimpleExec_DataPstart*** (*BootRec* b*)

Word ***SimpleExec_DataVstart*** (*BootRec* b*)

Word ***SimpleExec_DataSize*** (*BootRec* b*)

Word ***SimpleExec_BssPstart*** (*BootRec* b*)

Word ***SimpleExec_BssVstart*** (*BootRec* b*)

Word ***SimpleExec_BssSize*** (*BootRec* b*)
Delivers physical start address, virtual start address, and size of the code/data/bss section of the specified executable.

Word ***SimpleExec_InitialIP*** (*BootRec* b*)
Delivers virtual address of entry point for the specified executable.

Word ***SimpleExec_Flags*** (*BootRec* b*)

void ***SimpleExec_Set_Flags*** (*BootRec* b, Word w*)
Delivers/sets the flags field for the specified executable.

Word ***SimpleExec_Label*** (*BootRec* b*)

void ***SimpleExec_Set_Label*** (*BootRec* b, Word w*)
Delivers/sets the label field for the specified executable.

*char** ***SimpleExec_Cmdline*** (*BootRec* b*)
Delivers the command line of the specified executable, or 0 if command line does not exist.

Word ***EFI_Systab*** (*BootRec* b*)
Delivers the EFI system table, or 0 if system table not present.

Word ***EFI_Memmap*** (*BootRec* b*)

Word ***EFI_MemmapSize*** (*BootRec* b*)

Word ***EFI_MemdescSize*** (*BootRec* b*)

Word ***EFI_MemdescVersion*** (*BootRec* b*)
Delivers location of the EFI memory map, size of memory map, size of memory map descriptor entries, and version of memory map descriptor entries. If *EFI_MemmapSize ()* delivers 0, the other return values are undefined.

Word ***MBI_Address*** (*BootRec* b*)
Delivers the physical location of the first byte in the multiboot header.

Appendix F

Development Remarks

These remarks illuminate the design process from version 2 to version 4.

F.1 Exception Handling

The current model decided upon for exception handling in L4 is to associate an exception handler thread with each thread in the system (see page 60). This model was chosen because it allowed us to handle exceptions generically without introducing any new concepts into the API. It also closely resembles the current page fault handling model.

Another model for exception handling is to use callbacks. Using this model an instruction pointer for a callback function and a pointer to an exception state save area is associated with each thread. Upon catching an exception the kernel stores the cause of the exception into the save area and transfers execution to the exception callback function.

It is evident that the callback model can be faster than the IPC model because the callback model may require only one control transfer into the kernel whereas the IPC model will require at least two. Nevertheless, the IPC model was chosen because it introduces no new mechanisms into the kernel, and we are currently not aware of any real life scenario where the extra performance gain you very much. There exists a challenge to prove these claims wrong. See `http://l4hq.org/fun/` for the rules of the challenge.

Table of Procs, Types, and Constants

	used system call	page
AbortIpc_and_stop (ThreadId t) ThreadState	EXCHANGEREGISTERS	22
AbortIpc_and_stop (ThreadId t, Word& sp, ip, flags) ThreadState	EXCHANGEREGISTERS	22
AbortReceive_and_stop (ThreadId t) ThreadState	EXCHANGEREGISTERS	22
AbortReceive_and_stop (ThreadId t, Word& sp, ip, flags) ThreadState	EXCHANGEREGISTERS	22
AbortSend_and_stop (ThreadId t) ThreadState	EXCHANGEREGISTERS	22
AbortSend_and_stop (ThreadId t, Word& sp, ip, flags) ThreadState	EXCHANGEREGISTERS	22
Accept (Acceptor a) void	*–none–*	49
Accepted () Acceptor	*–none–*	49
Acceptor data type	*–n/a–*	49
− (Acceptor l, r) Acceptor	*–none–*	49
+ (Acceptor l, r) Acceptor	*–none–*	49
ActualSender () ThreadId	*–none–*	18
ActualSender () ThreadId	*–none–*	56
Address (Fpage f) Word	*–none–*	35
anylocalthread ThreadId const	*–n/a–*	15
anythread ThreadId const	*–n/a–*	15
ApiFlags () Word	*–none–*	8
ApiVersion () Word	*–none–*	8
Append (Msg& msg, Word w) void	*–none–*	48
ArchitectureSpecificMemoryType Word const	*–n/a–*	9
AssociateInterrupt (ThreadId InterruptThread, InterruptHandler) Word	*–none–*	26
BootInfo_EFITables Word const	*–n/a–*	135
BootInfo_Entries (void* BootInfo) Word	*–none–*	133
BootInfo_FirstEntry (void* BootInfo) BootRec*	*–none–*	133
BootInfo_Module Word const	*–n/a–*	135
BootInfo_Multiboot Word const	*–n/a–*	135
BootInfo_SimpleExec Word const	*–n/a–*	135
BootInfo_Size (void* BootInfo) Word	*–none–*	133
BootInfo_Valid (void* BootInfo) Bool	*–none–*	133
BootInfo (void* KernelInterface) Word	*–none–*	9
BootLoaderSpecificMemoryType Word const	*–n/a–*	9
BootRec data type	*–n/a–*	133
CacheControl (ThreadId SpaceSpecifier, Word control) Word	*–none–*	65
CachedMemory Word const	*–n/a–*	36
CacheFlushAll () Word	CACHECONTROL	66
CacheFlushDRange (ThreadId s, Word start, end) Word	CACHECONTROL	66
CacheFlushIRange (ThreadId s, Word start, end) Word	CACHECONTROL	66
CacheFlushRangeInvalidate (ThreadId s, Word start, end) Word	CACHECONTROL	66
CacheFlushRange (ThreadId s, Word start, end) Word	CACHECONTROL	66
CachingEnabledMemory Word const	*–n/a–*	126
CachingInhibitedMemory Word const	*–n/a–*	126
Call (ThreadId to) MsgTag	IPC	55
Clear (Msg& msg) void	*–none–*	48
Clear_ReceiveBlock (MsgTag& t) void	*–none–*	56
Clear_SendBlock (MsgTag& t) void	*–none–*	57
Clr_CopFlag (Word n) void	*–none–*	18
Clr_CopFlag (Word n) void	*–none–*	61
CoherentMemory Word const	*–n/a–*	36
CompleteAddressSpace Fpage const	*–n/a–*	35

	used system call	page
ConventionalMemoryType Word const	*–n/a–*	9
Copy_regs (ThreadId src, ThreadId dest) void	EXCHANGEREGISTERS	22
Copy_regs (ThreadId src, ThreadId dest, Word sp, ip) void	EXCHANGEREGISTERS	22
DeassociateInterrupt (ThreadId InterruptThread) Word	*–none–*	26
DedicatedMemoryType Word const	*–n/a–*	9
DefaultMemory Word const	*–n/a–*	126
DefaultMemory Word const	*–n/a–*	36
DefaultMemory Word const	*–n/a–*	88
DisablePreemptionCallback () Bool	*–none–*	32
EFI_MemdescSize (BootRec* b) Word	*–none–*	136
EFI_MemdescVersion (BootRec* b) Word	*–none–*	136
EFI_Memmap (BootRec* b) Word	*–none–*	136
EFI_MemmapSize (BootRec* b) Word	*–none–*	136
EFI_Systab (BootRec* b) Word	*–none–*	136
EnablePreemptionCallback () Bool	*–none–*	32
ErrInvalidParam Word const	*–n/a–*	31
ErrInvalidParam Word const	*–n/a–*	66
ErrInvalidRedirector Word const	*–n/a–*	26
ErrInvalidScheduler Word const	*–n/a–*	26
ErrInvalidSpace Word const	*–n/a–*	26
ErrInvalidSpace Word const	*–n/a–*	43
ErrInvalidThread Word const	*–n/a–*	22
ErrInvalidThread Word const	*–n/a–*	26
ErrInvalidThread Word const	*–n/a–*	31
ErrKipArea Word const	*–n/a–*	43
ErrNoMem Word const	*–n/a–*	26
ErrNoPrivilege Word const	*–n/a–*	26
ErrNoPrivilege Word const	*–n/a–*	31
ErrNoPrivilege Word const	*–n/a–*	43
ErrNoPrivilege Word const	*–n/a–*	63
ErrNoPrivilege Word const	*–n/a–*	66
ErrorCode () Word	*–none–*	18
ErrorCode () Word	*–none–*	22
ErrorCode () Word	*–none–*	26
ErrorCode () Word	*–none–*	31
ErrorCode () Word	*–none–*	43
ErrorCode () Word	*–none–*	56
ErrorCode () Word	*–none–*	63
ErrorCode () Word	*–none–*	66
ErrUtcbArea Word const	*–n/a–*	26
ErrUtcbArea Word const	*–n/a–*	43
ExceptionHandler () ThreadId	*–none–*	18
ExceptionHandler () ThreadId	*–none–*	60
ExchangeRegisters (ThreadId dest, Word control, sp, ip, flags, UserDefinedHandle, ThreadId pager, Word& old_control, old_sp, old_ip, old_flags, old_UserDefinedHandle, ThreadId& old_pager) ThreadId	EXCHANGEREGISTERS	21
eXecutable Word const	*–n/a–*	35
ExternalFreq (ProcDesc& p) Word	*–none–*	10
Feature (void* KernelInterface, Word num) char*	*–none–*	9
Fpage data type	*–n/a–*	35
− (Fpage f, Word AccessRights) Fpage	*–none–*	35
+ (Fpage f, Word AccessRights) Fpage	*–none–*	35
FpageLog2 (Word BaseAddress, int Log2FpageSize $<$ 64) Fpage	*–none–*	35
Fpage (Word BaseAddress, Word FpageSize $\geq$ 1K) Fpage	*–none–*	35
FullyAccessible Word const	*–n/a–*	35
Get (Msg& msg, Word& ut) void	*–none–*	47
Get (Msg& msg, Word u) Word	*–none–*	48
Get (Msg& msg, Word u, Word& w) void	*–none–*	48
Get_NotifyBits () Word	*–none–*	50
Get_NotifyMask () Word	*–none–*	50
GlobalId (Word threadno, version) ThreadId	*–none–*	15
GlobalMemoryType Word const	*–n/a–*	9
GlobalMemory Word const	*–n/a–*	126

	used system call	page
GuardedMemory Word const	*–n/a–*	126
High (MemoryDesc& m) Word	*–none–*	10
IntendedReceiver () ThreadId	*–none–*	18
IntendedReceiver () ThreadId	*–none–*	56
InternalFreq (ProcDesc& p) Word	*–none–*	10
IOCombinedMemory Word const	*–n/a–*	36
IoFpageLog2 (Word BaseAddress, int Log2FpageSize $<$ 64) Fpage	*–none–*	86
IoFpage (Word BaseAddress, int FpageSize) Fpage	*–none–*	86
IOMemory Word const	*–n/a–*	36
IpcFailed (MsgTag t) Bool	*–none–*	56
IpcPropagated (MsgTag t) Bool	*–none–*	56
IpcRedirected (MsgTag t) Bool	*–none–*	56
IpcSucceeded (MsgTag t) Bool	*–none–*	56
Ipc (ThreadId to, FromSpecifier, MsgTag tag, ThreadId& from) MsgTag	IPC	55
IpcXcpu (MsgTag t) Bool	*–none–*	56
IsNilFpage (Fpage f) Bool	*–none–*	35
IsNilThread (ThreadId t) Bool	*–none–*	15
IsVirtual (MemoryDesc& m) Bool	*–none–*	9
KernelGenDate (void* KernelInterface, Word& year, month, day) void	*–none–*	8
KernelId () Word	*–none–*	8
KernelInterface () void*	KERNELINTERFACE	8
KernelInterface (Word& ApiVersion, ApiFlags, KernelId) void *	KERNELINTERFACE	8
KernelSupplier (void* KernelInterface) Word	*–none–*	8
KernelVersionString (void* KernelInterface) char*	*–none–*	9
KernelVersion (void* KernelInterface) Word	*–none–*	8
KipAreaSizeLog2 (void* KernelInterface) Word	*–none–*	9
Label (Msg& msg) Word	*–none–*	48
Label (Msg Tag t) Word	*–none–*	47
LargeSpace Word const	*–n/a–*	87
Lcall (ThreadId to) MsgTag	LIPC	56
Lipc (ThreadId to, FromSpecifier, MsgTag tag, ThreadId& from) MsgTag	LIPC	55
LoadMR (int i, Word w) void	*–none–*	11
LoadMR (int i, Word w) void	*–none–*	48
LoadMRs (int i, k, Word& [k] w) void	*–none–*	11
LoadMRs (int i, k, Word& [k] w) void	*–none–*	48
Load (Msg& msg) void	*–none–*	48
LocalMemory Word const	*–n/a–*	126
Low (MemoryDesc& m) Word	*–none–*	9
LreplyWait (ThreadId to, ThreadId& from) MsgTag	LIPC	56
MapControl (ThreadId SpaceSpecifier, Word control) Word	MAPCONTROL	40
Map (ThreadId s, Fpage f, PhysDesc p) Fpage	MAPCONTROL	40
Map (ThreadId s, Word n, Fpage& [n] fpages PhysDesc& [n] p) Fpage	MAPCONTROL	40
MBI_Address (BootRec* b) Word	*–none–*	136
MemoryDesc data type	*–n/a–*	8
MemoryDesc (void* KernelInterface, Word num) MemoryDesc*	*–none–*	9
MessageRegisters (void* KernelInterface) Word	*–none–*	9
Module_Cmdline (BootRec* b) char*	*–none–*	135
Module_Size (BootRec* b) Word	*–none–*	135
Module_Start (BootRec* b) Word	*–none–*	135
Msg data type	*–n/a–*	47
MsgTag data type	*–n/a–*	47
$==$ (MsgTag l, r) Bool	*–none–*	47
MsgTag (Msg& msg) MsgTag	*–none–*	47
MsgTag () MsgTag	*–none–*	47
$+$ (MsgTag t, Word label) MsgTag	*–none–*	47
$-=$ (Acceptor l, r) Acceptor	*–none–*	49
$+=$ (Acceptor l, r) Acceptor	*–none–*	49
$-=$ (Fpage f, Word AccessRights) Fpage	*–none–*	35
$+=$ (Fpage f, Word AccessRights) Fpage	*–none–*	35
$!=$ (MsgTag l, r) Bool	*–none–*	47
$+=$ (MsgTag t, Word label) MsgTag	*–none–*	47
$!=$ (ThreadId l, r) Bool	*–none–*	15
MyGlobalId () ThreadId	*–none–*	15

	used system call	page
MyGlobalId () ThreadId	*–none–*	17
Myself () ThreadId	*–none–*	15
Myself () ThreadId	*–none–*	17
Next (BootRec* BootRec) BootRec*	*–none–*	133
Nilpage Fpage const	*–n/a–*	35
Niltag MsgTag const	*–n/a–*	47
nilthread ThreadId const	*–n/a–*	15
NoAccess Word const	*–n/a–*	35
NotifyBits () Word	*–none–*	18
NotifyMask () Word	*–none–*	18
NotifyMsgAcceptor Acceptor const	*–n/a–*	49
Notify (ThreadId to, Word& mask) MsgTag	LIPC	56
NUMAMemoryType Word const	*–n/a–*	9
NumMemoryDescriptors (void* KernelInterface) Word	*–none–*	8
NumProcessors (void* KernelInterface) Word	*–none–*	8
PageRights (void* KernelInterface) Word	*–none–*	8
Pager () ThreadId	*–none–*	18
Pager (ThreadId t) ThreadId	EXCHANGEREGISTERS	21
PageSizeMask (void* KernelInterface) Word	*–none–*	8
PhysDesc (Word PhysAddress, Word Attribute) PhysDesc	*–none–*	36
PreemptedIP () Word	*–none–*	18
PreemptedIP () Word	*–none–*	32
ProcDesc data type	*–n/a–*	8
ProcDesc (void* KernelInterface, Word num) ProcDesc*	*–none–*	9
ProcessorControl (Word ProcessorNo, InternalFrequency, ExternalFrequency, voltage) Word	*–none–*	63
ProcessorNo () Word	*–none–*	18
Put (Msg& msg, Word l, int u, Word& [u] ut) void	*–none–*	47
Put (Msg& msg, Word u, Word w) void	*–none–*	48
Readable Word const	*–n/a–*	35
ReadeXecOnly Word const	*–n/a–*	35
Receive (ThreadId from) MsgTag	IPC	56
Reply (ThreadId to) MsgTag	IPC	56
ReplyWait (ThreadId to, ThreadId& from) MsgTag	IPC	56
ReservedMemoryType Word const	*–n/a–*	9
Rights (Fpage f) Word	*–none–*	35
SchedulePrecision (void* KernelInterface) Word	*–none–*	9
Schedule (ThreadId dest, ProcessorControl, prio, PreemptionControl) Word	SCHEDULE	31
Send (ThreadId to) MsgTag	IPC	55
Set_CopFlag (Word n) void	*–none–*	18
Set_CopFlag (Word n) void	*–none–*	61
Set_ExceptionHandler (ThreadId NewHandler) void	*–none–*	18
Set_ExceptionHandler (ThreadId new) void	*–none–*	60
Set_Label (Msg& msg, Word label) void	*–none–*	48
Set_MsgTag (Msg& msg, MsgTag t) void	*–none–*	47
Set_MsgTag (MsgTag t) void	*–none–*	47
Set_NotifyBits (Word bits) void	*–none–*	18
Set_NotifyBits (Word bits) void	*–none–*	50
Set_NotifyMask (Word mask) void	*–none–*	18
Set_NotifyMask (Word mask) void	*–none–*	50
Set_Notify (MsgTag& t) void	*–none–*	56
Set_Pager (ThreadId NewPager) void	*–none–*	18
Set_Pager (ThreadId t, p) void	EXCHANGEREGISTERS	21
Set_PreemptCallbackIP (Word ip) void	*–none–*	18
Set_PreemptCallbackIP (Word ip) void	*–none–*	32
Set_Priority (ThreadId dest, Word prio) Word	*–none–*	31
Set_ProcessorNo (ThreadId dest, Word ProcessorNo) Word	*–none–*	31
Set_Propagation (MsgTag& t) void	*–none–*	56
Set_ReceiveBlock (MsgTag& t) void	*–none–*	56
Set_ReceiveRedirector (ThreadId Thread, ThreadId Redirector) void	*–none–*	26
Set_Rights (Fpage& f, Word AccessRights) void	*–none–*	35
Set_SendBlock (MsgTag& t) void	*–none–*	57
Set_SendRedirector (ThreadId Thread, ThreadId Redirector) void	*–none–*	26

	used system call	page
Set_Timeslice (ThreadId dest, Word ts, Word tq) Word	*–none–*	31
Set_UserDefinedHandle (ThreadId t, Word handle) void	EXCHANGEREGISTERS	21
Set_UserDefinedHandle (Word NewValue) void	*–none–*	18
Set_VirtualSender (ThreadId t) void	*–none–*	18
Set_VirtualSender (ThreadId t) void	*–none–*	57
SimpleExec_BssPstart (BootRec* b) Word	*–none–*	135
SimpleExec_BssSize (BootRec* b) Word	*–none–*	136
SimpleExec_BssVstart (BootRec* b) Word	*–none–*	135
SimpleExec_Cmdline (BootRec* b) char*	*–none–*	136
SimpleExec_DataPstart (BootRec* b) Word	*–none–*	135
SimpleExec_DataSize (BootRec* b) Word	*–none–*	135
SimpleExec_DataVstart (BootRec* b) Word	*–none–*	135
SimpleExec_Flags (BootRec* b) Word	*–none–*	136
SimpleExec_InitialIP (BootRec* b) Word	*–none–*	136
SimpleExec_Label (BootRec* b) Word	*–none–*	136
SimpleExec_Set_Flags (BootRec* b, Word w) void	*–none–*	136
SimpleExec_Set_Label (BootRec* b, Word w) void	*–none–*	136
SimpleExec_TextPstart (BootRec* b) Word	*–none–*	135
SimpleExec_TextSize (BootRec* b) Word	*–none–*	135
SimpleExec_TextVstart (BootRec* b) Word	*–none–*	135
Size (Fpage f) Word	*–none–*	35
SizeLog2 (Fpage f) Word	*–none–*	35
SmallSpace (Word location, size) Word	*–none–*	87
SpaceControl (ThreadId SpaceSpecifier, Word control, Fpage KernelInterfacePageArea, UtcbArea, Word& old_Control) Word	SPACECONTROL	43
SpeculativeMemory Word const	*–n/a–*	126
Start (ThreadId t) void	EXCHANGEREGISTERS	21
Start (ThreadId t, Word sp, ip, flags) void	EXCHANGEREGISTERS	21
Start (ThreadId t, Word sp, ip) void	EXCHANGEREGISTERS	21
Stop (ThreadId t) ThreadState	EXCHANGEREGISTERS	21
Stop (ThreadId t, Word& sp, ip, flags) ThreadState	EXCHANGEREGISTERS	22
StoreMR (int i, Word& w) void	*–none–*	11
StoreMR (int i, Word& w) void	*–none–*	48
StoreMRs (int i, k, Word& $[k]$ w) void	*–none–*	11
StoreMRs (int i, k, Word& $[k]$ w) void	*–none–*	48
Store (MsgTag t, Msg& msg) void	*–none–*	48
ThreadControl (ThreadId dest, SpaceSpecifier, Scheduler, Pager, SendRedirector, ReceiveRedirector, void* UtcbLocation) Word	THREADCONTROL	25
ThreadIdBits (void* KernelInterface) Word	*–none–*	8
ThreadId data type	*–n/a–*	14
$==$ (ThreadId l, r) Bool	*–none–*	15
ThreadIdSystemBase (void* KernelInterface) Word	*–none–*	8
ThreadIdUserBase (void* KernelInterface) Word	*–none–*	8
ThreadNo (ThreadId t) Word	*–none–*	15
ThreadState data type	*–n/a–*	22
ThreadSwitch (ThreadId dest) void	THREADSWITCH	28
ThreadWasHalted (ThreadState s) Bool	*–none–*	22
ThreadWasIpcing (ThreadState s) Bool	*–none–*	22
ThreadWasReceiving (ThreadState s) Bool	*–none–*	22
ThreadWasSending (ThreadState s) Bool	*–none–*	22
Timeslice (ThreadId dest, Word & ts, Word & tq) Word	*–none–*	31
TracebufferMemoryType Word const	*–n/a–*	9
Type (BootRec* BootRec) Word	*–none–*	133
Type (MemoryDesc& m) Word	*–none–*	9
UncacheableMemory Word const	*–n/a–*	88
UncachedMemory Word const	*–n/a–*	36
UndefinedMemoryType Word const	*–n/a–*	9
Unmap (ThreadId s, Fpage f) Fpage	MAPCONTROL	40
Unmap (ThreadId s, Word n, Fpage& [n] fpages) Fpage	MAPCONTROL	40
UntypedWordsAcceptor Acceptor const	*–n/a–*	49
UntypedWords (Msg Tag t) Word	*–none–*	47
UserDefinedHandle (ThreadId t) Word	EXCHANGEREGISTERS	21
UserDefinedHandle () Word	*–none–*	18

	used system call	page
UtcbAlignmentLog2 (void* KernelInterface) Word	*–none–*	9
UtcbAreaSizeLog2 (void* KernelInterface) Word	*–none–*	9
UtcbSize (void* KernelInterface) Word	*–none–*	9
Version (ThreadId t) Word	*–none–*	15
waitnotify ThreadId const	*–n/a–*	15
WaitNotify (Word& mask, ThreadId& from) MsgTag	LIPC	56
Wait (ThreadId& from) MsgTag	IPC	56
Writable Word const	*–n/a–*	35
WriteBackMemory Word const	*–n/a–*	126
WriteBackMemory Word const	*–n/a–*	36
WriteBackMemory Word const	*–n/a–*	88
WriteCombiningMemory Word const	*–n/a–*	88
WriteProtectedMemory Word const	*–n/a–*	88
WriteThroughMemory Word const	*–n/a–*	126
WriteThroughMemory Word const	*–n/a–*	36
WriteThroughMemory Word const	*–n/a–*	88
Yield () void	THREADSWITCH	28

Index

!=, 15
+, 35, 47, 49
+=, 35, 47, 49
−, 35, 49
– (ignored), vii
−=, 35, 49
≡ (unchanged), vii
==, 15, 47
∼ (undefined), vii

AbortIpc_and_stop, 22
AbortReceive_and_stop, 22
AbortSend_and_stop, 22
Accept, 49
Accepted, 49
acceptor, 49
ActualSender, 18, 56
Address, 35
address space
    creation/deletion, 41
    mapping, 38
anylocalthread, 15
anythread, 15
ApiFlags, 8
ApiVersion, 8
Append, 48
ArchitectureSpecificMemoryType, 9
AssociateInterrupt, 26
asynch notify, 51
    protocol, 73

BootInfo, 9
BootInfo_EFITables, 135
BootInfo_Entries, 133
BootInfo_FirstEntry, 133
BootInfo_Module, 135
BootInfo_Multiboot, 135
BootInfo_SimpleExec, 135
BootInfo_Size, 133
BootInfo_Valid, 133
booting, 74–76
    arm, 106
    ia32, 91
    mips64, 118
    powerpc, 130
BootLoaderSpecificMemoryType, 9

cache, 65
    address range, 65
    operations, 64
cacheability, 88, 100, 114, 126
CacheControl, 65
CachedMemory, 36
CacheFlushAll, 66
CacheFlushDRange, 66
CacheFlushIRange, 66
CacheFlushRange, 66
CacheFlushRangeInvalidate, 66
CachingEnabledMemory, 126
CachingInhibitedMemory, 126
Call, 55
Clear, 48
Clear_ReceiveBlock, 56
Clear_SendBlock, 57
Clr_CopFlag, 18, 61
CoherentMemory, 36
CompleteAddressSpace, 35
convenience programming interface, vi
ConventionalMemoryType, 9
coprocessors, 61
Copy_regs, 22

DeassociateInterrupt, 26
debug registers, 90
DedicatedMemoryType, 9
DefaultMemory, 36, 88, 126
DisablePreemptionCallback, 32

EFI_MemdescSize, 136
EFI_MemdescVersion, 136
EFI_Memmap, 136
EFI_MemmapSize, 136
EFI_Systab, 136
EnablePreemptionCallback, 32
endian, 3
ErrInvalidParam, 31, 66
ErrInvalidRedirector, 26
ErrInvalidScheduler, 26
ErrInvalidSpace, 26, 43
ErrInvalidThread, 22, 26, 31
ErrKipArea, 43
ErrNoMem, 26
ErrNoPrivilege, 26, 31, 43, 63, 66
ErrorCode, 18, 22, 26, 31, 43, 56, 63, 66
ErrUtcbArea, 26, 43
exception
    handling, 60
    message
        arm, 103
        ia32, 89
        mips64, 115
        powerpc, 127
    protocol, 72
ExceptionHandler, 18, 60
ExchangeRegisters, 21
eXecutable, 35
ExternalFreq, 10

Feature, 9
Fpage, 35
fpage, 34–35, 40
    mapping, 38

unmapping, 34
FpageLog2, 35
FullyAccessible, 35

generic binary interface, vi
generic bootinfo, 131–136
data structure, 131–132
generic record, 132–133
generic programming interface, vi
Get, 47, 48
Get_NotifyBits, 50
Get_NotifyMask, 50
global thread ID, 14
GlobalId, 15
GlobalMemory, 126
GlobalMemoryType, 9
GuardedMemory, 126

High, 10

include files, viii
IntendedReceiver, 18, 56
InternalFreq, 10
interrupt
association, 23
thread ID, 14
IO fpage, 86
IOCombinedMemory, 36
IoFpage, 86
IoFpageLog2, 86
IOMemory, 36
IPC, 51–57
aborting, 19
cross cpu, 54
propagation, 53
Ipc, 55
ipc control registers, 49
IpcFailed, 56
IpcPropagated, 56
IpcRedirected, 56
IpcSucceeded, 56
IpcXcpu, 56
IsNilFpage, 35
IsNilThread, 15
IsVirtual, 9

kernel features, 5
arm, 99
ia32, 85
kernel interface page
location, 41
kernel interface page, 2–10
data structure, 2–6
retrieving, 7–10
KernelGenDate, 8
KernelId, 8
KernelInterface, 8
KernelSupplier, 8
KernelVersion, 8
KernelVersionString, 9
KipAreaSizeLog2, 9

Label, 47, 48
LargeSpace, 87
Lcall, 56
Lipc, 55
lipc, 51
Load, 48
LoadMR, 11, 48
LoadMRs, 11, 48
local ipc, 51
LocalMemory, 126
logical interface, vi
Low, 9
LreplyWait, 56

Map, 40
map item, 37
MapControl, 40
MBI_Address, 136
memory descriptor, 6, 75–76
MemoryDesc, 9
message registers, 46
arm, 94–95
ia32, 81
mips64, 108–109
powerpc, 120–121
MessageRegisters, 9
messages
generating, 46–48
model specific registers, 90
Module_Cmdline, 135
Module_Size, 135
Module_Start, 135
MR, *see* message registers
MsgTag, 47
MyGlobalId, 15, 17
Myself, 15, 17

Next, 133
Nilpage, 35
Niltag, 47
nilthread, 15
NoAccess, 35
notification, 51
bits, 49
mask, 49
protocol, 73
Notify, 56
NotifyBits, 18
NotifyMask, 18
NotifyMsgAcceptor, 49
NUMAMemoryType, 9
NumMemoryDescriptors, 8
NumProcessors, 8

page
access rights, 4, 34, 37, 70
changing, 37, 38
attributes, 36, 37
arm, 100
ia32, 88
mips64, 114
powerpc, 126
size, 3
pagefault
protocol, 70
Pager, 18, 21
pager, 70
changing, 18, 21, 24
PageRights, 8
PageSizeMask, 8
PhysDesc, 36
physical address, 37

physical descriptor, 36
PreemptedIP, 18, 32
preemption, 32
    protocol, 71
privileged threads, vii
ProcDesc, 9
processor-specific binary interface, vii
ProcessorControl, 63
ProcessorNo, 16
ProcessorNo, 18
propagation, 53
Put, 47, 48

RDMSR, 90
Readable, 35
ReadeXecOnly, 35
Receive, 56
redirection, 24, 53
Reply, 56
ReplyWait, 56
ReservedMemoryType, 9
Rights, 35

Schedule, 31
SchedulePrecision, 9
segments, 90
Send, 55
Set_CopFlag, 18, 61
Set_ExceptionHandler, 18, 60
Set_Label, 48
Set_MsgTag, 47
Set_Notify, 56
Set_NotifyBits, 18, 50
Set_NotifyMask, 18, 50
Set_Pager, 18, 21
Set_PreemptCallbackIP, 18, 32
Set_Priority, 31
Set_ProcessorNo, 31
Set_Propagation, 56
Set_ReceiveBlock, 56
Set_ReceiveRedirector, 26
Set_Rights, 35
Set_SendBlock, 57
Set_SendRedirector, 26
Set_Timeslice, 31
Set_UserDefinedHandle, 18, 21
Set_VirtualSender, 18, 57
SimpleExec_BssPstart, 135
SimpleExec_BssSize, 136
SimpleExec_BssVstart, 135
SimpleExec_Cmdline, 136
SimpleExec_DataPstart, 135
SimpleExec_DataSize, 135
SimpleExec_DataVstart, 135
SimpleExec_Flags, 136
SimpleExec_InitialIP, 136
SimpleExec_Label, 136
SimpleExec_Set_Flags, 136
SimpleExec_Set_Label, 136
SimpleExec_TextPstart, 135
SimpleExec_TextSize, 135
SimpleExec_TextVstart, 135
Size, 35
SizeLog2, 35
small spaces, 87
SmallSpace, 87
SpaceControl, 43
SpeculativeMemory, 126
Start, 21
Stop, 21, 22
Store, 48
StoreMR, 11, 48
StoreMRs, 11, 48
system thread, 14
system-call links, 5
    arm, 96
    ia32, 82
    mips64, 110–113
    powerpc, 122–125
SystemBase, 4

TCR, *see* thread control registers
thread
    creation, 23
    halting, 19
    ID, 14
    id, 15, *see* thread ID
    migration, 29
    priority, 29
    privileged, vii
    startup protocol, 68
    state, 22, 30
    version, 14, 23
thread control registers, 16–18
    arm, 94
    ia32, 80
    mips64, 108
    powerpc, 120
thread ID, 14–15
    retrieving, 17
ThreadControl, 25
ThreadIdBits, 8
ThreadIdSystemBase, 8
ThreadIdUserBase, 8
ThreadNo, 15
ThreadSwitch, 28
ThreadWasHalted, 22
ThreadWasIpcing, 22
ThreadWasReceiving, 22
ThreadWasSending, 22
thumb-mode
    arm, 105
time quantum, 30
Timeslice, 31
timeslice, 29
    donation, 28
tracebuffer, 77
TracebufferMemoryType, 9
Type, 9, 133

UncacheableMemory, 88
UncachedMemory, 36
UndefinedMemoryType, 9
Unmap, 40
UntypedWords, 47
UntypedWordsAcceptor, 49
upward compatibility, vii
UserBase, 4
UserDefinedHandle, 16, 20
UserDefinedHandle, 18, 21
using the API, viii
UTCB
    location, 41
    size, 4, 24, 42

UtcbAlignmentLog2, 9
UtcbAreaSizeLog2, 9
UtcbSize, 9

Version, 15
virtual registers, 11

Wait, 56
WaitNotify, 56
waitnotify, 15
Word, vii
Word16, vii
Word32, vii
Word64, vii
Writable, 35
WriteBackMemory, 36, 88, 126
WriteCombiningMemory, 88
WriteProtectedMemory, 88
WriteThroughMemory, 36, 88, 126
WRMSR, 90

Yield, 28