L4 *eXperimental* Kernel Reference Manual

Version X.2

System Architecture Group
Dept. of Computer Science
Karlsruhe Institute of Technology
(L4Ka Team)
info@l4ka.org

Document Revision 7-6288f0536ce1
October 13, 2011

Contents

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 System Programmer's 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 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)
- Jan Stoess (stoess@ira.uka.de)

Credits

> This manual is 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, Philip Derrin, Kevin Elphinstone, Bryan Ford, Andreas Haeberlen, Hermann Härtig, Gernot Heiser, Michael Hohmuth, Trent Jaeger, Ben Leslie, Jork Löser, Frank Mehnert, Yoonho Park, Marc Salem, Carl van Schaik, Sebastian Schönberg, Cristan Szmajda, Harvey Tuch, Marcus Völp, Neal Walfield, 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

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 binding 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 92) 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 add 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 inaccessible. Any algorithm relying on the value of undefined parameters or bit fields is defined to be incorrect. + No covert channel.

$\boxed{-}$

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

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 specifies 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 definitons 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.
- Clarified order of IPC send and receive.
- Changed the AMD64 and IA32 specific IO port mapping interface. The kernel now uses a custom pagefault label to propagate IO pagefaults to the pager.
- Updated valid encodings for *API Version*, *Kernel Id*, and *Supplier* in the kernel-interface page.
- Make it clear on which processor a new thread starts executing.

- *ProcessorNo* now returns a word rather than int.
- Added functions for reading IO fpages. Fixed include path for using IO fpages.
- Define that the SCHEDULE system call is also allowed if the calling thread resides in same address space as the destination thread.
- Redefine values for IA32 memory attributes to better correspond with the architecture's default Page Attribute Table (PAT) values.

Revision 7

- Removed discontinued architectures IA-64, ARM, Alpha, MIPS, MIPS64, SPARC
- Introduced a new item called control transfer item (CtrlXferItem), which allows specifying control state for IPC and EXCHANGEREGISTERS system-calls in an architecture-dependent manner.
- Added three new flags W,R, and C to the *control* parameter in the EXCHANGEREGISTERS system-call. The new flags allow modifying thread state using CtrlXferItems.
- Added a set of extended protocols for pagefaults, exceptions, and preeemptions, which retrieving and updating thread state via CtrlXferItems.
- Added support for hardware-assisted virtualization for IA-32 and PowerPC-32 processors.
- Introduced protocol and a space control extensions for mapping extended physical (64-bit) pages on IA-32 and PowerPC-32
- Fixed wrong specification of SCHEDULE system-call for AMD-64 processors.

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 or granted 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).

L4 version parts				
Supplier	KernelVer	KernelGenDate	KernelId	KernDescPtr

		InternalFreq	ExternalFreq	ProcDescPtr

		MemoryDesc		MemDescPtr

~	Schedule *SC*	ThreadSwitch *SC*	SystemClock *SC*	+F0 / +1E0
ExchangeRegisters *SC*	Unmap *SC*	Lipc *SC*	Ipc *SC*	+E0 / +1C0
MemoryControl *pSC*	ProcessorControl *pSC*	ThreadControl *pSC*	SpaceControl *pSC*	+D0 / +1A0
ProcessorInfo	PageInfo	ThreadInfo	ClockInfo	+C0 / +180
ProcDescPtr	BootInfo	~		+B0 / +160
KipAreaInfo	UtcbInfo	~		+A0 / +140
~				+90 / +120
~				+80 / +100
~				+70 / +E0
~				+60 / +C0
~		MemoryInfo	~	+50 / +A0
~				+40 / +80
~				+30 / +60
~				+20 / +40
~				+10 / +20
KernDescPtr	API Flags	API Version	$0_{(0/32)}$ ’K’ 230 ’4’ ’L’	+0
+C / +18	+8 / +10	+4 / +8	+0	

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/subsubversion 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 $_{(8)}$	subversion $_{(8)}$	$\sim_{(16)}$

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		Dresden L4.Sec
0x86	*rev*	NICTA N1 (Revision *rev*)
0x04	*rev*	Version 4 (Revision *rev*)

API Flags

$\sim_{(28/60)}$	ww	ee

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

$s_{(4)}$	$\sim_{(12/44)}$	$processors - 1_{(16)}$

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 $_{(22/54)}$	$\sim_{(7)}$	$r\,w\,x$

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 $0 \dots t-1$ are significant for this kernel. Higher bits must all be 0.

$UserBase$

Lowest thread number available for user threads (see page 14). The first three thread numbers will be used for the initial thread of σ_0, σ_1, and root task respectively (see page 92). The version numbers (see page 14) for these initial threads will equal to one.

$SystemBase$

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

ClockInfo

SchedulePrecision $_{(16)}$	ReadPrecision $_{(16)}$

ReadPrecision

Specifies the minimal time difference $\neq 0$ that can be detected by reading the system clock through the SYSTEMCLOCK system call. Basically, this is the precision of the system clock when reading it.

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 as time periods (see page 30).

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 $2^a mk \leq 2^s$.

m UTCB size multiplier.

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

KipAreaInfo

$\sim$ $_{(26/58)}$	s $_{(6)}$

s The size of the kernel interface page area is 2^s.

BootInfo Prior to kernel initialization a boot loader can write an arbitrary value into the BootInfo field of the kernel configuration page (see page 92). 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 plaintext strings. The first plaintext string identifies the current kernel followed by further optional kernel-specific versioning information. The remaining plaintext strings identify architecture dependent kernel features (see Appendix A.3). A zero length string (i.e., a string containing only a 0-character) 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, NICTA
4	3	L4Ka::Strawberry	UKa
5	1	NICTA::Pistachio-embedded	NICTA

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

$MemDescPtr$
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

<table>
<tr><td>$high/2^{10}$ $_{(22/54)}$</td><td colspan="4">$\sim$ $_{(10)}$</td><td>+4 / +8</td></tr>
<tr><td>$low/2^{10}$ $_{(22/54)}$</td><td>v</td><td>$\sim$</td><td>t $_{(4)}$</td><td>$type$ $_{(4)}$</td><td>+0</td></tr>
</table>

$high$ 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., memory not available to user)
0x4	Shared memory (i.e., available to all users)
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 (see page **??**).

t, type ≠ 0xE, type ≠ 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 or granted 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 $_{(8)}$	$\sim_{(16)}$

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 ***ReadPrecision*** (*void* KernelInterface*)

Word ***SchedulePrecision*** (*void* KernelInterface*)

Delivers the SYSTEMCLOCK read precision/maximal jitter for wakeups (both in μs).

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 ***SharedMemoryType***

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 Appendices A.1, **??** and C.1 for specific hardware details.

There are three classes of virtual registers:

- *Thread Control Registers (TCRs),* see page 16
- *Message Registers (MRs),* see page 50
- *Buffer Registers (BRs),* see page 62

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.

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.

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

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

Delivers/sets the value of BR $_i$.

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

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

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

Chapter 2

Threads

2.1 ThreadId [Data Type]

Thread IDs identify threads and hardware interrupts. A thread ID can be *global* or *local*. Global thread IDs are unique through the entire system. They identify threads independently of the address space in which they are used. Local thread IDs exist per address space; the scope of a thread's local ID is only the thread's own address space. In different address spaces, the same local thread ID may identify different and unrelated threads.

Note that any thread has a global *and* a local thread ID. Both global and local thread IDs are encoded in a single word.

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. At least one of the lowermost 6 version bits must be 1 to differentiate a global from a local thread ID.

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 $[0\,, \text{SystemBase})$. The values *SystemBase*, *UserBase*, and t are published in the kernel interface page (see page 4).

global thread ID	thread no $_{(18/32)}$	version$_{(14/32)}$ $\neq 0 \pmod{64}$

global interrupt ID	intr no $_{(18/32)}$	$1_{(14/32)}$

Global thread IDs have a version field whose content can be freely set by those threads that can create and delete threads. However, the lowermost 6 bits of the version must not all be 0, i.e. $v \bmod 64 \neq 0$ must hold for every version v. For hardware interrupts, the version field is always 1.

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.

Local Thread ID

Local thread IDs identify threads within the same address space. They are identified by the 6 lowermost bits being 0.

local thread ID	local id/64 $_{(26/58)}$	000000

Special Thread IDs

Special IDs exist for *nilthread* and two 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.

nilthread	$0_{(32/64)}$

anythread	$-1_{(32/64)}$

anylocalthread	$-1_{(26/58)}$	000000

Generic Programming Interface

#include <l4/thread.h>

struct **THREADID** { Word raw }

ThreadId ***nilthread***

ThreadId ***anythread***

ThreadId ***anylocalthread***

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. The result of comparing a local ID with a global ID will always indicate a mismatch, even if the IDs refer to the same thread.

Bool ***SameThreads*** (*ThreadId l, r*)

{ GlobalId (l) == GlobalId (r) }

Check if thread IDs refer to the same thread. Also works if one ID is local and the other is global.

Bool ***IsNilThread*** (*ThreadId t*)

{ t == *nilthread* }

Bool ***IsLocalId*** (*ThreadId t*)

Bool ***IsGlobalId*** (*ThreadId t*)

Check if thread ID is a local/global one.

ThreadId ***LocalId*** (*ThreadId t*) [*LocalIdOf*]

ThreadId ***GlobalId*** (*ThreadId t*) [*GlobalIdOf*]

Delivers the local/global ID of the specified local thread. Specifying a non-local thread delivers *nilthread* (see EXCHANGEREGISTERS, page 18).

ThreadId ***MyLocalId*** ()

ThreadId ***MyGlobalId*** ()

Delivers the local/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.

VirtualSender/ActualSender $_{(32/64)}$	R/W	see *IPC*
IntendedReceiver $_{(32/64)}$	R-only	see *IPC*
XferTimeouts $_{(32/64)}$	R/W	see *IPC*
ErrorCode $_{(32/64)}$	R-only	see system-calls
Preempt Flags $_{(8)}$	R/W	see *Scheduling*
Cop Flags $_{(8)}$	W-only	see *Miscellaneous*
ExceptionHandler $_{(32/64)}$	R/W	see *Miscellaneous*
Pager $_{(32/64)}$	R/W	see *Protocols*
UserDefinedHandle $_{(32/64)}$	R/W	see *Threads*
ProcessorNo $_{(32/64)}$	R-only	see *Miscellaneous*
MyLocalId $_{(32/64)}$	R-only	see *Threads, IPC*
MyGlobalId $_{(32/64)}$	R-only	see *Threads, IPC*

MyGlobalId Global ID of the thread.

MyLocalId Local 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.

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 ***MyLocalId*** ()

ThreadId ***MyGlobalId*** ()

Delivers the local/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.

Word ***XferTimeouts*** ()

void ***Set_XferTimeouts*** (*Word NewValue*)

Delivers/sets the transfer timeouts for the currently running thread (see IPC, page 66).

ThreadId ***IntendedReceiver*** ()

Delivers the intended receiver of last received IPC (see IPC, page 67).

ThreadId ***ActualSender*** ()

Delivers the actual sender of the last propagated IPC (see IPC, page 66).

void ***Set_VirtualSender*** (*ThreadId t*)

Sets the virtual sender for the next deceiving IPC (see IPC, page 66).

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*	⟶	*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.

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.

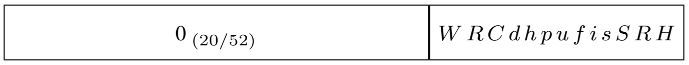
Modifications of the *pager* TCR and the *UserDefinedHandle* TCR become immediately effective, whether the destination thread executes in user mode or in kernel mode.

Input Parameters

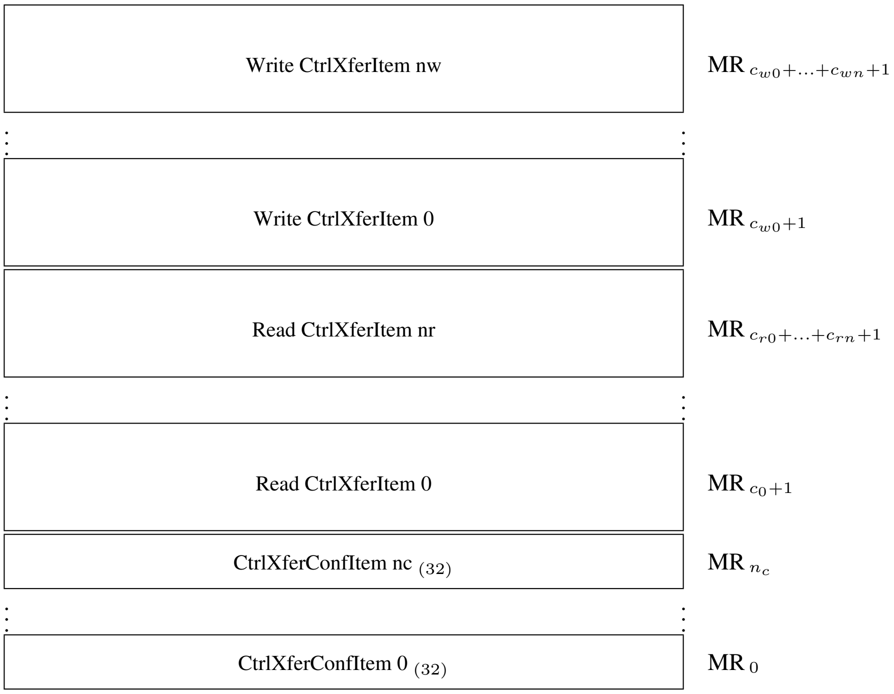
dest

Thread ID of the addressed thread. This may be a local or a global ID. However, the addressed thread must reside in the current address space. Using a local thread ID might be substantially faster in some implementations.

control



CtrlXferItems *MR* $_i$



$W\,R\,C$

The W, R, and C flags refer to extended, control transfer item based state manipulations. The C-flag configures the kernel use an extended control transfer protocol for kernel-generated messages (see Section 7.6). The R and W flags allow direct reading and writing for thread state using control transfer items. All control transfer items are passed in the message register of the invoking thread.

$C = 1$

The caller requests extended state to be sent on kernel-generated messages. The extended state is specified per fault, as control transfer configuration item CtrlXferConfItem (see below). The items start at MR $_0$.

idmask $_{(20/52)}$	fault $_{(8)}$	$1\,1\,0\,C$	MR $_i$

fault

The architecture-specific identifier of the fault, which the kernel-message is generated for.

idmask

A bitmask specifying the control transfer items to be included when sending the message. Identifiers are architecture specific. If bit n in the mask is set to 1, the kernel will append CtrlXferItem number n to the message. Note that the kernel does not permit selecting individual registers of a specific CtrlXferItem is not allowed; rather it always includes the *full contents* of the particular item.

C

The *continuation* flag c is set for all but the last CtrlXferConfItem .

$R = 1$

The caller requests extended state to be read from the destination thread. Extended state is passed in the caller's message registers, as a set of control transfer items. If $C = 1$, the first CtrlXferItem to be read follows *directly* after the last CtrlXferConfItem . If $C = 0$, read item starts at MR $_0$. tyeThe *continuation* flag c is set for all but the last CtrlXferConfItem . Note that each read CtrlXferConfItem must provide enough space to cover all registers in the item to be read (i.e., if the item's mask specifies k registers to be read, the item must contain words for k registers).

$W = 1$

The caller requests extended state to be written from the destination thread. Extended state is passed in the caller's message registers, as a set of control transfer items. If $R = 1$, the first write CtrlXferItem follows directly after the last read item. Else, if $C = 1$, the first write CtrlXferItem follows *directly* after the last CtrlXferConfItem . Else, $C = 0$, write starts at MR $_0$. The *continuation* flag c is set for all but the last CtrlXferConfItem .

$h\,p\,u\,f\,i\,s$

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 opereration 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 delayed 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.* An IPC operation that is currently sending a message will be *aborted.*

$R = 0$

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 delayed 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.

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 *≠ nilthread, input parameter* dest *was a local thread ID*

global thread ID of the addressed thread. ExchangeRegisters succeeded.

result *≠ nilthread, input parameter* dest *was a global thread ID*

local 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 ≠ 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>

ThreadId ***GlobalId*** (*ThreadId t*) [*GlobalIdOf*]

{ if (IsLocalId (t)) ExchangeRegisters (t,0,–. . .) else t }

Delivers global ID of specified local thread. Specifying a non-local thread delivers *nilthread*.

ThreadId ***LocalId*** (*ThreadId t*) [*LocalIdOf*]

{ if (IsGlobalId (t)) ExchangeRegisters (t,0,–. . .) else t }

Delivers local ID of specified local thread. Specifying a non-local thread delivers *nilthread*.

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

void ***Set_UserDefinedHandle*** (*ThreadId t, Word handle*) [*Set_UserDefinedHandleOf*]

Delivers/sets the user defined handle of specified local thread. Result of specifying a non-local thread is undefined.

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

void ***Set_Pager*** (*ThreadId t, p*) [*Set_PagerOf*]

Delivers/sets the pager for specified local thread. Result of specifying a non-local thread is undefined.

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 local thread (if halted). Abort any ongoing IPC operations. Optionally modify stack pointer, instruction pointer, and processor flags according to function parameters. Result of specifying a non-local thread is undefined.

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

ThreadState ***Stop*** (*ThreadId t, Word& sp, ip, flags*) [*Stop_SpIpFlags*]

Halt execution of specified local 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. Result of specifying a non-local thread is undefined.

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.

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*			
*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 activeted 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*. The new thread will execute on the same processor where the activating *ThreadControl* was invoked

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. *Must be a global thread ID.* 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, 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.

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.

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.

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 alignement 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* paramter specified an invalid thread ID, or was set to *nilthrad* 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.

Pagefaults

No pagefaults will happen.

Generic Programming Interface

System-Call Function:

#include <l4/thread.h>

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

Convenience Programming Interface

Derived Functions:

#include <l4/thread.h>

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

Associate a handler thread with the specified interrupt source.

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

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

Support Functions:

Word ***ErrorCode*** ()

Word ***ErrNoPrivilege***

Word ***ErrInvalidThread***

Word ***ErrInvalidSpace***

Word ***ErrInvalidScheduler***

Word ***ErrUtcbArea***

Word ***ErrNoMem***

Chapter 3

Scheduling

3.1 Clock [Data Type]

On both 32-bit and 64-bit processors, the system clock is represented as a 64-bit unsigned counter. The clock measures time in 1 μs units, independent of the processor frequency. Although the clock base is undefined, it is guaranteed that the counter will not overflow for at least 1,000 years.

Generic Programming Interface

#include <l4/schedule.h>

struct **CLOCK** { Word64 raw }

Convenience Programming Interface

#include <l4/schedule.h>

Clock + (*Clock l, r*) [*ClockAdd*]

Clock + (*Clock l, Word64 r*) [*ClockAddUsec*]

Clock + (*Clock l, int r*)

Clock − (*Clock l, r*) [*ClockSub*]

Clock − (*Clock l, Word64 r*) [*ClockSubUsec*]

Clock − (*Clock l, int r*)

Adds/subtracts a number of μs to/from a clock value. Delivers new clock value. Does not modify the old clock value.

Bool < (*Clock l, r*) [*IsClockEarlier*]

Bool > (*Clock l, r*) [*IsClockLater*]

Bool <= (*Clock l, r*)

Bool >= (*Clock l, r*)

Bool == (*Clock l, r*) [*IsClockEqual*]

Bool != (*Clock l, r*) [*IsClockNotEqual*]

Compares two clock values.

3.2 SYSTEMCLOCK [Systemcall]

$\longrightarrow$ *Clock clock*

Delivers the current system clock. Typically, the operation does not enter kernel mode.

Pagefaults

No pagefaults will happen.

Generic Programming Interface

System-Call Function:

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

Clock SystemClock ()
```

3.3 Time [Data Type]

Time values are used to specify send/receive timeouts for IPC operations (see page 65) and time quanta for scheduling (see page 33). The unit for time periods as well as for time points is 1 μs. Clock ticks thus happen every μs.

Relative time values specify a time period. Time periods are encoded as un-normalized 16-bit floating-point numbers. (Note that for easier handling the mantissa can have leading 0-bits.) The shortest non-zero time period that can be specified is 1 μs, the longest finite period slightly exceeds 610 hours. Two special periods frequently used for timeouts are 0 and ∞, a never ending period. The values 0 and ∞ have special encodings.

time period:

0	$e_{(5)}$	$m_{(10)}$	$= 2^e m \ \mu\text{s}$

$0_{(16)}$	$= \infty$

0	$1_{(5)}$	$0_{(10)}$	$= 0$

Absolute time values specify a point in time. They are only valid for a limited period, at maximum 67 seconds.

time point:

1	e $_{(4)}$	c	$m_{(10)}$

For a semantical description of time-point values, we use $Clock$ to denote the current clock value in μs, $x_{[i]}$ to denote bit i of x, and $x_{[i,j]}$ to denote the number consisting of bits i to j of x. Then, the time-point value (c, m, e) specifies the point:

$$t = \begin{cases} 2^e \cdot \left(m + Clock_{[63,e+9]} \cdot 2^{10}\right) & \text{if } \ Clock_{[e+10]} = c \\ 2^e \cdot \left(m + Clock_{[63,e+9]} \cdot 2^{10} + 2^{10}\right) & \text{if } \ Clock_{[e+10]} \neq c \end{cases}$$

Absolute time values are thus the more precise the nearer in the future they are.

Absolute time values with maximal precision become invalid just after the clock has reached the specified point in time. The validity interval can be expanded, but only by reducing the precision. In general, a time-point value (c, m, e) that is constructed when the current clock value is C_0 is valid from C_0 up to

$$C_0 + (2^{10} - 1) \cdot 2^e$$

Therefore, a time-point value that should remain valid for 10 ms can have a precision of 10 μs whereas a value that should remain valid for an entire second can only have a precision of 1 ms. In general, a precision of 0.1% *of the required validity interval* can be achieved.

Generic Programming Interface

#include <l4/schedule.h>

struct **TIME** { Word16 raw }

Time ***Never***

Time ***ZeroTime***

Time ***TimePeriod*** (*Word64 microseconds*)

Time ***TimePoint*** (*Clock at*)

Convenience Programming Interface

#include <l4/schedule.h>

Time $+$ (*Time l, Word r*) [*TimeAddUsec*]

Time $+=$ (*Time l, Word r*) [*TimeAddUsecTo*]

Time $-$ (*Time l, Word r*) [*TimeSubUsec*]

Time $-=$ (*Time l, Word r*) [*TimeSubUsecFrom*]

Adds/subtracts a number of microseconds to/from a time value.

Time $+$ (*Time l, r*) [*TimeAdd*]

Time $+=$ (*Time l, r*) [*TimeAddTo*]

Time $-$ (*Time l, r*) [*TimeSub*]

Time $-=$ (*Time l, r*) [*TimeSubFrom*]

Adds/subtracts a time period to/from a time value. The result of adding/subtracting a time point is undefined.

Bool $>$ (*Time l, r*) [*IsTimeLonger*]

Bool $>=$ (*Time l, r*)

Bool $<$ (*Time l, r*) [*IsTimeShorter*]

Bool $<=$ (*Time l, r*)

Bool $==$ (*Time l, r*) [*IsTimeEqual*]

Bool $!=$ (*Time l, r*) [*IsTimeNotEqual*]

Compares two time values. The result of comparing a time period with a time point, or vice versa, is undefined.

3.4 THREADSWITCH [Systemcall]

ThreadId dest ⟶ *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 *≠ 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.

3.5 SCHEDULE [Systemcall]

ThreadId	*dest*	$\longrightarrow$	*Word*	*result*
Word	*time control*		*Word*	*time control*
Word	*processor control*			
Word	*prio*			
Word	*preemption control*			

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 resides in the same address space 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 reside in the same address space 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 $\neq -1$.

time control

ts len $_{(16)}$	total quantum $_{(16)}$

ts len

New timeslice length for the destination thread. The timeslice length is specified as a time period (see page 30). Absolute time values and the value 0 are illegal. A timeslice length of ∞, however, can be specified. 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.

Writing the timeslice length initializes the current quantum with the new length. After the quantum is exhausted, the thread is preempted while the quantum is reloaded with *ts len* for the next timeslice.

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. The total quantum is specified as a time period (see page 30). Absolute time values are illegal. A total quantum of ∞ can be specified.

prio

0 $_{(24/56)}$	prio $_{(8)}$

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

preemption control

0 $_{(8/40)}$	sensitive prio $_{(8)}$	maximum delay $_{(16)}$

sensitive prio Preemptions by threads that run on a priority lower or equal to this *sensitive prio* will, (a) if the *delay-preemption* flag is set, be delayed until the thread executes a *thread switch (nilthread)* system call; and (b) if the *signal-preemption* flag is set, raise a preemption fault to the exception handler.
No preemption delays or signaling will occur if preempted by a thread having a higher priority than *sensitive prio*, regardless of the state of the *delay-preemption* and *signal-preemption* flags.

maximum delay The maximum time in μs a pending preemption can be delayed in the destination thread. The value 0 effectively disables preemption delay.

processor control

0 $_{(16/48)}$	processor number $_{(16)}$

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.

Output Parameters

result

$\sim$ $_{(24/56)}$	$tstate$ $_{(8)}$

$tstate$ = 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.

***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 control

rem ts $_{(16)}$	rem total $_{(16)}$

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, Word TimeControl, ProcessorControl, prio, PreemptionControl, Word& old_TimeControl*)

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, Time & ts, Time & tq*)
Delivers the remaining timeslice and total quantum of the given thread.

Word ***Set_Timeslice*** (*ThreadId dest, Time ts, Time tq*)
{ Schedule (dest, ts * 2^{16} + tq, -1, -1, -1) }

Word ***Set_PreemptionDelay*** (*ThreadId dest, Word sensitivePrio, Word maxDelay*)
{ Schedule (dest, -1, -1, -1, SensitivePrio * 2^{16} + MaxDelay) }

Support Functions:

Word ***ErrorCode*** ()

Word ***ErrNoPrivilege***

Word ***ErrInvalidThread***

Word ***ErrInvalidParam***

3.6 Preempt Flags [TCR]

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

Preempt Flags

I	d	s	$\sim$

The ds-flags are used to control the microkernel. User threads can set/reset them. The I-flag signals an event to the user. It is set by the microkernel and typically read/reset by the user.

$s = 0$ Asynchronous preemptions are not signaled to the exception handler.

$s = 1$ Asynchronous preemptions are signaled as preemption faults to the exception handler. If $d = 0$ this happens immediately. Otherwise, it is delayed until the thread continues execution after the preemption.

$d = 0$ All asynchronous preemptions happen immediately. If they are signaled as preemption faults $(s = 1)$, this happens *after* the preemption took place, i.e., when the thread gets reactivated.

$d = 1$ Asynchronous preemptions are delayed if the priority of the preemptor is lower or equal than the *sensitive priority* for the current thread. (The sensitive priority is set by the scheduler, see page 34.) A delayed preemption does not interrupt the current thread immediately but is postponed until the current thread invokes a systemcall *thread switch (nilthread).* However, a pending preemption must not be delayed for longer than the *maximum delay* that was set by the thread's scheduler. Such a preemption-delay overflow resets the d bit and is signaled to the exception handler.

$I = 0$ No asynchronous preemption is pending.

$I = 1$ An asynchronous preemption is currently pending, i.e., the thread should as soon as possible reset the d-flag and invoke *thread switch*. Invoking *thread switch* re-enables the *maximum delay* for the next delayed asynchronous preemption.
Invoking *thread switch* is not required if no asynchronous preemption is pending $(I = 0)$ after the user thread has reset the d-flag.

Generic Programming Interface

#include <l4/schedule.h>

Bool ***EnablePreemptionFaultException*** ()

Bool ***DisablePreemptionFaultException*** ()

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

Bool ***DisablePreemption*** ()

Bool ***EnablePreemption*** ()

Sets/resets the d-flag and delivers the old d-flag value (true = set).

Bool ***PreemptionPending*** ()

Resets the I-flag and delivers the old I-flag value (true = set).

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 sans 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, i.e., $b \equiv 0 \pmod{2^s}$, 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 $(b, 2^s)$	$b/2^{10}$ $_{(22/54)}$	s $_{(6)}$	$0\,r\,w\,x$

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

complete	0 $_{(22/54)}$	$s = 1$ $_{(6)}$	$0\,r\,w\,x$

nilpage	0 $_{(32/64)}$

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/granted page *provided that the mapper itself* possesses that access right. If the mapper does not have the access right itself or if the bit is set to zero the mapped/granted page will not get the corresponding 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, int 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 UNMAP [Systemcall]

Word control $\longrightarrow$ *void*

The specified fpages (located in MR $_{0\ldots}$) are unmapped. Fpages are mapped as part of the IPC operation (see page 64).

Input Parameters

control

0 $_{(25/57)}$	f	k $_{(6)}$

k Specifies the highest MR $_k$ that holds an fpage to be unmapped. The number of fpages is thus $k+1$.

$f=0$ The fpages are unmapped recursively in all address spaces in which threads of the current address space have mapped them before. However, the fpages remain unchanged in the current address space.

$f=1$ The fpages are unmapped like in the $f = 0$ case and, in addition, also in the current address space.

FpageList *MR* $_{0\ldots k}$ Fpages to be processed.

Fpage *MR* $_i$

fpage $_{(28/58)}$	$0\,r\,w\,x$

Fpage to be unmapped. (The term *unmapped* is used even if effectively no access right is removed.) A nilpage specifies a no-op.

$0rwx$ Any access bit set to 1 revokes the corresponding access right. A 0-bit specifies that the corresponding access right should not be affected. Typical examples:

=0111 Complete unmap of the fpage.

=0010 Partial unmap, revoke writability only. As a result, the fpage is set to read-only.

=0000 No unmap. This case is particularly useful if only *dirty* and *accessed* bits should be read and reset without changing the mapping.

Output Parameters

FpageList *MR* $_{0\ldots k}$ The accessed status bits in the fpages are updated.

Fpage *MR* $_i$

fpage $_{(28/58)}$	0 RWX

The status bits *Referenced*, *Written*, and *eXecuted* of all pages processed by the unmap operation are reset and the bitwise OR-ed old values of all the processed pages are delivered in MR $_{0...k}$. For processors that do not differentiate between read access and execute access, the R and X bits are unified: either both are set or both are reset. Resetting status bits is not a recursive operation. However, the status bit values for pages within the current space will also reflect accesses performed on recursive mappings.

$R = 0$ No part of the fpage has been *Referenced* after the last unmap operation (or after the initial map operation). This includes all recursively mapped pages.
Remark: The meaning of *referenced* slightly differs from *read*. Not being referenced means that not only no read access but that also no write and execute access occurred.

$R = 1$ At least one page of the specified fpage (including all recursive mappings) has been referenced after the last unmap operation (or after the initial map operation). All in-kernel R bits are reset
Remark: The meaning of *referenced* slightly differs from *read*. Write accesses and execute accesses also set the R bit.

$W = 0$ No part of the fpage has been written after the last unmap operation (or after the initial map operation), i.e., the fpage is *clean*. This includes all recursively mapped pages.

$W = 1$ At least one page of the specified fpage (including all recursive mappings) has been written after the last unmap operation (or after the initial map operation), i.e., the fpage is *dirty*.
All in-kernel dirty bits are reset.

$X = 0$ No part of the fpage has been *eXecuted* after the last unmap operation (or after the initial map operation). This includes all recursively mapped pages.

$X = 1$ At least one page of the specified fpage (including all recursive mappings) has been executed after the last unmap operation (or after the initial map operation). All in-kernel X bits are reset.
Remark: For processors that do not differentiate between read and execute accesses, the X bit is set to 1 iff $R = 1$.

Pagefaults

No pagefaults will happen.

Generic Programming Interface

System-Call Function:

#include <l4/space.h>

void ***Unmap*** (*Word control*)

Convenience Programming Interface

Derived Functions:

#include <l4/space.h>

Fpage ***Unmap*** (*Fpage f*) [*UnmapFpage*]
{ LoadMR (0, f); Unmap (0); StoreMR (0, f); f }

void ***Unmap*** (*Word n, Fpage& [n] fpages*) [*UnmapFpages*]
{ LoadMRs (0, n, fpages); Unmap ($n - 1$); StoreMRs (0, n, fpages); }

Recursively unmaps the specified fpage(s) from all address spaces except the current one.

Fpage ***Flush*** (*Fpage f*)

{ LoadMR (0, f); Unmap (64); StoreMR (0, f); f }

void ***Flush*** (*Word n, Fpage& [n] fpages*) [*FlushFpages*]

{ LoadMRs (0, n, fpages); Unmap ($64 + n - 1$); StoreMRs (0, n, fpages); }

Recursively unmaps the specified fpage(s) from all address spaces, including the current one.

Fpage ***GetStatus*** (*Fpage f*)

{ LoadMR (0, f $-$ *FullyAccessible*); Unmap (0); StoreMR (0, f); f }

Resets and delivers the status bits of the specified fpage.

Bool ***WasReferenced*** (*Fpage f*)

Bool ***WasWritten*** (*Fpage f*)

Bool ***WaseXecuted*** (*Fpage f*)

Checks the status bits of specified fpage. The specified fpage must be the output of an *Unmap ()*, *Flush ()*, or *GetStatus ()* function.

4.3 SPACECONTROL [Privileged Systemcall]

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

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.

KipAreaInfo [KernelInterfacePage Field]

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

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

s The size of the kernel interface page area is 2^s.

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.

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 $2^a m$.

Redirector = *nilthread*

The current redirector setting for the specified space is not modified.

Redirector = *anythread*

All threads within the specified space are allowed to communicate with any thread in the system.

Redirector ≠ *anythread,* ≠ *nilthread*

All threads within the specified address space are only allowed to send an IPC to a local thread or to a thread in the same address space as the specified redirector. All other send operations will be deflected to the redirector, the *redirected bit* (see page 67) in the received message will be set, and the *IntendedReceiver* TCR will indicate the intended receiver of the message.

---

control The control field is architecture specific (see Appendix A.5). 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* ≠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, ThreadId Redirector, 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 64). Basically, the sender writes a message into the sender's message registers (MRs) and the receiver reads it from the receiver's MRs. Each thread has 64 MRs, MR $_{0...63}$. A message can use some or all MRs to transfer untyped words; it can include memory strings and 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 3 sections: the mandatory *message tag,* followed by an optional *untyped-words* section, followed by an optional *typed-items* 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 $_{(4)}$	t $_{(6)}$	u $_{(6)}$

u Number of untyped words following word 0. MR $_{1...u}$ hold the untyped words. $u = 0$ denotes a message without untyped words.

t Number of typed-item words following the untyped words or the message tag if no untyped words are present. The typed items use MR $_{u+1...u+t}$. A message without typed items has $t = 0$.

flags Message flags, see IPC systemcall, page 64.

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

***untyped words* [MR$_{1...u}$]**

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* [MR$_{u+1...u+t}$]**

The optional typed-items section is a sequence of items such as *string items* (page 59), *map items* (page 55), *grant items* (page 57), and (page 58) *ctrl transfer items*.
Typed message items have their type encoded in the lowermost 4 bits of their first word:

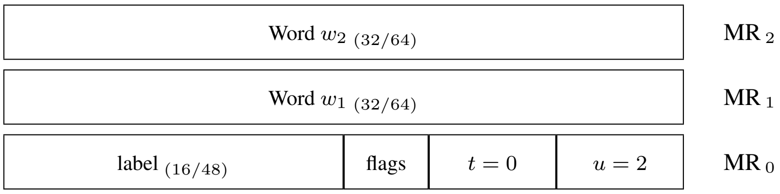
$0hhC$	StringItem	see page 59
$100C$	MapItem	see page 55
$101C$	GrantItem	see page 57
$110C$	CtrlXferItem	see page 58
$111C$	*Reserved*	

The C bit signals whether the typed item is followed by another typed item ($C = 1$) or is the last one of the typed-item section ($C = 0$). The typed items *must* exactly fit into MR $_{u+1...u+t}$.

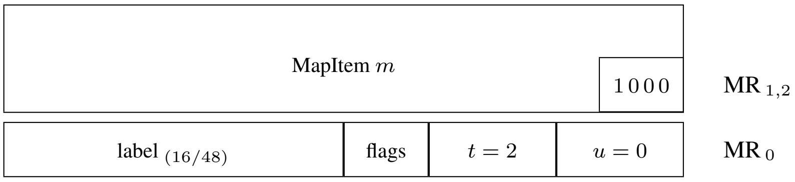
Note that C and t redundantly describe the message. This is by intention. The C bit allows efficient message parsing, whereas $t + u$ can be used to store all MRs of a message to memory without parsing the complete message. Upon message sending, the C bits are completely ignored. The kernel will, however, ensure that the MRs on the receiver side will have the C bits set properly.

Example Messages

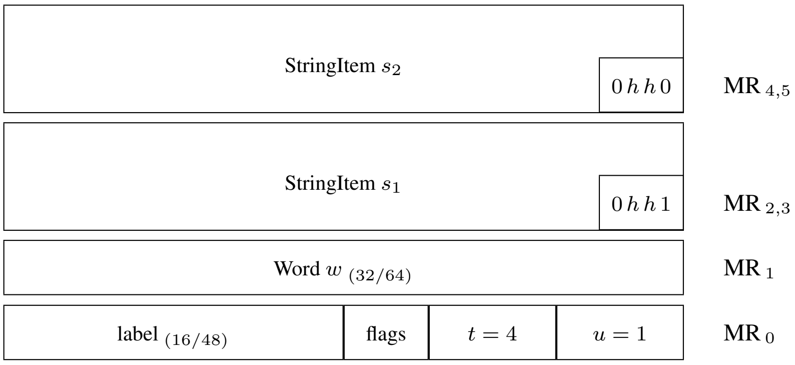
struct (**label*****, Word [2]** w**)***



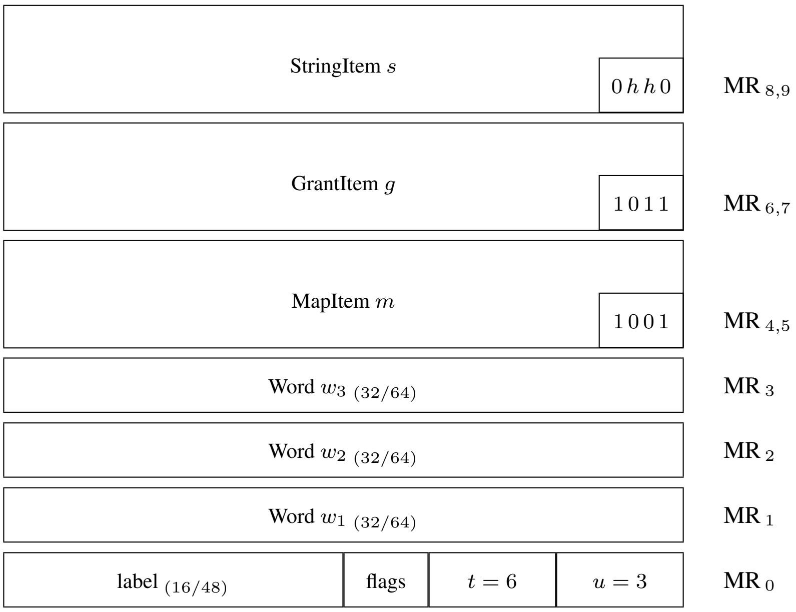
struct (**label*****, MapItem** m**)***



struct (**label*****, Word** w**, StringItem** s_1, s_2**)***



struct (**label*****, Word [3]** w**, MapItem** m**, GrantItem** g**, StringItem** s**)***



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 or typed words, 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*)

Word ***TypedWords*** (*Msg Tag t*)

Delivers the message label, number of untyped words, and number of typed 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, int t,* {*MapItem, GrantItem, StringItem*}*& Items*) [*MsgPut*]

Loads the specified parameters into the memory object *msg*. The parameters u and t respectively indicate number of untyped words and number of typed words (i.e., the total size of all typed items). It is assumed that the *msg* object is large enough to contain all items.

void ***Get*** (*Msg& msg, Word& ut,* {*MapItem, GrantItem, StringItem*}*& Items*) [*MsgGet*]

Stores the *msg* object into the specified parameters. Type consistency between the message in the memory object and the specified parameter list is *not* checked.

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 MR $_{0...}$ 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*]

void ***Append*** (*Msg& msg, MapItem m*) [*MsgAppendMapItem*]

void ***Append*** (*Msg& msg, GrantItem g*) [*MsgAppendGrantItem*]

void ***Append*** (*Msg& msg, StringItem s*) [*MsgAppendSimpleStringItem*]

void ***Append*** (*Msg& msg, StringItem& s*) [*MsgAppendStringItem*]

Appends an untyped or a typed item to the *msg* object. Compound strings must always be passed in by reference. A compound string passed by value will be treated as a simple string (see page 59). 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.

void ***Put*** (*Msg& msg, Word t, MapItem m*) [*MsgPutMapItem*] *void* ***Put*** (*Msg& msg, Word t, GrantItem g*) [*MsgPutGrantItem*]

void ***Put*** (*Msg& msg, Word t, StringItem s*) [*MsgPutSimplStringItem*]

void ***Put*** (*Msg& msg, Word t, StringItem& s*) [*MsgPutStringItem*]

void ***Put*** (*Msg& msg, Word t, CtrlXferItem c*) [*MsgPutCtrlXferItem*]

Puts a typed item into the *msg* object, starting at typed word position t (first typed word has position 0). Compound strings must always be passed in by reference. A compound string passed by value will be treated as a simple string (see page 59). It is assumed that that the object has enough typed words to contain the new item.

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.

Word ***Get*** (*Msg& msg, Word t, MapItem& m*) [*MsgGetMapItem*]

Word ***Get*** (*Msg& msg, Word t, GrantItem& g*) [*MsgGetGrantItem*]

Word ***Get*** (*Msg& msg, Word t, StringItem& s*) [*MsgGetStringItem*]

Word ***Get*** (*Msg& msg, Word t, CtrlXferItem& c*) [*MsgGetCtrlXferItem*]

Delivers the typed item starting at typed word position t. It is assumed that the requested item is of the right size and type. Returns the size (in words) of the delivered item.

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

5.2 MapItem [Data Type]

An *fpage* (see page 40) or IO fpage that should be mapped is sent to the mappee as part of a message. A map operation is a no-op within the same address space. The fpage is specified by a two-word descriptor:

<table>
<tr><td colspan="2">snd fpage $_{(28/60)}$</td><td>$0\,r\,w\,x$</td><td>MR $_{i+1}$</td></tr>
<tr><td>snd base / 1024 $_{(22/54)}$</td><td>0 $_{(6)}$</td><td>$1\,0\,0\,C$</td><td>MR $_{i}$</td></tr>
</table>

access rights rwx The effective access rights for the newly mapped page are calculated by bitwise AND-ing the access rights specified in the *snd fpage* and the access rights that the mapper itself has on that fpage. As such, the mapper can restrict the effective access rights but not widen them.

snd base The send base specifies the semantics of the map operation if the size of the *snd fpage* is larger or smaller than the window in which the receiver is willing to accept a mapping (see page 62). If the size of the *snd fpage*, 2^s, is larger than the receive window, 2^r, the send base indicates which region of the *snd fpage* is transmitted. More precisely:

$$\begin{aligned} send\ region = fpage\,(addr_s + 2^r k, 2^r),\ \text{for some } k \geq 0 :\\ addr_s + 2^r k \leq addr_s + (snd\,base \bmod 2^s) < addr_s + 2^r k + 2^r \end{aligned}$$

and where $addr_s$ is the base address of the *snd fpage*. If the size of the *snd fpage*, 2^s, is smaller than the receive window, 2^r, the send base indicates where in the receive window the *snd fpage* is mapped. More precisely:

$$\begin{aligned} receive\ region = fpage\,(addr_r + 2^s k, 2^s),\ \text{for some } k \geq 0 :\\ addr_r + 2^s k \leq addr_r + (snd\,base \bmod 2^r) < addr_r + 2^s k + 2^s \end{aligned}$$

and where $addr_r$ is the base address of the receive window.

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 address space as the existing mapping; *and*
- the mapper maps from the same virtual source 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 not be revoked by mapping. The access rights of the resulting mapping are a bitwise OR of the existing and the new mapping's access rights. Access rights are not extended recursively.

Generic Programming Interface

#include <l4/ipc.h>

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

MapItem ***MapItem*** (*Fpage f, Word SndBase*)

Delivers a map item with the specified fpage and send base.

Bool ***MapItem*** (*MapItem m*) [*IsMapItem*]

Delivers true if map item is valid. Otherwise delivers false.

Fpage ***SndFpage*** (*MapItem m*) [*MapItemSndFpage*]

Word ***SndBase*** (*MapItem m*) [*MapItemSndBase*]

Delivers fpage/send base of map item.

5.3 GrantItem [Data Type]

An *fpage* (see page 40) or IO fpage that should be granted is sent to the mappee as part of a message. It is specified by a two-word descriptor:

snd fpage $_{(28/60)}$		$0\,r\,w\,x$	MR $_{i+1}$
snd base / 1024 $_{(22/54)}$	0 $_{(6)}$	$1\,0\,1\,C$	MR $_i$

access rights rwx The effective access rights for the granted page are calculated by bitwise anding the access rights specified in the *snd fpage* and the access rights that the mapper itself has on that fpage. As such, the granter can restrict the effective access rights but not widen them.

snd base The send base specifies the semantics of the map operation if the size of the *snd fpage* is larger or smaller than the window in which the receiver is willing to accept a mapping (see page 62). If the size of the *snd fpage*, 2^s, is larger than the receive window, 2^r, the send base indicates which region of the *snd fpage* is transmitted. More precisely:

$$\begin{aligned} &\mathit{send\ region} = \mathit{fpage}\,(\mathit{addr}_s + 2^r k, 2^r),\ \text{for some } k \geq 0: \\ &\qquad \mathit{addr}_s + 2^r k \leq \mathit{addr}_s + (\mathit{snd\,base} \bmod 2^s) < \mathit{addr}_s + 2^r k + 2^r \end{aligned}$$

and where addr_s is the base address of the *snd fpage*. If the size of the *snd fpage*, 2^s, is smaller than the receive window, 2^r, the send base indicates where in the receive window the *snd fpage* is mapped. More precisely:

$$\begin{aligned} &\mathit{receive\ region} = \mathit{fpage}\,(\mathit{addr}_r + 2^s k, 2^s),\ \text{for some } k \geq 0: \\ &\qquad \mathit{addr}_r + 2^s k \leq \mathit{addr}_r + (\mathit{snd\,base} \bmod 2^r) < \mathit{addr}_r + 2^s k + 2^s \end{aligned}$$

and where addr_r is the base address of the receive window.

Pages already mapped in the grantee's address space that would conflict with new mappings are implicitly unmapped before new pages are mapped.

Generic Programming Interface

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

struct GRANTITEM { Word raw [2] }
```

GrantItem ***GrantItem*** (*Fpage f, Word SndBase*)

Delivers a grant item with the specified fpage and send base.

Bool ***GrantItem*** (*GrantItem g*) [*IsGrantItem*]

Delivers true if grant item is valid. Otherwise delivers false.

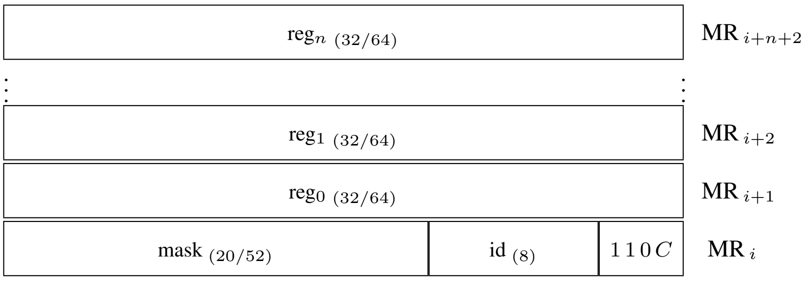
Fpage ***SndFpage*** (*GrantItem g*) [*GrantItemSndFpage*]

Word ***SndBase*** (*GrantItem g*) [*GrantItemSndBase*]

Delivers fpage/send base of grant item.

5.4 CtrlXferItem [Data Type]

A control transfer item specifies a control state such as instruction pointer, stack pointer, or general-purpose registers for the receiver of the message. The new values are automatically set by the kernel upon receiving the item. The contents of a control transfer item are architecture-specific. In general, a control transfer item is specified as follows:



id An identifier specifying the set of control transfer registers set by this item. Identifiers are architecture specific.

mask A bitmask specifying the registers in the set to be modified.

Generic Programming Interface

#include <l4/ipc.h>

struct **CTRLXFERITEM** { Word raw [*] }

Bool ***CtrlXferItem*** (*CtrlXferItem& c*) [*IsCtrlXferItem*]

Delivers true if control transfer item is valid. Otherwise delivers false.

void ***CtrlXferItemInit*** (*CtrlXferItem& c, Word id*)

Initializes the control transfer item with given id.

5.5 StringItem [Data Type]

A string item specifies a sequence of bytes in user space. No alignment is required, the maximal string size is 4 MB. In send messages, such a string is copied to the receiver buffer when transferring the message. String items are also used to specify receive buffers in buffer registers on the receiver's side.

Simple String

A simple string is a contiguous sequence of bytes.

string ptr $_{(32/64)}$				MR $_{i+1}$
string length $_{(22/54)}$	0	0 $_{(5)}$	$0\,h\,h\,C$	MR $_{i}$

string ptr The start address of the string to be sent or the start address of the buffer for receiving a string (no alignment restrictions). However, the string/buffer must fit entirely into the legally addressable user space.

string length The length of the string to be sent or the size of the receive buffer. In the second case, strings up to (including) this length can be received. Maximum string length is 4 M bytes, even if the according field is 54 bits wide on 64-bit processors.

$h\,h$ Cacheability hint. Except for $hh = 00$, the semantics of this parameter depends on the processor type (see Appendices A.6 and **??**).

$hh = 00$ Use the processor's default cacheability strategy. Typically, cache lines are allocated for data read and written (assuming that the processor's default strategy is write-back and write-allocate).

Compound String

A compound string is a noncontiguous string that consists of multiple contiguous substrings which can be scattered around the entire user address space. The substrings must not overlap. For send and receive IPC operations, a compound string is handled as a single logical string. When sending such a string through IPC, the substrings are transferred as if they were one contiguous string (gather). On the receiver side, a compound string buffer is treated as one logical buffer. The corresponding received string is scattered among the compound buffer's substrings.

A compound string can be specified as a sequence of substrings where each substring has the form of a simple string except that the *continuation* flag c is set for all but the last substring. If j subsequent substrings have the same size, e.g., for equally sized buffers, a single length word can be used for all j substrings so that only $j + 1$ words instead of $2j$ words are required.

length word

substring length $_{(22/54)}$	c	$j-1$ $_{(5)}$	$0\,h\,h\,C$

The type information $0hhC$ is only required for the first word of a string descriptor. The field is ignored for further length words in a compound-string descriptor.

j Number of subsequent string-ptr words. These string ptrs specify j substrings that have all the same substring length.

$c = 0$ Continuation flag reset. The compound string descriptor ends with the j^{th} string ptr word following the current length word.

$c = 1$ Continuation flag set. The current length word and j string-ptr words are followed by (at least) one substring descriptor, i.e., another length word, etc.

Example

substring$_{j+1}$ ptr $_{(32/64)}$				MR $_{i+j+2}$
substring$_{j+1}$ length $_{(22/54)}$	0	0 $_{(5)}$	0 $_{(4)}$	MR $_{i+j+1}$
substring$_{j}$ ptr $_{(32/64)}$				MR $_{i+j}$
⋮			⋮	
substring$_{1}$ ptr $_{(32/64)}$				MR $_{i+1}$
substring$_{1\ldots j}$ length $_{(22/54)}$	1	$j-1$ $_{(5)}$	$0\,h\,h\,C$	MR $_{i}$

Generic Programming Interface

#include <l4/ipc.h>

struct **STRINGITEM** { Word raw [*] }

Bool ***StringItem*** (*StringItem& s*) [*IsStringItem*]

Delivers true if string item is valid. Otherwise delivers false.

Bool ***CompoundString*** (*StringItem& s*)

Delivers the c-flag value (true = set).

Word ***Substrings*** (*StringItem& s*)

*void** ***Substring*** (*StringItem& s, Word n*)

Delivers number of substrings/address of nth substring.

StringItem ***StringItem*** (*int size, void* address*)

Delivers a simple string item with the specified size and location.

StringItem & += (*StringItem& dest, StringItem AdditionalSubstring*) [*AddSubstringTo*]

Append substring to the string item. It is assumed that there is enough memory in the string item to contain the new substring.

StringItem & += (*StringItem& dest, void* AdditionalSubstringAddress*) [*AddSubstringAddressTo*]

Append a new substring pointer to the string item. It is assumed that there is enough memory in the string item to contain the new substring pointer.

Convenience Programming Interface

Support Functions:

#include <l4/ipc.h>

struct **CACHEALLOCATIONHINT** { Word raw }

CacheAllocationHint ***UseDefaultCacheLineAllocation***

Bool == (*CacheAllocationHint l, r*) [*IsCacheAllocationHintEqual*]

Bool != (*CacheAllocationHint l, r*) [*IsCacheAllocationHintNotEqual*]

Compares two cache allocation hints.

CacheAllocationHint ***CacheAllocationHint*** (*StringItem s*)

Delivers the cache allocation hint of the string item.

StringItem + (*StringItem s, CacheAllocationHint h*) [*AddCacheAllocationHint*]

StringItem += (*StringItem s, CacheAllocationHint h*) [*AddCacheAllocationHintTo*]

Adds a cache allocation hint to a string item. An already existing hint is overwritten.

5.6 String Buffers And Buffer Registers (BRs) [Pseudo Registers]

For receiving messages that contain string items, the receiver has to specify appropriate string buffers. Such buffers are described by string items (see page 59). A buffer can be contiguous (simple string) or non-contiguous (compound string).

Such buffer descriptors are held in 33 per-thread Buffer Registers BR $_{0\ldots32}$. The number of buffer registers is sufficient to specify, for example, one compound buffer of 31 equally-sized sub-buffers. Up to 16 buffers can be specified provided that not more than 33 BRs are required.

When a message is received, the first message string item is copied into the first buffer string item which starts at BR $_1$; the next message string item is copied to the next buffer string item, etc. The list of buffer strings is terminated by having the C bit in the item type specifier of the last string zeroed.

BRs are *registers* in the sense that they are per-thread objects and can only be addressed directly, not indirectly through pointers. BRs are static objects like TCRs, i.e., they keep their values until explicitly modified. BRs can be mapped to either special registers or to memory locations.

Acceptor **[BR$_0$]**

RcvWindow $_{(28/60)}$	$0\,0\,c\,s$

BR $_0$ specifies which typed items are accepted when a message is received.

RcvWindow Fpage (without access bits) that specifies the address-space window in which mappings and grants are accepted. *Nilpage* denies any mapping or granting; *CompleteAddressSpace* accepts any mapping or granting.

c Control transfer items are accepted iff $c = 1$.

s StringItems are accepted iff $s = 1$.

buffer string items **[BR$_{1\ldots}$]**
contain the valid buffer string items. Ignored if $s = 0$ in BR $_0$.

Generic Programming Interface

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

Acceptor

#include <l4/ipc.h>

struct **ACCEPTOR** { Word raw }

Acceptor ***UntypedWordsAcceptor***

Acceptor ***StringItemsAcceptor***

Acceptor ***CtrlXferItemsAcceptor***

Acceptor ***MapGrantItems*** (*Fpage RcvWindow*)

Delivers an acceptor which allows untyped words, string items, or mappings and grants.

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

Acceptor += (*Acceptor l, r*) [*AddAcceptorTo*]

Adds mappings/grants or string items to an acceptor. Adding a non-nil receive window will replace an existing window.

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

Acceptor −= (*Acceptor l, r*) [*RemoveAcceptorFrom*]

Removes mappings/grants or string items from an acceptor. Removing a non-nil receive window will deny *all* mappings or grants, regardless of the size of the receive window.

Bool ***StringItems*** (*Acceptor a*) [*HasStringItems*]

Bool ***MapGrantItems*** (*Acceptor a*) [*HasMapGrantItems*]

Checks whether string items/mappings are allowed.

Fpage ***RcvWindow*** (*Acceptor a*)

Delivers the address space window where mappings and grants are accepted. Delivers *nilpage* if mappings or grants are not allowed.

void ***Accept*** (*Acceptor a*)

Sets BR $_0$.

void ***Accept*** (*Acceptor a, MsgBuffer& b*) [*AcceptStrings*]

Sets BR $_0$ and loads the buffer description *b* into BR $_{1\ldots}$.

Acceptor ***Accepted*** ()

Delivers BR $_0$.

Convenience Programming Interface

MsgBuffer

#include <l4/ipc.h>

struct **MsgBuffer** { Word raw [32] }

void ***Clear*** (*MsgBuffer& b*) [*MsgBufferClear*]

Clears the message buffer (i.e., inserts a single empty string into it).

void ***Append*** (*MsgBuffer& b, StringItem s*) [*MsgBufferAppendSimpleRcvString*]

void ***Append*** (*MsgBuffer& b, StringItem * s*) [*MsgBufferAppendRcvString*]

Appends a string buffer to the message buffer. Compound strings must always be passed in by reference. A compound string passed by value will be treated as a simple string. It is assumed that there is enough memory in the message buffer object to contain the new string buffer.

Low-Level BR Access

#include <l4/ipc.h>

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

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

Delivers/sets the value of BR $_i$.

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

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

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

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 BRs.

5.7 IPC [Systemcall]

ThreadId	*to*	⟶	*ThreadId*	*from*
ThreadId	*FromSpecifier*			
Word	*Timeouts*			

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 synchronous and unbuffered: 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 until a period specified by the sender has elapsed without the destination becoming ready to receive.

IPC can be used to copy data as well as to *map* or *grant* fpages from the sender to the recipient. For the description of messages see page 50. A single IPC call combines an optional send phase followed by 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, BRs and some TCRs. *RcvTimeout* and *SndTimeout* are directly specified as system-call parameters. Each timeout can be 0, ∞ (i.e., never expire), relative or absolute. For details on timeouts see page 30.

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 specified as a local thread ID; *and*
- a receive phase is included; *and*
- the destination thread runs on the same processor; *and*
- the RcvTimeout is ∞, *and*
- the IPC includes no map/grant operations.

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 = *anythread*

IPC includes a receive phase. Incoming messages are accepted from any thread (including hardware interrupts).

FromSpecifier = *anylocalthread*

IPC includes a receive phase. Incoming messages are accepted from any thread that resides in the current address space.

FromSpecifier ≠ *nilthread,* ≠ *anythread,* ≠ *anylocalthread*

Ipc includes a receive phase. Incoming messages are accepted only from the specified thread. (Note that hardware interrupts can be specified.)

Timeouts

SndTimeout $_{(16)}$	RcvTimeout $_{(16)}$

RcvTimeout The receive phase waits until either a message transfer starts or the *RcvTimeout* expires. Ignored for send-only IPC operations.
For relative receive timeout values, the receive timeout starts to run *after* the send phase has successfully completed. If the receive timeout expires before the message transfer has been started IPC fails with “receive timeout”. A pending incoming message *is* received if the timeout period is 0.

SndTimeout If the send timeout expires before the message transfer could start the IPC operation fails with “send timeout”. A send timeout of 0 ensures that IPC happens only if the addressed receiver is ready to receive when the send IPC operation is invoked. Otherwise, IPC fails immediately, i.e., without blocking.

MsgTag **[MR$_0$]**

label $_{(16/48)}$	0 $_{(3)}$	p	t $_{(6)}$	u $_{(6)}$

Message head of the message to be sent. Only the upper 16/48 bits are freely available. The lower 16 bits hold the *SndControl* parameter. It describes the message to be sent and contains some control bits; ignored if no send phase.

u Number of untyped words following word 0. MR $_{1...u}$ hold the untyped words. $u = 0$ denotes a message with no untyped words.

t Number of words holding typed items that follow the untyped words (or the message tag if no untyped words are present). The typed items use MR $_{u+1}$ and following MRs, potentially up to MR $_{63}$. $t = 0$ denotes a message without typed items.

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.

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

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

[MR$_{u+1...u+t}$] Typed items to be sent. Ignored if no send phase.

XferTimeouts **[TCR]**

XferTimeout Snd $_{(16)}$	XferTimeout Rcv $_{(16)}$

Once a message transfer has been started, the time for transferring the message is roughly bounded by the minimum of sender's and receiver's *XferTimeout.* "Roughly" means that xfer timeouts are only checked when message copy raises a pagefault in the sender's or in the receiver's address space. Copying data and mapping/granting is assumed to take no time. A relative transfer timeout always refers to the beginning of the message transfer (actually when the first page fault is raised). Logically, at that point it is transferred into an absolute timeout which then is used as send and receive timeout for the first and all subsequent page-fault RPCs in the message transfer.
If the effective transfer timeout expires during the message transfer, IPC fails with "xfer timeout" (on both sides). Additional information specifies whether the page fault was in the receiver's or in the sender's address space and which part of the message was already transferred. Each thread has two transfer timeouts. One for the send phase and one for the receive phase.

Acceptor **[BR$_0$]**

RcvWindow $_{(28/60)}$	$0\,0\,c\,s$

BR $_0$ specifies which typed items are accepted when a message is received.

RcvWindow Fpage (without access bits) that specifies the address-space window in which mappings and grants are accepted. *Nilpage* denies any mapping or granting; *CompleteAddressSpace* accepts any mapping or granting.

s StringItems are accepted iff $s = 1$.

c CtrlXferItems are accepted iff $c = 1$.

buffer string items **[BR$_1$...]**
contain the valid buffer string items. Ignored if $s = 0$ in BR $_0$.

Output Parameters

from Thread ID of the sender from which the IPC was received. Thread IDs are delivered as *local thread IDs* iff they identify a thread executing in the same address space as the current thread. It does not matter whether the sender specified the destination as local or global id.
Only defined for IPC operations that include a receive phase.

MsgTag **[MR$_0$]**

label $_{(16/48)}$	$E\,X\,r\,p$	t $_{(6)}$	u $_{(6)}$

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.

t Number of received words that hold typed items. $t = 0$ means no typed items. For IPC operations without receive phase, $t = 0$ is delivered.

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

r — Redirected IPC. If reset ($r = 0$) 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.

X — Cross-processor IPC. If reset ($X = 0$) 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 reset ($E = 0$) the IPC operation terminated successful.
If set ($E = 1$) IPC failed. If the send phase was successful but a receive timeout occurred afterwards, or if a message could only be partially transferred, the entire IPC fails. The error code and additional information can be retrieved from the ErrorCode TCR. The fields *label,* t, 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.

[MR$_{u+1\ldots u+k}$] — Typed items that have been received. Undefined if no receive phase.

***ErrorCode* [TCR]**

x (28/56)	e (3)	p

Only defined if the error indicator E in MR $_0$ *is set.* IPC failed, i.e., was not correctly completed. The x field depends on the error code, see below. 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

$\sim$ (28/60)	e (3)	p

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

$e = 1$ — *Timeout*.
From is undefined in this case.

$e = 2$ — *Non-existing* partner. If the error occurred in the send phase, *to* does not exist. (*Anythread* 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.)

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

errors 4,5,6,7

offset (28/60)	e (3)	p

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

offset — The message transfer has been started and could not be completed. The *offset* identifies exactly the number of bytes that have been been transferred successfully so far through string items.

$e = 4$ — *Message Overflow*.
A message overflow can occur (1) if a receiving buffer string is too short, (2) if not enough buffer string items are present, and (4) if a map/grant of an fpage fails because the system has not enough page-table space available. The *offset* in conjunction with the received MRs permits sender and receiver to exactly determine the reason.

$e = 5$ — *Xfer timeout* during page fault in the invoker's address space.

$e = 6$ *Xfer timeout* during page fault in the partner's address space.

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

Pagefaults

Three different types of pagefault can occur during ipc: pre-send, post-receive, and xfer pagefaults. Only xfer pagefault are critical from a security point of view. Fortunately, messages without strings will never raise xfer pagefaults and need thus no special pagefault provisions:

Pre-send pagefaults
happen in the sender's context *before* the message transfer has really started. The destination thread is not involved; in particular, it is not locked. Therefore, the destination thread might receive another message or time out while the sender's pre-send pagefault is handled. Send and transfer timeouts do not control pre-send pagefaults. Pre-send pagefaults are uncritical from a security point of view, since only the sender's own pager is involved and only the sender could suffer from its potential misbehavior.

Post-receive pagefaults
happen in the receiver's context *after* the message has been transferred. The sender thread is no longer involved, especially, it is no longer locked. Consequently, post-receive pagefault are not subject to send and transfer timeouts. Like pre-send pagefaults, post-receive pagefaults are also uncritical from a security perspective since only the receiver and its pager are involved.

Xfer pagefaults happen while the message is being transferred and both sender and receiver are involved. Therefore, xfer pagefaults are critical from a security perspective: If such a pagefault occurs in the receiver's space, the sender may be starved by a malicious receiver pager. An xfer pagefault in the sender's space and a malicious sender pager may starve the receiver. As such, xfer pagefaults are controlled by the minimum of sender's and receiver's xfer timeouts.

> However, xfer pagefaults can only happen when transferring strings. ***Send messages without strings or receive messages without receive string buffers are guaranteed not to raise xfer pagefaults.***

Generic Programming Interface

System-Call Function:

#include <l4/ipc.h>

MsgTag ***Ipc*** (*ThreadId to, FromSpecifier, Word Timeouts, ThreadId& from*)

MsgTag ***Lipc*** (*ThreadId to, FromSpecifier, Word Timeouts, 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*)
{ Call (to, never, never) }

MsgTag ***Call*** (*ThreadId to, Time SndTimeout, RcvTimeout*) [*Call_Timeouts*]

{ Ipc (to, to, Timeouts (SndTimeout, RcvTimeout), –) }

MsgTag ***Send*** (*ThreadId to*)

{ Send (to, never) }

MsgTag ***Send*** (*ThreadId to, Time SndTimeout*) [*Send_Timeout*]

{ Ipc (to, nilthread, Timeouts (SndTimeout, –), –) }

MsgTag ***Reply*** (*ThreadId to*)

{ Send (to, ZeroTime) }

MsgTag ***Receive*** (*ThreadId from*)

{ Receive (from, never) }

MsgTag ***Receive*** (*ThreadId from, Time RcvTimeout*) [*Receive_Timeout*]

{ Ipc (nilthread, from, Timeouts (–, RcvTimeout), –) }

MsgTag ***Wait*** (*ThreadId& from*)

{ Wait (never, from) }

MsgTag ***Wait*** (*Time RcvTimeout, ThreadId& from*) [*Wait_Timeout*]

{ Ipc (nilthread, anythread, Timeouts (–, RcvTimeout), from) }

MsgTag ***ReplyWait*** (*ThreadId to, ThreadId& from*)

{ ReplyWait (to, never, from) }

MsgTag ***ReplyWait*** (*ThreadId to, Time RcvTimeout, ThreadId& from*) [*ReplyWait_Timeout*]

{ Ipc (to, anythread, Timeouts (TimePeriod(0), RcvTimeout), from) }

void ***Sleep*** (*Time t*)

{ Set_MsgTag (Receive (MyLocalId, t)) }

MsgTag ***Lcall*** (*ThreadId to*)

{ Lipc (to, to, Timeouts (never, never), –) }

MsgTag ***LreplyWait*** (*ThreadId to, ThreadId& from*)

{ Lipc (to, anylocalthread, Timeouts (TimePeriod (0), never), 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_VirtualSender*** (*ThreadId t*)

Sets the virtual sender TCR.

Word ***Timeouts*** (*Time SndTimeout, RcvTimeout*)

Delivers a word containing both timeout values.

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, BR $_0$, 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

$c_7 \dots c_0$

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 MemoryControl [Privileged Systemcall]

Word	*control*	$\longrightarrow$	*Word*	*result*
Word	$attribute_0$			
Word	$attribute_1$			
Word	$attribute_2$			
Word	$attribute_3$			

Set the page attributes of the fpages (MR $_{0...k}$) to the *attribute* specified with the fpage.

Input Parameters

control

0 $_{(26/58)}$	k $_{(6)}$

k — Specifies the highest MR $_k$ that holds an fpage to set the attributes. The number of fpages is thus $k+1$.

attribute$_i$ — Specifies the attribute to associate with an fpage. The semantics of the *attribute*$_i$ values are hardware specific, except for the value 0 which specifies default semantics.

FpageList *MR* $_{0...k}$ — Fpages to be processed.

Fpage *MR* $_i$

fpage $_{(28/60)}$	0 0	a $_{(2)}$

Fpage to change the attributes. A nilpage specifies a no-op.

a — selects *attribute*$_a$ to be set as the fpages memory attributes.

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. Invalid or unsupported memory attribute.

Pagefaults

No pagefaults will happen.

Generic Programming Interface

System-Call Function:

#include <l4/misc.h>

Word ***MemoryControl*** (*Word control, Word& attributes[4]*)

Word ***DefaultMemory***

Convenience Programming Interface

Derived Functions:

#include <l4/misc.h>

Word ***Set_PageAttribute*** (*Fpage f, Word attribute*)
{ Word attributes[4]; attributes[0] = attribute; Set_Rights(f, 0); LoadMR (0, f);
MemoryControl (0, &attributes); }

Word ***Set_PagesAttributes*** (*Word n, Fpage& [n] fpages, Word& [4] attributes*)
{ LoadMRs (0, n, fpages); MemoryControl ($n - 1$, attributes); }

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 $_1$
	$0_{\ (16/48)}$	$0_{\ (4)}$	$t = 0_{\ (6)}$	$u = 2_{\ (6)}$	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

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

To Interrupt Thread

$0_{\ (16/48)}$	$0_{\ (4)}$	$t = 0_{\ (6)}$	$u = 0_{\ (6)}$	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

faulting user-level IP $_{(32/64)}$	MR $_2$

fault address $_{(32/64)}$	MR $_1$

-2 $_{(12/44)}$	$0\,r\,w\,x$	0 $_{(4)}$	$t = 0$ $_{(6)}$	$u = 2$ $_{(6)}$	MR $_0$

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.

***Acceptor* [BR$_0$]**

0 $_{(22/54)}$	$s = 1$ $_{(6)}$	$0\,0\,0\,0$	BR $_0$

The acceptor covers the complete user address space. The kernel accepts mappings or grants into this region on behalf of the faulting thread. The received message is discarded.

From Pager

MapItem / GrantItem	MR $_{1,2}$

0 $_{(16/48)}$	0 $_{(4)}$	$t = 2$ $_{(6)}$	$u = 0$ $_{(6)}$	MR $_0$

7.4 Preemption Protocol [Protocol]

From Preempted Thread

Clock $/2^{(32/64)}{}_{(32/64)}$					MR $_2$
Clock $\mathrm{mod} 2^{(32/64)}{}_{(32/64)}$					MR $_1$
$-3_{(12/44)}$	$0_{(4)}$	$0_{(4)}$	$t = 0_{(6)}$	$u = 2_{(6)}$	MR $_0$

The preemption message contains the system clock when the thread was preempted. The preemption message is sent with relative timeout 0. If the message can not be delivered (e.g., due to timeouts) the message is dropped.

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

exception word $_{k-1\ (32/64)}$					MR $_{k+1}$
⋮					
exception word $_{0\ (32/64)}$					MR $_{2}$
IP $_{(32/64)}$					MR $_{1}$
label $_{(12/44)}$	0 $_{(4)}$	0 $_{(4)}$	$t = 0$ $_{(6)}$	$u = k$ $_{(6)}$	MR $_{0}$

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

exception reply word $_{k-1\ (32/64)}$				MR $_{k+1}$
⋮				
exception reply word $_{0\ (32/64)}$				MR $_{2}$
IP $_{(32/64)}$				MR $_{1}$
0 $_{(16/48)}$	0 $_{(4)}$	$t = 0$ $_{(6)}$	$u = k$ $_{(6)}$	MR $_{0}$

k Number of exception reply words.

IP Location where execution is resumed in the faulting thread.

7.6 Extended Control Transfer Protocol [Protocol]

To facilitate building L4-based virtualization solutions, the kernel can be configured to include extended control transfer state for kernel-generated messages, that is, for thread startups, pagefaults, exceptions, preemptions, and all other, architecture-specific types of messages.

Configuring default control transfer state

By default, the kernel will use the protocols specified in the previous sections. Upon request, the kernel will switch to an extended protocol based on control transfer items. Requests to enable/disable the extended protocol are performed using the EXCHANGEREGISTERS system call and appropriate control transfer configuration items CtrlXferConfItem (CtrlXferConfItem , see Section 2.3).

CtrlXfer Item based kernel message protocol

Extended Thread Start Protocol

Newly created active threads start immediately by receiving a message from its pager. The received message contains one or more control transfer items:

From Pager

CtrlXferItem n				MR $_{c_0+...+c_n+1}$
⋮			⋮	
CtrlXferItem 0				MR $_{c_0+1}$
$0_{\ (16/48)}$	$0_{\ (4)}$	$t = \sum c_i\ _{(6)}$	$u = 0_{\ (6)}$	MR $_0$

Extended Pagefault Protocol

To Pager

CtrlXferItem n				MR $_{c_0+...+c_n+1}$
⋮			⋮	
CtrlXferItem 0				MR $_{c_0+1}$
faulting user-level IP $_{(32/64)}$				MR $_2$
fault address $_{(32/64)}$				MR $_1$
$0_{\ (16/48)}$	$0_{\ (4)}$	$t = \sum c_i\ _{(6)}$	$u = 2_{\ (6)}$	MR $_0$

***Acceptor* [BR₀]**

$0_{(22/54)}$	$s = 1_{(6)}$	0010	BR_0

The acceptor covers the complete user address space and accepts all control transfer items. The kernel accepts mappings or grants into this region on behalf of the faulting thread, and sets the thread state based upon the control transfer items enclosed in the reply message. The received message is discarded.

From Pager

CtrlXferItem n				$\mathrm{MR}_{c_0+\ldots+c_n+3}$
⋮			⋮	
CtrlXferItem 0				MR_{c_0+3}
MapItem / GrantItem				$\mathrm{MR}_{1,2}$
$0_{(16/48)}$	$0_{(4)}$	$t = 2 + \sum c_i\ _{(6)}$	$u = 0_{(6)}$	MR_0

Extended Exception Protocol

To Exception Handler

CtrlXferItem n				$\mathrm{MR}_{c_0+\ldots+c_n+1}$
⋮			⋮	
CtrlXferItem 0				MR_{c_0+1}
exception error code $_{(32/64)}$				MR_2
exception number $_{(32/64)}$				MR_1
$0_{(16/48)}$	$0_{(4)}$	$t = \sum c_i\ _{(6)}$	$u = 2_{(6)}$	MR_0

***Acceptor* [BR₀]**

$0_{(22/54)}$	$s = 1_{(6)}$	0010	BR_0

The acceptor covers the complete user address space and accepts all control transfer items. The kernel accepts mappings or grants into this region on behalf of the faulting thread, and sets the thread state based upon the control transfer items enclosed in the reply message. The received message is discarded.

From Exception Handler

CtrlXferItem n				MR $_{c_0+\ldots+c_n+3}$
⋮				
CtrlXferItem 0				MR $_{c_0+3}$
MapItem / GrantItem				MR $_{1,2}$
0 $_{(16/48)}$	0 $_{(4)}$	$t = 2 + \sum c_i$ $_{(6)}$	$u = 0$ $_{(6)}$	MR $_0$

Extended Preemption Protocol

To Scheduler

CtrlXferItem n				MR $_{c_0+\ldots+c_n+1}$
⋮				
CtrlXferItem 0				MR $_{c_0+1}$
Clock $/2^{(32/64)}$ $_{(32/64)}$				MR $_2$
Clock $\bmod 2^{(32/64)}$ $_{(32/64)}$				MR $_1$
0 $_{(16/48)}$	0 $_{(4)}$	$t = \sum c_i$ $_{(6)}$	$u = 2$ $_{(6)}$	MR $_0$

***Acceptor* [BR$_0$]**

0 $_{(22/54)}$	$s = 1$ $_{(6)}$	0010	BR $_0$

The acceptor covers the complete user address space and accepts all control transfer items. The kernel accepts mappings or grants into this region on behalf of the faulting thread, and sets the thread state based upon the control transfer items enclosed in the reply message. The received message is discarded.

From Scheduler

CtrlXferItem n				MR $_{c_0+\ldots+c_n+3}$
⋮				
CtrlXferItem 0				MR $_{c_0+3}$
MapItem / GrantItem				MR $_{1,2}$
0 $_{(16/48)}$	0 $_{(4)}$	$t = 2 + \sum c_i$ $_{(6)}$	$u = 0$ $_{(6)}$	MR $_0$

7.7 Sigma0 RPC protocol [Protocol]

σ_0 is the initial address space. Although it is *not* part of the kernel, its basic protocol is defined with the kernel. Specific σ_0 implementations may extend this protocol.

The address space σ_0 is idempotent, i.e., all virtual addresses in this address space are identical to the corresponding physical address. Note that pages requested from σ_0 continue to be mapped idempotently if the receiver specifies its complete address space as receive fpage.

σ_0 gives pages to the kernel and to arbitrary tasks, but only once. The idea is that all pagers request the memory they need in the startup phase of the system so that afterwards σ_0 has exhausted all its memory. Further requests will then automatically be denied.

Kernel Protocol

To σ_0

$\sim$ $_{(32/64)}$					MR $_2$
requested fpage $_{(32/64)}$					MR $_1$
-6 $_{(12/44)}$	0 $_{(4)}$	0 $_{(4)}$	$t = 0$ $_{(6)}$	$u = 2$ $_{(6)}$	MR $_0$

requested fpage

-1 $_{(22/54)}$	s $_{(6)}$	$0\,r\,w\,x$

$s = 0$ Kernel requests the amount of memory recommended by σ_0 for kernel use (pagetable and other kernel-internal data).

$s \neq 0$ Kernel requests an fpage of size 2^s. The fpage can be located at an arbitrary position but must contain ordinary memory. If a free fpage of size 2^s is available, it is *granted* to the kernel.

rwx The rwx bits are ignored. σ_0 always grants fpages with maximum access rights to the kernel.

From σ_0

Kernel memory recommendation

0 $_{(32/64)}$				MR $_2$
amount $_{(32/64)}$				MR $_1$
0 $_{(16/48)}$	0 $_{(4)}$	$t = 0$ $_{(6)}$	$u = 2$ $_{(6)}$	MR $_0$

amount Amount of memory recommended for kernel use (in bytes).

Grant Response

GrantItem				MR $_{1,2}$
0 $_{(16/48)}$	0 $_{(4)}$	$t = 2$ $_{(6)}$	$u = 0$ $_{(6)}$	MR $_0$

Grant Reject

nilpage $_{(32/64)}$				MR $_2$
$0_{\ (28/60)}$			1010	MR $_1$
$0_{\ (16/48)}$	$0_{\ (4)}$	$t = 2_{\ (6)}$	$u = 0_{\ (6)}$	MR $_0$

User Protocol

To σ_0

high address $_{(32/64)}$					MR $_3$
requested attributes $_{(32/64)}$					MR $_2$
requested fpage $_{(32/64)}$					MR $_1$
$-6_{\ (12/44)}$	$0_{\ (4)}$	$0_{\ (4)}$	$t = 0_{\ (6)}$	$u = 2_{\ (6)}$	MR $_0$

requested fpage

$b/2^{10}_{\ (22/54)}$	s $_{(6)}$	e	$r\,w\,x$

σ_0 deals with fpages of arbitrary size. A successful response from σ_0 contains an fpage of physically contiguous memory.

$b \neq -1$ Requests the specific fpage with base address b and size 2^s. If the fpage is neither owned by the kernel nor by a user thread (not even partially), the requested fpage is mapped to the requestor's address space and the fpage is marked as owned by the requesting thread (i.e., fpage is *not* marked as being owned by the address space in which thread resides). Any fpage not belonging to *reserved memory* (see page 93) can be requested. If the requested fpage is already owned by the requestor only the page attributes are modified. No new mapping operation happens.

$b = -1$ Requests an fpage of size 2^s but with arbitrary address. If a free fpage of size 2^s is available, it is mapped to the requestor's address space and marked as owned by the requesting thread (i.e., fpage is *not* marked as being owned by the address space in which thread resides). σ_0 is free to use the *requested-attribute* for choosing a best fitting page. Only fpages belonging to *conventional memory* (see page 93) are considered free and handed out upon such anonymous requests.

e Setting this bit to 1 instructs σ_0 to map an address longer than the usual address size of the system (e.g. a 64 bit address on a 32 bit system). In this case, the lowermost bits of the requested address are specified as ususal in the *requested fpage* field, while the highermost bits are specified in a separate message register (see *high address* below).

rwx The rwx bits are ignored. σ_0 always maps fpages with maximum access rights to the requestor.

requested attributes

= 0 The page is requested with default attributes.

≠ 0 The page is requested with some architecture dependent attributes.

high address this field contains the part of the requested address that didn't fit in the *requested fpage* field. σ_0 concatenates this field with the base address of the *requested fpage* field and then tries to map the result into the requesters address space. Note that this field will not be included in the response's *MapItem*.
This field is only read if the e bit is set to 1. If the e bit is 0, this field is ignored.

From σ_0

Map Response

MapItem				MR $_{1,2}$
0 $_{(16/48)}$	0 $_{(4)}$	$t = 2$ $_{(6)}$	$u = 0$ $_{(6)}$	MR $_0$

Map Reject

nilpage $_{(32/64)}$				MR $_2$
0 $_{(28/60)}$			1000	MR $_1$
0 $_{(16/48)}$	0 $_{(4)}$	$t = 2$ $_{(6)}$	$u = 0$ $_{(6)}$	MR $_0$

σ_0 responds with a *map reject* message if the page is reserved (i.e., kernel space) or already mapped to a different thread, or if memory is exhausted.

Pagefault Protocol

σ_0 also understands the pagefault protocol (see page 82) and will convert pagefault requests into σ_0 user protocol requests. Further, only memory marked as *conventional memory* (see page 93) can be requested using the pagefault protocol. Any non-conventional memory (including boot loader specific memory) must be requested explicitly using the regular σ_0 protocol.

Incoming pagefault message

faulting user-level IP $_{(32/64)}$					MR $_2$
fault address $_{(32/64)}$					MR $_1$
-2 $_{(12/44)}$	$0\,r\,w\,x$	0 $_{(4)}$	$t = 0$ $_{(6)}$	$u = 2$ $_{(6)}$	MR $_0$

Converted pagefault message

0 $_{(32/64)}$					MR $_2$
fault address$/2^{10}$ $_{(22/54)}$		s $_{(6)}$		0000	MR $_1$
-6 $_{(12/44)}$	0 $_{(4)}$	0 $_{(4)}$	$t = 0$ $_{(6)}$	$u = 2$ $_{(6)}$	MR $_0$

s The minimum supported page size as defined by the PageInfo field in the kernel interface page (see page 3).

7.8 Generic Booting [Protocol]

Machine-specific boot procedures are described on pages 112 ff.

After booting, L4 initializes itself. It generates the basic address space-servers σ_0, σ_1 and a *root server* which is intended to boot the higher-level system.

σ_0, σ_1 and the *root server* are user-level servers and not part of the pure kernel. The predefined ones can be replaced by modifying the following table in the L4 image before starting L4. An empty area specifies that the corresponding server should not be started. Note, that σ_0 is a mandatory service. 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
σ_1.high	σ_1.low	σ_1.IP	σ_1.SP	+30 / +60
σ_0.high	σ_0.low	σ_0.IP	σ_0.SP	+20 / +40
Kdebug.high	Kdebug.low	Kdebug.entry	Kdebug.init	+10 / +20
~		API Version	~$(0/32)$ '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

<table>
<tr><td colspan="1">$high/2^{10}$ $_{(22/54)}$</td><td colspan="4">$\sim$ $_{(10)}$</td><td>+4 / +8</td></tr>
<tr><td>$low/2^{10}$ $_{(22/54)}$</td><td>v</td><td>$\sim$</td><td>t $_{(4)}$</td><td>$type$ $_{(4)}$</td><td>+0</td></tr>
</table>

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., memory not available to user)
0x4	Shared memory (i.e., available to all users)
0xE	Defined by boot loader
0xF	Architecture dependent

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

$type = 0xE$

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 (see page **??**).

type ≠ 0xE, type ≠ 0xF

The type of the memory descriptor is solely defined by the $type$ field. The content of the t field is undefined.

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. ThreadWord0 and ThreadWord1 are free to be used by systems software (e.g., IDL compilers). The kernel associates no semantics with these words.

$\sim_{(32)}$			⟵ UTCB address
⋮		⋮	
ThreadWord 0 $_{(32)}$			–16
ThreadWord 1 $_{(32)}$			–20
VirtualSender/ActualSender $_{(32)}$			–24
IntendedReceiver $_{(32)}$			–28
XferTimeouts $_{(32)}$			–32
ErrorCode $_{(32)}$			–36
$\sim_{(16)}$	cop flags $_{(8)}$	preempt flags $_{(8)}$	–40
ExceptionHandler $_{(32)}$			–44
Pager $_{(32)}$			–48
UserDefinedHandle $_{(32)}$			–52
ProcessorNo $_{(32)}$			–56
MyGlobalId $_{(32)}$			–60

MyLocalId = UTCB address $_{(32)}$	gs:[0]

The TCR *MyLocalId* is not part of the UTCB. On ia32 it is identical with the UTCB address and can be loaded from memory location gs:[0].

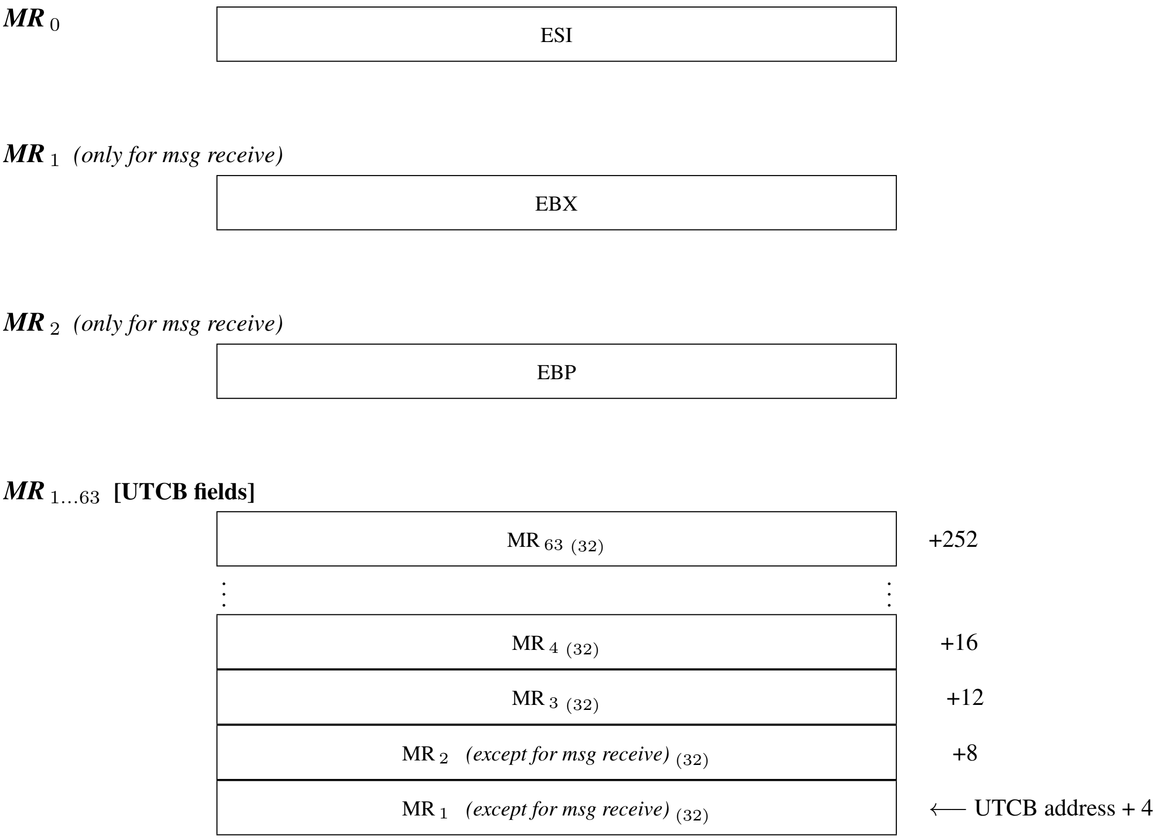
Message Registers (MRs)

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 $_{3...63}$ are always mapped to memory.


MR 0
ESI
MR 1 (only for msg receive)
EBX
MR 2 (only for msg receive)
EBP
MR 1...63 [UTCB fields]
MR 63 (32)
+252
MR 4 (32)
+16
MR 3 (32)
+12
MR 2 (except for msg receive) (32)
+8
MR 1 (except for msg receive) (32)
⟵ UTCB address + 4

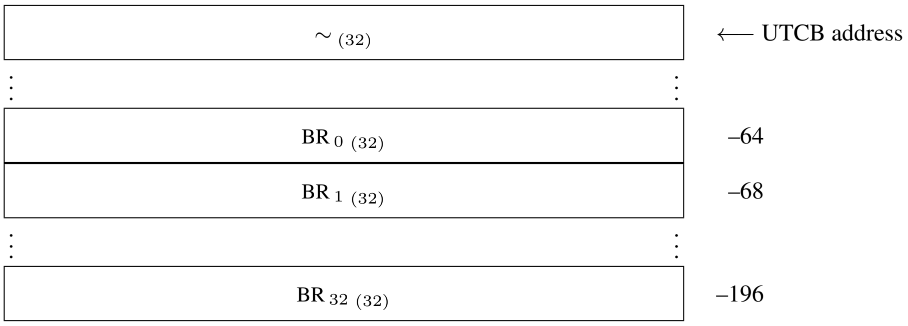
Buffer Registers (BRs)

BRs 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.

BR $_{0\ldots32}$ **[UTCB fields]**



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.

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	∼
UtcbLocation	EDI		EDI	∼
–	EBX		EBX	∼
–	EBP		EBP	∼
–	ESP		ESP	≡

SystemClock [Systemcall]

–	EAX	– **SystemClock** →	EAX	*clock 0...31*
–	ECX		ECX	∼
–	EDX		EDX	*clock 32...63*
–	ESI	call *SystemClock*	ESI	∼
–	EDI		EDI	∼
–	EBX		EBX	≡
–	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*
prio	ECX		ECX	∼
time control	EDX		EDX	*time control*
processor control	ESI	call *Schedule*	ESI	∼
preemption control	EDI		EDI	∼
–	EBX		EBX	∼
–	EBP		EBP	∼
–	ESP		ESP	≡

Ipc [Systemcall]

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

Lipc [Systemcall]

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

Unmap [Systemcall]

control	EAX	– **Unmap** →	EAX	∼
–	ECX		ECX	∼
–	EDX		EDX	∼
MR_0	ESI	call *Unmap*	ESI	MR_0
UTCB	EDI		EDI	≡
–	EBX		EBX	∼
–	EBP		EBP	∼
–	ESP		ESP	≡

SpaceControl **[Privileged Systemcall]**

SpaceSpecifier	EAX	– **Space Control** →	EAX	*result*
control	ECX		ECX	*control*
KernelInterfacePageArea	EDX		EDX	∼
UtcbArea	ESI	call *SpaceControl*	ESI	∼
Redirector	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	≡

MemoryControl **[Privileged Systemcall]**

control	EAX	– **Memory Control** →	EAX	*result*
$attribute_0$	ECX		ECX	∼
$attribute_1$	EDX		EDX	∼
MR_0	ESI	call *MemoryControl*	ESI	∼
UTCB	EDI		EDI	∼
$attribute_2$	EBX		EBX	∼
$attribute_3$	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]

IO Fpages

On IA-32 processors, IO-ports are handled as fpages. IO fpages can be mapped, granted, 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 $_{(16)}$	s' $_{(6)}$	$s = 2$ $_{(6)}$	0110

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. There are no distinct rights associated with IO ports, i.e., a task can be granted either read- and write-access to an IO port, ore none at all.

IO Pagefault Protocol

A thread generating an IO port exception will cause the kernel to transparently generate an IO-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

faulting user-level IP $_{(32)}$	MR $_2$

faulting port $_{(16)}$	size $_{(6)}$	$s = 2$ $_{(6)}$	0110	MR $_1$

-8 $_{(12)}$	0110	0 $_{(4)}$	$t = 0$ $_{(6)}$	$u = 2$ $_{(6)}$	MR $_0$

***Acceptor* [BR$_0$]**

0 $_{(16)}$	16 $_{(6)}$	$s = 2$ $_{(6)}$	0000	BR $_0$

The acceptor covers the complete IO-address space. The kernel accepts mappings or grants into this region on behalf of the faulting thread. The received message is discarded.

Generic Programming Interface

#include <l4/arch.h>

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

Fpage ***IoFpageLog2*** (*Word BasePort, int Log2FpageSize* <= *16*)

Delivers an IO fpage with the specified location and size.

Word ***IoFpagePort*** (*Fpage f*)

Word ***IoFpageSize*** (*Fpage f*)

Word ***IoFpageSizeLog2*** (*Fpage f*)

Delivers port/size of specified IO fpage.

Bool ***IsIoFpage*** (*Fpage f*)

Delivers true if fpage is an IO fpage.

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 Parameters

control

s	0	t	0 $_{(21)}$	small $_{(8)}$

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 $s = 0$.

t A value of 1 instructs the kernel to add an entry to the translation table for extended mappings. This table allows mapping of memory addresses longer than 32 bits on 32-bit systems. The desired mapping is specified in the remaining parameters of the SpaceControl system call as follows: The redirector field must contain the highest 32 bits of the desired address, while the utcb_area field must contain the lower 32 bits. The kip_area field contains a regular fpage, which specifies a region of 32 bit addresses that should be mapped to a 64 bit address. If any address in this fpage is mapped to a thread, the address will be translated to the corresponding 64 bit address. If the mapping is successfull, the translation table entry is deleted.

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: $size = 2^{pos} * 4\,\text{MB}$. 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: $location = small * 2\,\text{MB}$. 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 102).

Output Parameter

control

e	0	t	0 $_{(21)}$	small $_{(8)}$

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

t Indicates if an entry was successfully added to the kernel's translation table for extended mappings.

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 Cacheability Hints [ia32]

String items can specify cacheability hints to the kernel (see page 59). For ia32, the cacheability hints have the following semantics.

$hh = 00$ Use the processor's default cacheability strategy. Typically, cache lines are allocated for data read and written (assuming that the processor's default strategy is write-back and write-allocate).

$hh = 01$ Allocate cache lines in the entire cache hierarchy for data read or written.

$hh = 10$ Do not allocate new cache lines (entire cache hierarchy) for data read or written.

$hh = 11$ Allocate only new L1 cache line for data read or written. Do not allocate cache lines in lower cache hierarchies.

Convenience Programming Interface

#include <l4/ipc.h>

CacheAllocationHint ***UseDefaultCacheLineAllocation***

CacheAllocationHint ***AllocateNewCacheLines***

CacheAllocationHint ***DoNotAllocateNewCacheLines***

CacheAllocationHint ***AllocateOnlyNewL1CacheLines***

A.7 Memory Attributes [ia32]

The ia32 architecture in general supports the following memory attributes values.

attribute	value
Default	0
Write Back	1
Write Through	2
Uncacheable	4
Write Combining	5
Write Protected	8

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 ***WriteBackMemory***

Word ***WriteThroughMemory***

Word ***UncacheableMemory***

Word ***WriteCombiningMemory***

Word ***WriteProtectedMemory***

A.8 Exception Message Format [ia32]

To Exception Handler

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

#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.9 IA-32 Control Transfer Items [ia32]

***General Purpose Register CtrlXferItem** ($id = 0$)*

EAX $_{(32)}$			MR $_{i+9}$
ECX $_{(32)}$			MR $_{i+8}$
EDX $_{(32)}$			MR $_{i+7}$
EBX $_{(32)}$			MR $_{i+6}$
ESP $_{(32)}$			MR $_{i+5}$
EBP $_{(32)}$			MR $_{i+4}$
ESI $_{(32)}$			MR $_{i+3}$
EFLAGS $_{(32)}$			MR $_{i+2}$
EIP $_{(32)}$			MR $_{i+1}$
0x3FF $_{(20)}$	1 $_{(8)}$	$1\,1\,0\,C$	MR $_i$

***Floating Point Register CtrlXferItem** ($id = 1$)*

FPU register 128/512 $_{(32)}$			FPU $_{256/512}$
⋮		⋮	
FPU register 0 $_{(32)}$			FPU $_0$
$\sim$ $_{(20)}$	2 $_{(8)}$	$1\,1\,0\,C$	MR $_i$

Convenience Programming Interface

#include <l4/ia32/arch.h>

struct **GPRegsCtrlXferItem** { Word raw [11] }

struct **FPURegsCtrlXferItem** { Word raw [1] } *void* ***Append*** (*Msg& msg, GPRegsCtrlXferItem c*) [*MsgAppendGPRegsCtrlXferItem*]

void ***Append*** (*Msg& msg, FPURegsCtrlXferItem c*) [*MsgAppendFPURegsCtrlXferItem*]

A.10 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.11 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	$\sim$	$*P$
$\langle P + 0 \rangle$		$\sim$ OR 1
$\langle P + 4 \rangle$	n/a	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.

A.12 Support for Hardware-assisted Virtualization [ia32]

In addition to its normal execution mode, L4 provides support for virtualization mode. Virtualization mode is largely common to L4's normal execution model. However, in virtualization mode, threads have access to an extended ISA, and have restricted access to L4-specific features.

Hardware virtualization mode (HVM) is based on IA-32 virtualization hardware extensions: Intel VT-x or AMD Pacifica. Threads that execute in that mode have access to an extended architecture that includes the entire privileged instruction set (ideally, within the limits of the hardware facilities). Such a thread can be seen as a virtual CPU, which contains all of the state held by a physical CPU. In addition to the "normal" page faults and exceptions already handled by L4, HVM threads generate virtualization faults on all events that would be observable by the hardware connected to a physical CPU (and some events that would be internal to a physical CPU).

The virtualization extensions introduce new kernel feature strings:

String	Feature
"x86-vmx"	Kernel has full virtualization support using Intel's VT-x.
"x86-svm"	Kernel has full virtualization support using AMD's Pacifica.

Extended Thread State

An thread inside a HVM space represents a virtualized physical processor for the virtualization HVM space. It holds all privileged and unprivileged registers of the physical processor. VM-exits cause virtualization fault messages to efficiently manage critical instructions. Virtualization fault replies allow mapping memory into the HVM space and protocol items allow read/write access to the VCPU state. EXCHANGEREGISTERS grants asynchronous access by forcing virtualization faults.

Address Space

In hardware virtualization mode, the L4 execution and resource model is mapped onto a *physical* machine model. A thread that executes in HVM has access to the privileged part of the platform architecture and runs with an additional memory translation. Depending on the hardware support for double paging, L4 either utilizes the hardware features or provides a transparent translation of guest-virtual-to-host-physical translations, based on the guest's virtual to physical, and the host's virtual to physical mappings.

SPACECONTROL

The SPACECONTROL system call has an architecture dependent *control* parameter to specify various address space characteristics. For IA-32, the *control* parameter has the following semantics.

Input Parameters

control

s	v	t	$0_{(21)}$	small $_{(8)}$

v The v field denotes the virtualization mode for all threads in the address space. The v field can only be specified for inactive address spaces and is ignored for active address spaces. The availability of the virtualization features is announced as a KIP feature string.

v=0 An address space with no virtualization support.

v=1 *Hardware virtualization mode* is the hardware assisted virtualization support for IA-32, either Intel's VT-x or AMD's Pacifica. In hardware virtualization mode, the complete address space is empty and under control of the pager thread. The thread's state is extended by IA-32 processor state including control registers, all segment selectors, debugging registers, etc.

Output Parameters

control

e	v	t	$0_{(21)}$	small $_{(8)}$

v Indicates if enabling the requested virtualization mode has succeeded ($v = 1$). Zero if $v = 0$ in the input parameter.

IA-32 HVM Control Transfer Items

***Control Register CtrlXferItem** ($id = 2$)*

CR4 Read Shadow $_{(32)}$			MR $_{i+8}$
CR4 Guest/Host Mask $_{(32)}$			MR $_{i+7}$
CR4 $_{(32)}$			MR $_{i+6}$
CR3 $_{(32)}$			MR $_{i+5}$
CR2 $_{(32)}$			MR $_{i+4}$
CR0 Guest/Host Mask $_{(32)}$			MR $_{i+3}$
CR0 Read Shadow $_{(32)}$			MR $_{i+2}$
CR0 $_{(32)}$			MR $_{i+1}$
0x7F $_{(20)}$	4 $_{(8)}$	$1\,1\,0\,C$	MR $_{i}$

***Debug Register CtrlXferItem** ($id = 3$)*

DR7 $_{(32)}$			MR $_{i+6}$
DR6 $_{(32)}$			MR $_{i+5}$
DR3 $_{(32)}$			MR $_{i+4}$
DR2 $_{(32)}$			MR $_{i+3}$
DR1 $_{(32)}$			MR $_{i+2}$
DR0 $_{(32)}$			MR $_{i+1}$
0x3F $_{(20)}$	5 $_{(8)}$	$1\,1\,0\,C$	MR $_{i}$

***Code Segment Register CtrlXferItem** ($id = 4$)*

CS_ATTR $_{(32)}$			MR $_{i+4}$
CS_LIMIT $_{(32)}$			MR $_{i+3}$
CS_BASE $_{(32)}$			MR $_{i+2}$
CS $_{(32)}$			MR $_{i+1}$
0xF $_{(20)}$	6 $_{(8)}$	$1\,1\,0\,C$	MR $_{i}$

***Stack Segment Register CtrlXferItem** ($id = 5$)*

SS_ATTR $_{(32)}$			MR $_{i+4}$
SS_LIMIT $_{(32)}$			MR $_{i+3}$
SS_BASE $_{(32)}$			MR $_{i+2}$
SS $_{(32)}$			MR $_{i+1}$
0xF $_{(20)}$	7 $_{(8)}$	$110C$	MR $_{i}$

***Data Segment Register CtrlXferItem** $(id = 6)$*

DS_ATTR $_{(32)}$			MR $_{i+4}$
DS_LIMIT $_{(32)}$			MR $_{i+3}$
DS_BASE $_{(32)}$			MR $_{i+2}$
DS $_{(32)}$			MR $_{i+1}$
0xF $_{(20)}$	8 $_{(8)}$	$110C$	MR $_{i}$

***Extra Segment Register CtrlXferItem** $(id = 7)$*

ES_ATTR $_{(32)}$			MR $_{i+4}$
ES_LIMIT $_{(32)}$			MR $_{i+3}$
ES_BASE $_{(32)}$			MR $_{i+2}$
ES $_{(32)}$			MR $_{i+1}$
0xF $_{(20)}$	9 $_{(8)}$	$110C$	MR $_{i}$

***F-Segment Register CtrlXferItem** $(id = 8)$*

FS_ATTR $_{(32)}$			MR $_{i+4}$
FS_LIMIT $_{(32)}$			MR $_{i+3}$
FS_BASE $_{(32)}$			MR $_{i+2}$
FS $_{(32)}$			MR $_{i+1}$
0xF $_{(20)}$	10 $_{(8)}$	$110C$	MR $_{i}$

***G-Segment Register CtrlXferItem** $(id = 9)$*

GS_ATTR $_{(32)}$			MR $_{i+4}$
GS_LIMIT $_{(32)}$			MR $_{i+3}$
GS_BASE $_{(32)}$			MR $_{i+2}$
GS $_{(32)}$			MR $_{i+1}$
0xF $_{(20)}$	11 $_{(8)}$	$1\,1\,0\,C$	MR $_{i}$

***Task Register CtrlXferItem* $(id = 10)$**

TR_ATTR $_{(32)}$			MR $_{i+4}$
TR_LIMIT $_{(32)}$			MR $_{i+3}$
TR_BASE $_{(32)}$			MR $_{i+2}$
TR $_{(32)}$			MR $_{i+1}$
0xF $_{(20)}$	12 $_{(8)}$	$1\,1\,0\,C$	MR $_{i}$

***Local Descriptor Register CtrlXferItem* $(id = 11)$**

LDTR_ATTR $_{(32)}$			MR $_{i+4}$
LDTR_LIMIT $_{(32)}$			MR $_{i+3}$
LDTR_BASE $_{(32)}$			MR $_{i+2}$
LDTR $_{(32)}$			MR $_{i+1}$
0xF $_{(20)}$	13 $_{(8)}$	$1\,1\,0\,C$	MR $_{i}$

***Interrupt Descriptor Register CtrlXferItem* $(id = 12)$**

IDTR_ATTR $_{(32)}$			MR $_{i+3}$
IDTR_LIMIT $_{(32)}$			MR $_{i+2}$
IDTR_BASE $_{(32)}$			MR $_{i+1}$
0x7 $_{(20)}$	14 $_{(8)}$	$1\,1\,0\,C$	MR $_{i}$

***Global Descriptor Register CtrlXferItem* $(id = 13)$**

GDTR_ATTR $_{(32)}$			MR $_{i+3}$
GDTR_LIMIT $_{(32)}$			MR $_{i+2}$
GDTR_BASE $_{(32)}$			MR $_{i+1}$
0x7 $_{(20)}$	15 $_{(8)}$	$1\,1\,0\,C$	MR $_{i}$

***Guest Non-Reg and Exception State CtrlXferItem** ($id = 14$)*

IDT_EEC $_{(32)}$			MR $_{i+9}$
IDT_INFO $_{(32)}$			MR $_{i+8}$
EXIT_EEC $_{(32)}$			MR $_{i+7}$
EXIT_INFO $_{(32)}$			MR $_{i+6}$
ENTRY_ILEN $_{(32)}$			MR $_{i+5}$
ENTRY_EEC $_{(32)}$			MR $_{i+4}$
ENTRY_INFO $_{(32)}$			MR $_{i+3}$
PENDING_DEBUG_EXC $_{(32)}$			MR $_{i+3}$
INTERRUPTIBILITY_STATE $_{(32)}$			MR $_{i+2}$
ACTIVITY_STATE $_{(32)}$			MR $_{i+1}$
0x3FF $_{(20)}$	16 $_{(8)}$	$1\,1\,0\,C$	MR $_{i}$

***Guest Execution Control CtrlXferItem** ($id = 15$)*

EXC_BITMAP $_{(32)}$			MR $_{i+3}$
CPU_EXEC_CTRL $_{(32)}$			MR $_{i+2}$
PIN_EXEC_CTRL $_{(32)}$			MR $_{i+1}$
0x7f $_{(20)}$	17 $_{(8)}$	$1\,1\,0\,C$	MR $_{i}$

***Other Guest State CtrlXferItem** ($id = 16$)*

TPR_THRESHOLD $_{(32)}$			MR $_{i+9}$
VAPIC_ADDRESS $_{(32)}$			MR $_{i+8}$
RDTSC_OFS_HIGH $_{(32)}$			MR $_{i+7}$
RDTSC_OFS_LOW $_{(32)}$			MR $_{i+6}$
DEBUGCTL_MSR_HIGH $_{(32)}$			MR $_{i+5}$
DEBUGCTL_MSR_LOW $_{(32)}$			MR $_{i+4}$
SYSENTER_ESP_MSR $_{(32)}$			MR $_{i+3}$
SYSENTER_EIP_MSR $_{(32)}$			MR $_{i+2}$
SYSENTER_CS_MSR $_{(32)}$			MR $_{i+1}$
0x1FF $_{(20)}$	18 $_{(8)}$	$1\,1\,0\,C$	MR $_{i}$

Virtualization Fault Protocol

The virtualization protocol is defined between a VCPU thread and its registered pager thread. It substitutes the page fault and exception protocol used for normal threads. Virtualization fault messages are sent to the pager on events related to virtualization that are not handled directly by the hardware or by the L4 microkernel. By default, the kernel will append the fault-specific state specified below when sending kernel messages. Like with the normal fault protocols (see Section 7, the kernel will append additional control transfer items upon requests. Requests to add or remove control transfer items protocol are performed using the EXCHANGEREGISTERS system call and appropriate control transfer configuration items (see Section 2.3).

Virtualization Fault

From Pager:

guest address / instruction info $_{(32)}$					MR $_3$
instruction length $_{(32)}$					MR $_2$
fault qualification $_{(32)}$					MR $_1$
$-9 - faultID$ $_{(16)}$	0 $_{(4)}$	0 $_{(4)}$	$t = 0$ $_{(6)}$	$u = 3$ $_{(6)}$	MR $_0$

fault ID	Implementation-specific fault identifiier. For Intel VT-x, the identifier corresponds to the VM exit reason.
fault qualification	Additional information about the cause of exits.
instruction length	The length of the faulting instruction.
operand info	Guest linear address / Additional information about the faulting instruction .
value	For a read fault, the virtual value of the faulting register, if the register is part of the VCPU state.

Virtualization Fault Reply

From Pager:

CtrlXferItem n				MR $_{c_0+\ldots+c_n+3}$
⋮				
CtrlXferItem 0				MR $_{c_0+3}$
MapItem / GrantItem				MR $_{1,2}$
0 $_{(148)}$	0 $_{(4)}$	$t = 2 + \sum c_i$ $_{(6)}$	$u = 0$ $_{(6)}$	MR $_0$

A.13 MSR-Fpage

Access to processor's model specific registers is controlled via fpages. The minimal granularity is 1. An MSR-fpage of size $2^{s'}$ has a 2^s-aligned offset address $sndbase + offset$, i.e $offset$ mod 2^s=0.

control

offset $_{(16)}$	s' $_{(6)}$	$s = 3$	$v\,r\,w\,x$

r Allow read access to the specified MSRs.

w Allow write access to the specified MSRs.

g Ignored for mappings into non-HVM spaces. For mappings into HVM space $v = 0$ grants access to the system MSR. On $v = 0$ the kernel installs a VCPU local MSRs which is transparently multiplexed.

s' $2^{s'}$ is the size of the region.

offset Offset specifies the lowest 16 bits of a MSR base address.

Appendix B

AMD64 Interface

B.1 Virtual Registers [amd64]

Thread Control Registers (TCRs)

TCRs are implemented as part of the amd64-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. ThreadWord0 and ThreadWord1 are free to be used by systems software (e.g., IDL compilers). The kernel associates no semantics with these words.

<table>
<tr><td colspan="3">ThreadWord 0 $_{(64)}$</td><td>⟵ UTCB address – 32</td></tr>
<tr><td colspan="3">ThreadWord 1 $_{(64)}$</td><td>– 40</td></tr>
<tr><td colspan="3">VirtualSender/ActualSender $_{(64)}$</td><td>– 48</td></tr>
<tr><td colspan="3">IntendedReceiver $_{(64)}$</td><td>– 56</td></tr>
<tr><td colspan="3">XferTimeouts $_{(64)}$</td><td>– 64</td></tr>
<tr><td colspan="3">ErrorCode $_{(64)}$</td><td>– 72</td></tr>
<tr><td>~ $_{(48)}$</td><td>cop flags $_{(8)}$</td><td>preempt flags $_{(8)}$</td><td>– 80</td></tr>
<tr><td colspan="3">ExceptionHandler $_{(64)}$</td><td>– 88</td></tr>
<tr><td colspan="3">Pager $_{(64)}$</td><td>– 96</td></tr>
<tr><td colspan="3">UserDefinedHandle $_{(64)}$</td><td>–104</td></tr>
<tr><td colspan="3">ProcessorNo $_{(64)}$</td><td>–112</td></tr>
<tr><td colspan="3">MyGlobalId $_{(64)}$</td><td>–120</td></tr>
</table>

MyLocalId = UTCB address $_{(64)}$	gs:[0]

The TCR *MyLocalId* is not part of the UTCB. On amd64 it is identical with the UTCB address and can be loaded from memory location gs:[0].

Message Registers (MRs)

Memory-mapped MRs are implemented as part of the amd64-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.

The first 8 message registers MR $_0$ through MR $_7$ are always mapped to general register. MR $_{8...63}$ are always mapped to memory.

MR $_{0...7}$

MR $_7$	R15
MR $_6$	R14
MR $_5$	R13
MR $_4$	R12
MR $_3$	R10
MR $_2$	RBX
MR $_1$	RAX
MR $_0$	R09

MR $_{8...63}$ **[UTCB fields]**

MR $_{63\ (64)}$	+ 504
⋮	
MR $_{10\ (64)}$	+ 80
MR $_{9\ (64)}$	+ 72
MR $_{8\ (64)}$	⟵ UTCB address + 64

Buffer Registers (BRs)

BRs are implemented as part of the amd64-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.

BR $_{0...32}$ **[UTCB fields]**

BR$_{0\ (64)}$	←— UTCB address –128
BR$_{1\ (64)}$	–136
⋮	
BR$_{32\ (64)}$	–384

B.2 Systemcalls [amd64]

The system-calls which are invoked by the call instruction take the target of the calls the from system-call link fields in the kernel interface page (see page 2). Each system-call link specifies an absolute 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]

–	RAX	– **KernelInterface** →	RAX	*base address*
–	RCX		RCX	*API Version*
–	RDX		RDX	*API Flags*
–	RSI	lock: nop	RSI	*Kernel ID*
–	RDI		RDI	≡
–	RBX		RBX	≡
–	RBP		RBP	≡
–	R08		R08	≡
–	R09		R09	≡
–	R10		R10	≡
–	R11		R11	≡
–	R12		R12	≡
–	R13		R13	≡
–	R14		R14	≡
–	R15		R15	≡
–	RSP		RSP	≡

EXCHANGEREGISTERS [Systemcall]

dest	RAX	– **Exchange Registers** →	RAX	*result*
–	RCX		RCX	∼
SP	RDX		RDX	*SP*
control	RSI	call *ExchangeRegisters*	RSI	*control*
pager	RDI		RDI	*pager*
–	RBX		RBX	∼
–	RBP		RBP	∼
IP	R08		R08	*IP*
FLAGS	R09		R09	*FLAGS*
UserDefinedHandle	R10		R10	*UserDefinedHandle*
–	R11		R11	∼
–	R12		R12	∼
–	R13		R13	∼
–	R14		R14	∼
–	R15		R15	∼
–	RSP		RSP	∼

"FLAGS" refers to the user-modifiable amd64 processor flags that are held in the RFLAGS register.

ThreadControl [Privileged Systemcall]

–	RAX	– **Thread Control** →	RAX	*result*
–	RCX		RCX	∼
scheduler	RDX		RDX	∼
pager	RSI	call *ThreadControl*	RSI	∼
dest	RDI		RDI	∼
–	RBX		RBX	∼
–	RBP		RBP	∼
SpaceSpecifier	R08		R08	∼
UTCBLocation	R09		R09	∼
–	R10		R10	∼
–	R11		R11	∼
–	R12		R12	∼
–	R13		R13	∼
–	R14		R14	∼
–	R15		R15	∼
–	RSP		RSP	∼

SystemClock [Systemcall]

–	RAX	– **SystemClock** →	RAX	*clock*
–	RCX		RCX	∼
–	RDX		RDX	∼
–	RSI	call *SystemClock*	RSI	∼
–	RDI		RDI	∼
–	RBX		RBX	∼
–	RBP		RBP	∼
–	R08		R08	∼
–	R09		R09	∼
–	R10		R10	∼
–	R11		R11	∼
–	R12		R12	∼
–	R13		R13	∼
–	R14		R14	∼
–	R15		R15	∼
–	RSP		RSP	∼

ThreadSwitch [Systemcall]

–	RAX	– **ThreadSwitch** →	RAX	∼
–	RCX		RCX	∼
–	RDX		RDX	∼
–	RSI	call *ThreadSwitch*	RSI	∼
dest	RDI		RDI	∼
–	RBX		RBX	∼
–	RBP		RBP	∼
–	R08		R08	∼
–	R09		R09	∼
–	R10		R10	∼
–	R11		R11	∼
–	R12		R12	∼
–	R13		R13	∼
–	R14		R14	∼
–	R15		R15	∼
–	RSP		RSP	∼

SCHEDULE [Systemcall]

–	RAX	– **Schedule** →	RAX	*result*
–	RCX		RCX	∼
time control	RDX		RDX	*time control*
prio	RSI	call *Schedule*	RSI	∼
dest	RDI		RDI	∼
–	RBX		RBX	∼
–	RBP		RBP	∼
processor control	R08		R08	∼
preemption control	R09		R09	∼
–	R10		R10	∼
–	R11		R11	∼
–	R12		R12	∼
–	R13		R13	∼
–	R14		R14	∼
–	R15		R15	∼
–	RSP		RSP	∼

IPC [Systemcall]

MR_1	RAX	– **Ipc** →	RAX	MR_1
–	RCX		RCX	∼
FromSpecifier	RDX		RDX	∼
to	RSI	call *Ipc*	RSI	*from*
UTCB	RDI		RDI	≡
MR_2	RBX		RBX	MR_2
–	RBP		RBP	∼
Timeouts	R08		R08	∼
MR_0	R09		R09	MR_0
MR_3	R10		R10	MR_3
–	R11		R11	∼
MR_4	R12		R12	MR_4
MR_5	R13		R13	MR_5
MR_6	R14		R14	MR_6
MR_7	R15		R15	MR_7
–	RSP		RSP	∼

LIPC [Systemcall]

MR_1	RAX	– **Lipc** →	RAX	MR_1
–	RCX		RCX	∼
FromSpecifier	RDX		RDX	∼
to	RSI	call *Lipc*	RSI	*from*
UTCB	RDI		RDI	≡
MR_2	RBX		RBX	MR_2
–	RBP		RBP	∼
Timeouts	R08		R08	∼
MR_0	R09		R09	MR_0
MR_3	R10		R10	MR_3
–	R11		R11	∼
MR_4	R12		R12	MR_4
MR_5	R13		R13	MR_5
MR_6	R14		R14	MR_6
MR_7	R15		R15	MR_7
–	RSP		RSP	∼

UNMAP **[Systemcall]**

MR_1	RAX	– **Unmap** →	RAX	MR_1
–	RCX		RCX	∼
control	RDX		RDX	∼
∼	RSI	call *Unmap*	RSI	∼
UTCB	RDI		RDI	≡
MR_2	RBX		RBX	MR_2
–	RBP		RBP	∼
–	R08		R08	∼
MR_0	R09		R09	MR_0
MR_3	R10		R10	MR_3
–	R11		R11	∼
MR_4	R12		R12	MR_4
MR_5	R13		R13	MR_5
MR_6	R14		R14	MR_6
MR_7	R15		R15	MR_7
–	RSP		RSP	∼

SPACECONTROL **[Privileged Systemcall]**

–	RAX	– **Space Control** →	RAX	*result*
–	RCX		RCX	∼
KernelInterfacePageArea	RDX		RDX	*control*
control	RSI	call *SpaceControl*	RSI	∼
SpaceSpecifier	RDI		RDI	∼
–	RBX		RBX	∼
–	RBP		RBP	∼
UTCBArea	R08		R08	∼
Redirector	R09		R09	∼
–	R10		R10	∼
–	R11		R11	∼
–	R12		R12	∼
–	R13		R13	∼
–	R14		R14	∼
–	R15		R15	∼
–	RSP		RSP	∼

PROCESSORCONTROL **[Privileged Systemcall]**

–	RAX	– **Processor Control** →	RAX	*result*
–	RCX		RCX	∼
ExternalFrequency	RDX		RDX	∼
InternalFrequency	RSI	call *ProcessorControl*	RSI	∼
ProcessorNo	RDI		RDI	∼
–	RBX		RBX	∼
–	RBP		RBP	∼
voltage	R08		R08	∼
–	R09		R09	∼
–	R10		R10	∼
–	R11		R11	∼
–	R12		R12	∼
–	R13		R13	∼
–	R14		R14	∼
–	R15		R15	∼
–	RSP		RSP	∼

MemoryControl [Privileged Systemcall]

MR_1	RAX	– **Memory Control** →	RAX	$\sim$
$attribute_0$	RCX		RCX	$\sim$
control	RDX		RDX	*result*
$attribute_1$	RSI	call *MemoryControl*	RSI	$\sim$
UTCB	RDI		RDI	$\equiv$
MR_2	RBX		RBX	$\sim$
–	RBP		RBP	$\sim$
$attribute_2$	R08		R08	$\sim$
MR_0	R09		R09	$\sim$
MR_3	R10		R10	$\sim$
$attribute_3$	R11		R11	$\sim$
MR_4	R12		R12	$\sim$
MR_5	R13		R13	$\sim$
MR_6	R14		R14	$\sim$
MR_7	R15		R15	$\sim$
–	RSP		RSP	$\sim$

B.3 IO Ports [amd64]

IO Fpages

On AMD64 processors, IO-ports are handled as fpages. IO fpages can be mapped, granted, 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 $_{(48)}$	s' $_{(6)}$	$s = 2$ $_{(6)}$	0 1 1 0

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. There are no distinct rights associated with IO ports, i.e., a task can be granted either read- and write-access to an IO port, ore none at all.

IO Pagefault Protocol

A thread generating an IO port exception will cause the kernel to transparently generate an IO-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

faulting user-level IP $_{(64)}$	MR $_2$

faulting port $_{(48)}$	size $_{(6)}$	$s = 2$ $_{(6)}$	0 1 1 0	MR $_1$

-8 $_{(44)}$	0 1 1 0	0 $_{(4)}$	$t = 0$ $_{(6)}$	$u = 2$ $_{(6)}$	MR $_0$

***Acceptor* [BR$_0$]**

0 $_{(48)}$	16 $_{(6)}$	$s = 2$ $_{(6)}$	0 0 0 0	BR $_0$

The acceptor covers the complete IO address space. The kernel accepts mappings or grants into this region on behalf of the faulting thread. The received message is discarded.

Generic Programming Interface

#include <l4/amd64/specials.h>

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

Fpage ***IoFpageLog2*** (*Word BaseAddress, int Log2FpageSize* $<=$ *16*)

Delivers an IO fpage with the specified location and size.

B.4 Cacheability Hints [amd64]

String items can specify cacheability hints to the kernel (see page 59). For amd64, the cacheability hints have the following semantics.

$hh = 00$ Use the processor's default cacheability strategy. Typically, cache lines are allocated for data read and written (assuming that the processor's default strategy is write-back and write-allocate).

$hh = 01$ Allocate cache lines in the entire cache hierarchy for data read or written.

$hh = 10$ Do not allocate new cache lines (entire cache hierarchy) for data read or written.

$hh = 11$ Allocate only new L1 cache line for data read or written. Do not allocate cache lines in lower cache hierarchies.

Convenience Programming Interface

#include <l4/ipc.h>

CacheAllocationHint ***UseDefaultCacheLineAllocation***

CacheAllocationHint ***AllocateNewCacheLines***

CacheAllocationHint ***DoNotAllocateNewCacheLines***

CacheAllocationHint ***AllocateOnlyNewL1CacheLines***

B.5 Memory Attributes [amd64]

The AMD64 architecture in general supports the following memory attributes values.

attribute	value
Default	0
Uncacheable	1
Write Combining	2
Write Through	5
Write Protected	6
Write Back	7

Note that some attributes are only supported on certain processors. See the "AMD64 Architecture Programmer's Manual Volume 2: System Programming" 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***

B.6 Exception Message Format [amd64]

To Exception Handler

<table>
<tr><td colspan="5">ErrorCode</td><td>MR $_{20}$</td></tr>
<tr><td colspan="5">ExceptionNo</td><td>MR $_{19}$</td></tr>
<tr><td colspan="5">RFLAGS</td><td>MR $_{18}$</td></tr>
<tr><td colspan="5">RSP</td><td>MR $_{17}$</td></tr>
<tr><td colspan="5">R11</td><td>MR $_{16}$</td></tr>
<tr><td colspan="5">R09</td><td>MR $_{15}$</td></tr>
<tr><td colspan="5">R08</td><td>MR $_{14}$</td></tr>
<tr><td colspan="5">RBP</td><td>MR $_{13}$</td></tr>
<tr><td colspan="5">RDI</td><td>MR $_{12}$</td></tr>
<tr><td colspan="5">RSI</td><td>MR $_{11}$</td></tr>
<tr><td colspan="5">RDX</td><td>MR $_{10}$</td></tr>
<tr><td colspan="5">RCX</td><td>MR $_{9}$</td></tr>
<tr><td colspan="5">RAX</td><td>MR $_{8}$</td></tr>
<tr><td colspan="5">R15</td><td>MR $_{7}$</td></tr>
<tr><td colspan="5">R14</td><td>MR $_{6}$</td></tr>
<tr><td colspan="5">R13</td><td>MR $_{5}$</td></tr>
<tr><td colspan="5">R12</td><td>MR $_{4}$</td></tr>
<tr><td colspan="5">R10</td><td>MR $_{3}$</td></tr>
<tr><td colspan="5">RBX</td><td>MR $_{2}$</td></tr>
<tr><td colspan="5">RIP</td><td>MR $_{1}$</td></tr>
<tr><td>$-4/-5_{\ (44)}$</td><td>$0_{\ (4)}$</td><td>$0_{\ (4)}$</td><td>$t = 0_{\ (6)}$</td><td>$u = 20_{\ (6)}$</td><td>MR $_{0}$</td></tr>
</table>

#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.

B.7 Processor Mirroring [amd64]

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.

B.8 Booting [amd64]

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 C

PowerPC Interface

C.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 address of the current thread's UTCB is identical to the thread's local ID, and is thus immutable. The UTCB address is provided in the general purpose register R2 at application start. The R2 register must contain the UTCB address for every system call invocation. 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. ThreadWord0 and ThreadWord1 are free to be used by systems software (e.g., IDL compilers). The kernel associates no semantics with these words.

$\sim_{(32)}$			⟵ UTCB address
⋮		⋮	
ThreadWord 0 $_{(32)}$			–16
ThreadWord 1 $_{(32)}$			–20
VirtualSender/ActualSender $_{(32)}$			–24
IntendedReceiver $_{(32)}$			–28
XferTimeouts $_{(32)}$			–32
ErrorCode $_{(32)}$			–36
$\sim_{(16)}$	cop flags $_{(8)}$	preempt flags $_{(8)}$	–40
ExceptionHandler $_{(32)}$			–44
Pager $_{(32)}$			–48
UserDefinedHandle $_{(32)}$			–52
ProcessorNo $_{(32)}$			–56
MyGlobalId $_{(32)}$			–60

MyLocalId = UTCB address $_{(32)}$	R2

The TCR *MyLocalId* is not part of the UTCB. On PowerPC it is identical with the UTCB address and can be loaded from register R2.

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 40 in the UTCB, and successive message registers follow in memory.

MR$_{0\ldots9}$

MR $_9$	R0
MR $_8$	R10
MR $_7$	R9
MR $_6$	R8
MR $_5$	R7
MR $_4$	R6
MR $_3$	R5
MR $_2$	R4
MR $_1$	R3
MR $_0$	R14

MR$_{10\ldots63}$ **[UTCB fields]**

MR $_{63}$ $_{(32)}$	+252
⋮	
MR $_{11}$ $_{(32)}$	+44
MR $_{10}$ $_{(32)}$	⟵ UTCB address + 40

Buffer Registers (BRs)

The buffer registers map to memory locations in the UTCB. BR $_0$ is at byte offset -64 in the UTCB, BR $_1$ at byte offset -68, etc.

BR$_{0\ldots32}$ **[UTCB fields]**

~ $_{(32)}$	⟵ UTCB address
⋮	
BR $_0$ $_{(32)}$	–64
BR $_1$ $_{(32)}$	–68
⋮	
BR $_{32}$ $_{(32)}$	–196

UTCB Memory With Undefined Semantics

The kernel will associate no semantics with memory located at *UTCB address*. . . *UTCB address* + 39. 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.

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

C.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]

UTCB	*R2*	− **KernelInterface** →	*R2*	≡
–	*R3*		*R3*	*KIP base address*
–	*R4*		*R4*	*API Version*
–	*R5*	tlbia	*R5*	*API Flags*
–	*R6*		*R6*	*Kernel ID*
–	*R7*		*R7*	≡
–	*R8*		*R8*	≡
–	*R9*		*R9*	≡
–	*R10*		*R10*	≡

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

EXCHANGEREGISTERS [Systemcall]

UTCB	*R2*	− **Exchange Registers** →	*R2*	≡
dest	*R3*		*R3*	*result*
control	*R4*		*R4*	*control*
SP	*R5*	call *ExchangeRegisters*	*R5*	*SP*
IP	*R6*		*R6*	*IP*
FLAGS	*R7*		*R7*	*FLAGS*
UserDefinedHandle	*R8*		*R8*	*UserDefinedHandle*
pager	*R9*		*R9*	*pager*
–	*R10*		*R10*	∼

"FLAGS" refers to the user-modifiable PowerPC processor flags that are held in the MSR register. See the PowerPC Processor Mirroring section (page 148).

THREADCONTROL [Privileged Systemcall]

UTCB	*R2*	− **Thread Control** →	*R2*	≡
dest	*R3*		*R3*	*result*
SpaceSpecifier	*R4*		*R4*	∼
Scheduler	*R5*	call *ThreadControl*	*R5*	∼
Pager	*R6*		*R6*	∼
UtcbLocation	*R7*		*R7*	∼
–	*R8*		*R8*	∼
–	*R9*		*R9*	∼
–	*R10*		*R10*	∼

SYSTEMCLOCK [Systemcall]

UTCB	*R2*	− **SystemClock** →	*R2*	≡
–	*R3*		*R3*	*clock 32...63*
–	*R4*		*R4*	*clock 0...31*
–	*R5*	call *SystemClock*	*R5*	∼
–	*R6*		*R6*	∼
–	*R7*		*R7*	∼
–	*R8*		*R8*	∼
–	*R9*		*R9*	∼
–	*R10*		*R10*	∼

THREADSWITCH [Systemcall]

UTCB	*R2*	− **ThreadSwitch** →	*R2*	≡
dest	*R3*		*R3*	∼
–	*R4*		*R4*	∼
–	*R5*	call *ThreadSwitch*	*R5*	∼
–	*R6*		*R6*	∼
–	*R7*		*R7*	∼
–	*R8*		*R8*	∼
–	*R9*		*R9*	∼
–	*R10*		*R10*	∼

SCHEDULE [Systemcall]

UTCB	*R2*	− **Schedule** →	*R2*	≡
dest	*R3*		*R3*	*result*
time control	*R4*		*R4*	*time control*
processor control	*R5*	call *Schedule*	*R5*	∼
prio	*R6*		*R6*	∼
preemption control	*R7*		*R7*	∼
–	*R8*		*R8*	∼
–	*R9*		*R9*	∼
–	*R10*		*R10*	∼

Ipc [Systemcall]

MR_9	*R0*	**− Ipc →**	*R0*	MR_9
−	*R1*		*R1*	≡
UTCB	*R2*		*R2*	≡
MR_1	*R3*	call *Ipc*	*R3*	MR_1
MR_2	*R4*		*R4*	MR_2
MR_3	*R5*		*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*
Timeouts	*R17*		*R17*	∼

Lipc [Systemcall]

MR_9	*R0*	**− Lipc →**	*R0*	MR_9
−	*R1*		*R1*	≡
UTCB	*R2*		*R2*	≡
MR_1	*R3*	call *Lipc*	*R3*	MR_1
MR_2	*R4*		*R4*	MR_2
MR_3	*R5*		*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*
Timeouts	*R17*		*R17*	∼

Unmap [Systemcall]

MR_9	*R0*	**− Unmap →**	*R0*	MR_9
−	*R1*		*R1*	≡
UTCB	*R2*		*R2*	≡
MR_1	*R3*	call *Unmap*	*R3*	MR_1
MR_2	*R4*		*R4*	MR_2
MR_3	*R5*		*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
control	*R15*		*R15*	∼

SPACECONTROL [Privileged Systemcall]

UTCB	*R2*	– **Space Control** →	*R2*	≡
SpaceSpecifier	*R3*		*R3*	*result*
control	*R4*		*R4*	*control*
KernelInterfacePageArea	*R5*	call *SpaceControl*	*R5*	∼
UtcbArea	*R6*		*R6*	∼
Redirector	*R7*		*R7*	∼
–	*R8*		*R8*	∼
–	*R9*		*R9*	∼
–	*R10*		*R10*	∼

PROCESSORCONTROL [Privileged Systemcall]

UTCB	*R2*	– **Processor Control** →	*R2*	≡
processor no	*R3*		*R3*	*result*
InternalFreq	*R4*		*R4*	∼
ExternalFreq	*R5*	call *ProcessorControl*	*R5*	∼
voltage	*R6*		*R6*	∼
–	*R7*		*R7*	∼
–	*R8*		*R8*	∼
–	*R9*		*R9*	∼
–	*R10*		*R10*	∼

MEMORYCONTROL [Privileged Systemcall]

MR_9	*R0*	– **Memory Control** →	*R0*	∼
–	*R1*		*R1*	≡
UTCB	*R2*		*R2*	≡
MR_1	*R3*	call *MemoryControl*	*R3*	*result*
MR_2	*R4*		*R4*	∼
MR_3	*R5*		*R5*	∼
MR_4	*R6*		*R6*	∼
MR_5	*R7*		*R7*	∼
MR_6	*R8*		*R8*	∼
MR_7	*R9*		*R9*	∼
MR_8	*R10*		*R10*	∼
–	*R11*		*R11*	∼
–	*R12*		*R12*	∼
–	*R13*		*R13*	∼
MR_0	*R14*		*R14*	∼
control	*R15*		*R15*	∼
$attribute_0$	*R16*		*R16*	∼
$attribute_1$	*R17*		*R17*	∼
$attribute_2$	*R18*		*R18*	∼
$attribute_3$	*R19*		*R19*	∼

C.3 Memory Attributes [powerpc]

The PowerPC architecture supports the following memory/cache attribute values, to be used with the MEMORYCONTROL system-call:

attribute	value
Default	0
Write-through	1
Write-back	2
Caching-inhibited	3
Caching-enabled	4
Memory-global (coherent)	5
Memory-local (not coherent)	6
Guarded	7
Speculative	8

The default attributes enable write-back, caching, and speculation. Only if the kernel is compiled with support for multiple processors will memory coherency be enabled by default.

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***

C.4 Space Control [powerpc]

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

Input Parameters

control

$0_{(2)}$	t	$0_{(29)}$

t A value of 1 instructs the kernel to add an entry to the translation table for extended mappings. This table allows mapping of memory addresses longer than 32 bits on 32-bit systems. The desired mapping is specified in the remaining parameters of the SpaceControl system call as follows: The redirector field must contain the highest 32 bits of the desired address, while the utcb_area field must contain the lower 32 bits. The kip_area field contains a regular fpage, which specifies a region of 32 bit addresses that should be mapped to a 64 bit address. If any address in this fpage is mapped to a thread, the address will be translated to the corresponding 64 bit address. If the mapping is successfull, the translation table entry is deleted.

Output Parameter

control

$0_{(2)}$	t	$0_{(29)}$

t Indicates if an entry was successfully added to the kernel's translation table for extended mappings.

C.5 Exception Message Format [powerpc]

System Call Trap

System Call Trap Message to Exception Handler

<table>
<tr><td colspan="4">Flags $_{(32)}$</td><td>MR $_{12}$</td></tr>
<tr><td colspan="4">SP $_{(32)}$</td><td>MR $_{11}$</td></tr>
<tr><td colspan="4">IP $_{(32)}$</td><td>MR $_{10}$</td></tr>
<tr><td colspan="4">R0 $_{(32)}$</td><td>MR $_{9}$</td></tr>
<tr><td colspan="4">R10 $_{(32)}$</td><td>MR $_{8}$</td></tr>
<tr><td colspan="4">R9 $_{(32)}$</td><td>MR $_{7}$</td></tr>
<tr><td colspan="4">R8 $_{(32)}$</td><td>MR $_{6}$</td></tr>
<tr><td colspan="4">R7 $_{(32)}$</td><td>MR $_{5}$</td></tr>
<tr><td colspan="4">R6 $_{(32)}$</td><td>MR $_{4}$</td></tr>
<tr><td colspan="4">R5 $_{(32)}$</td><td>MR $_{3}$</td></tr>
<tr><td colspan="4">R4 $_{(32)}$</td><td>MR $_{2}$</td></tr>
<tr><td colspan="4">R3 $_{(32)}$</td><td>MR $_{1}$</td></tr>
<tr><td>-5 $_{(16/48)}$</td><td>0 $_{(4)}$</td><td>$t = 0$ $_{(6)}$</td><td>$u = 12$ $_{(6)}$</td><td>MR $_{0}$</td></tr>
</table>

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

<table>
<tr><td colspan="4">LocalID $_{(32)}$</td><td>MR $_6$</td></tr>
<tr><td colspan="4">ErrorCode $_{(32)}$</td><td>MR $_5$</td></tr>
<tr><td colspan="4">ExceptionNo $_{(32)}$</td><td>MR $_4$</td></tr>
<tr><td colspan="4">Flags $_{(32)}$</td><td>MR $_3$</td></tr>
<tr><td colspan="4">SP $_{(32)}$</td><td>MR $_2$</td></tr>
<tr><td colspan="4">IP $_{(32)}$</td><td>MR $_1$</td></tr>
<tr><td>-5 $_{(16/44)}$</td><td>0 $_{(4)}$</td><td>$t = 0$ $_{(6)}$</td><td>$u = 6$ $_{(6)}$</td><td>MR $_0$</td></tr>
</table>

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.

C.6 Processor Mirroring [powerpc]

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

The kernel will emulate the MFSPR and MTSPR instructions for accessing cpu-specific performance monitor registers on behalf of privileged tasks. The performance monitor registers are global, and not per-thread.

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. The kernel also exposes additional cpu-specific bits.

C.7 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

C.8 Support for Hardware-accelerated Virtualization [powerpc]

In addition to its normal execution mode, L4 provides support for virtualization mode. Virtualization mode is largely common to L4's normal execution model. However, in virtualization mode, threads have access to an extended ISA, and have restricted access to L4-specific features.

Hardware-accelerated virtualization mode (HVM) is based on trap-and-emulate of privileged PowerPC instructions. Threads that execute in that mode have access to an extended architecture that includes the entire privileged instruction set (ideally, within the limits of the hardware facilities). Such a thread can be seen as a virtual CPU, which contains all of the state held by a physical CPU. In addition to the "normal" page faults and exceptions already handled by L4, HVM threads generate virtualization faults on all events that would be observable by the hardware connected to a physical CPU (and some events that would be internal to a physical CPU).

The virtualization extensions introduce new kernel feature strings:

String	Feature
"powerpc-hvm"	Kernel has virtualization support

Extended Thread State

An thread inside a HVM space represents a virtualized physical processor for the virtualization HVM space. It holds all privileged and unprivileged registers of the physical processor. VM-exits cause virtualization fault messages to efficiently manage critical instructions. Virtualization fault replies allow mapping memory into the HVM space and protocol items allow read/write access to the VCPU state. EXCHANGEREGISTERS grants asynchronous access by forcing virtualization faults.

Address Space

In hardware-accelerated virtualization mode, the L4 execution and resource model is mapped onto a *physical* machine model. A thread that executes in HVM has access to the privileged part of the platform architecture and runs with an additional memory translation. Depending on the hardware support for double paging, L4 provides a transparent translation of guest-virtual-to-host-physical translations, based on the guest's TLB entries, and the host's virtual to physical mappings (achieved via L4 mappings).

SPACECONTROL

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

Input Parameters

control

0 (31)	v

v The v field denotes the virtualization mode for all threads in the address space. The v field can only be specified for inactive address spaces and is ignored for active address spaces. The availability of the virtualization features is announced as a KIP feature string.

v=0 An address space with no virtualization support.

v=1 *Hardware virtualization mode* is the hardware-accelerated virtualization support for PowerPC. In hardware virtualization mode, the complete address space is empty and under control of the pager thread. The thread's state is extended by the privileged PowerPC processor.

Output Parameters

control

0 (31)	v

v Indicates if enabling the requested virtualization mode has succeeded ($v = 1$). Zero if $v = 0$ in the input parameter.

PowerPC HVM Control Transfer Items

GPR Group 0 CtrlXferItem ($id = 2$)

R15 $_{(32)}$			MR $_{i+16}$
⋮		⋮	
R2 $_{(32)}$			MR $_{i+3}$
R1 $_{(32)}$			MR $_{i+2}$
R0 $_{(32)}$			MR $_{i+1}$
0xFFFF $_{(20)}$	2 $_{(8)}$	$110C$	MR $_{i}$

GPR Group 1 CtrlXferItem ($id = 3$)

R31 $_{(32)}$			MR $_{i+16}$
⋮		⋮	
R18 $_{(32)}$			MR $_{i+3}$
R17 $_{(32)}$			MR $_{i+2}$
R16 $_{(32)}$			MR $_{i+1}$
0xFFFF $_{(20)}$	3 $_{(8)}$	$110C$	MR $_{i}$

GPR Extended CtrlXferItem ($id = 4$)

IP $_{(32)}$			MR $_{i+5}$
CR $_{(32)}$			MR $_{i+4}$
CTR $_{(32)}$			MR $_{i+3}$
CR $_{(32)}$			MR $_{i+2}$
XER $_{(32)}$			MR $_{i+1}$
0x1F $_{(20)}$	4 $_{(8)}$	$110C$	MR $_{i}$

MMU CtrlXferItem ($id = 6$)

IP $_{(32)}$			MR $_{i+5}$
CR $_{(32)}$			MR $_{i+4}$
CTR $_{(32)}$			MR $_{i+3}$
CR $_{(32)}$			MR $_{i+2}$
XER $_{(32)}$			MR $_{i+1}$
0x1F $_{(20)}$	6 $_{(8)}$	$110C$	MR $_{i}$

Virtualization Fault Protocol

The virtualization protocol is defined between a VCPU thread and its registered pager thread. It substitutes the page fault and exception protocol used for normal threads. Virtualization fault messages are sent to the pager on events related to virtualization that are not handled directly by the hardware or by the L4 microkernel. By default, the kernel will append the fault-specific state specified below when sending kernel messages. Like with the normal fault protocols (see Section 7, the kernel will append additional control transfer items upon requests. Requests to add or remove control transfer items protocol are performed using the EXCHANGEREGISTERS system call and appropriate control transfer configuration items (see Section 2.3).

Virtualization Fault

From Pager:

<table>
<tr><td colspan="4">guest address / instruction info $_{(32)}$</td><td>MR $_3$</td></tr>
<tr><td colspan="4">instruction length $_{(32)}$</td><td>MR $_2$</td></tr>
<tr><td colspan="4">fault qualification $_{(32)}$</td><td>MR $_1$</td></tr>
<tr><td>$-9 - faultID$ $_{(16)}$</td><td>0 $_{(4)}$</td><td>$t = 0$ $_{(6)}$</td><td>$u = 3$ $_{(6)}$</td><td>MR $_0$</td></tr>
</table>

fault ID Implementation-specific fault identifiier. For Intel VT-x, the identifier corresponds to the VM exit reason.

fault qualification Additional information about the cause of exits.

instruction length The length of the faulting instruction.

operand info Guest linear address / Additional information about the faulting instruction .

value For a read fault, the virtual value of the faulting register, if the register is part of the VCPU state.

Virtualization Fault Reply

From Pager:

<table>
<tr><td colspan="4">CtrlXferItem n</td><td>MR $_{c_0+\ldots+c_n+3}$</td></tr>
<tr><td colspan="4">⋮</td><td></td></tr>
<tr><td colspan="4">CtrlXferItem 0</td><td>MR $_{c_0+3}$</td></tr>
<tr><td colspan="4">MapItem / GrantItem</td><td>MR $_{1,2}$</td></tr>
<tr><td>0 $_{(148)}$</td><td>0 $_{(4)}$</td><td>$t = 2 + \sum c_i$ $_{(6)}$</td><td>$u = 0$ $_{(6)}$</td><td>MR $_0$</td></tr>
</table>

Appendix D

PowerPC64 Interface

D.1 Virtual Registers [powerpc64]

Thread Control Registers (TCRs)

TCRs are mapped to memory locations. They are implemented as part of the ppc64-specific user-level thread control block (UTCB). The address of the current thread's UTCB is identical to the thread's local ID, and is thus immutable. 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 is provided in the abi thread register *r13* at application start. 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. ThreadWord0 and ThreadWord1 are free to be used by systems software (e.g., IDL compilers). The kernel associates no semantics with these words.

ThreadWord 1 $_{(64)}$			+ 88
ThreadWord 0 $_{(64)}$			+ 80
~ $_{(48)}$	cop flags $_{(8)}$	preempt flags $_{(8)}$	+72
ProcessorNo $_{(64)}$			+64
VirtualSender/ActualSender $_{(64)}$			+56
IntendedReceiver $_{(64)}$			+48
ErrorCode $_{(64)}$			+40
XferTimeouts $_{(64)}$			+32
UserDefinedHandle $_{(64)}$			+24
ExceptionHandler $_{(64)}$			+16
Pager $_{(64)}$			+8
MyGlobalId $_{(64)}$			⟵ UTCB address

MyLocalId = UTCB address $_{(64)}$	r13

The TCR *MyLocalId* is not part of the UTCB. On PowerPC64 it is identical with the UTCB address and can be loaded from register *r13*.

Message Registers (MRs)

Message registers MR $_0$ through MR $_9$ map to local registers in the processor's general purpose register file for IPC and LIPC calls, otherwise they are located in the UTCB. The remaining message registers map to memory locations in the UTCB. MR $_0$ starts at byte offset 512 in the UTCB, and successive message registers follow in memory.

MR $_{0\ldots9}$

MR $_9$	r23
MR $_8$	r22
MR $_7$	r21
MR $_6$	r20
MR $_5$	r19
MR $_4$	r18
MR $_3$	r17
MR $_2$	r16
MR $_1$	r15
MR $_0$	r14

MR $_{0\ldots63}$ **[UTCB fields]**

MR $_{63}$ $_{(64)}$	+1016
⋮	
MR $_0$ $_{(64)}$	⟵ UTCB address + 512

Buffer Registers (BRs)

The buffer registers map to memory locations in the UTCB. BR $_0$ is at byte offset 248 in the UTCB, BR $_1$ at byte offset 256, etc.

BR $_{0\ldots32}$ **[UTCB fields]**

BR $_{32}$ $_{(64)}$	+504
⋮	
BR $_1$ $_{(64)}$	+256
BR $_0$ $_{(64)}$	⟵ UTCB address + 248

UTCB Memory With Undefined Semantics

The kernel will associate no semantics with memory located at *UTCB address* + 80… *UTCB address* + 247. 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.

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

D.2 Systemcalls [powerpc64]

The system-calls which are invoked by the *bctrl* or 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 *lr*.

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.

The system call definitions below only specify the contexts of the general purpose registers. Except for the KERNELINTERFACE system-call, the contents of user accessible state registers are assumed to be scratched. The floating-point registers are assumed to be preserved accross system calls.

KERNELINTERFACE [Slow Systemcall]

–	*r0...r2*	– **KernelInterface** →	*r0...r2*	≡
–	*r3*		*r3*	*KIP base address*
–	*r4*		*r4*	*API Version*
–	*r5*	tlbia	*r5*	*API Flags*
–	*r6*		*r6*	*Kernel ID*
–	*r7...r31*		*r7...r31*	≡
–	*lr*		*lr*	≡
–	*ctr*		*ctr*	≡
–	*cr*		*cr*	≡
–	*xer*		*xer*	≡

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

EXCHANGEREGISTERS [Systemcall]

–	*r0*	– **Exchange Registers** →	*r0*	∼
–	*r1*		*r1*	≡
–	*r2*		*r2*	≡
dest	*r3*	bctrl	*r3*	*result*
control	*r4*		*r4*	*control*
SP	*r5*		*r5*	*SP*
IP	*r6*		*r6*	*IP*
FLAGS	*r7*		*r7*	*FLAGS*
UserDefinedHandle	*r8*		*r8*	*UserDefinedHandle*
pager	*r9*		*r9*	*pager*
isLocal	*r10*		*r10*	*isLocal*
–	*r11, r12*		*r11, r12*	∼
UTCB	*r13*		*r13*	*UTCB*
–	*r14...r29*		*r14...r29*	∼
–	*r30, r31*		*r30, r31*	≡
–	*lr*		*lr*	∼
ExchangeRegisters	*ctr*		*ctr*	∼
–	*cr*		*cr*	∼
–	*xer*		*xer*	∼

"FLAGS" refers to the user-modifiable powerpc64 processor flags that are held in the *msr* register.

ThreadControl **[Privileged Systemcall]**

−	*r0*	− **Thread Control** →	*r0*	∼
−	*r1*		*r1*	≡
−	*r2*		*r2*	≡
dest	*r3*	bctrl	*r3*	*result*
space	*r4*		*r4*	∼
scheduler	*r5*		*r5*	∼
pager	*r6*		*r6*	∼
UtcbLocation	*r7*		*r7*	∼
−	*r8...r12*		*r8...r12*	∼
UTCB	*r13*		*r13*	*UTCB*
−	*r14...r29*		*r14...r29*	∼
−	*r30, r31*		*r30, r31*	≡
−	*lr*		*lr*	∼
ThreadControl	*ctr*		*ctr*	∼
−	*cr*		*cr*	∼
−	*xer*		*xer*	∼

SystemClock **[Systemcall]**

−	*r0*	− **SystemClock** →	*r0*	∼
−	*r1*		*r1*	≡
−	*r2*		*r2*	≡
−	*r3*	bctrl	*r3*	*clock*
−	*r4...r12*		*r4...r12*	∼
UTCB	*r13*		*r13*	*UTCB*
−	*r14...r29*		*r14...r29*	∼
−	*r30, r31*		*r30, r31*	≡
−	*lr*		*lr*	∼
SystemClock	*ctr*		*ctr*	∼
−	*cr*		*cr*	∼
−	*xer*		*xer*	∼

ThreadSwitch **[Systemcall]**

−	*r0*	− **ThreadSwitch** →	*r0*	∼
−	*r1*		*r1*	≡
−	*r2*		*r2*	≡
dest	*r3*	bctrl	*r3*	∼
−	*r4...r12*		*r4...r12*	∼
UTCB	*r13*		*r13*	*UTCB*
−	*r14...r29*		*r14...r29*	∼
−	*r30, r31*		*r30, r31*	≡
−	*lr*		*lr*	∼
ThreadSwitch	*ctr*		*ctr*	∼
−	*cr*		*cr*	∼
−	*xer*		*xer*	∼

SCHEDULE [Systemcall]

–	*r0*	– **Schedule** →	*r0*	∼
–	*r1*		*r1*	≡
–	*r2*		*r2*	≡
dest	*r3*	bctrl	*r3*	*result*
time control	*r4*		*r4*	*time control*
processor control	*r5*		*r5*	∼
priority	*r6*		*r6*	∼
preemption control	*r7*		*r7*	∼
–	*r8. . . r12*		*r8. . . r12*	∼
UTCB	*r13*		*r13*	*UTCB*
–	*r14. . . r29*		*r14. . . r29*	∼
–	*r30, r31*		*r30, r31*	≡
–	*lr*		*lr*	∼
Schedule	*ctr*		*ctr*	∼
–	*cr*		*cr*	∼
–	*xer*		*xer*	∼

IPC [Systemcall]

–	*r0*	– **Ipc** →	*r0*	∼
–	*r1*		*r1*	≡
–	*r2*		*r2*	≡
to	*r3*	bctrl	*r3*	*from*
FromSpecifier	*r4*		*r4*	∼
Timeouts	*r5*		*r5*	∼
–	*r6. . . r12*		*r6. . . r12*	∼
UTCB	*r13*		*r13*	*UTCB*
MR_0	*r14*		*r14*	MR_0
MR_1	*r15*		*r15*	MR_1
MR_2	*r16*		*r16*	MR_2
MR_3	*r17*		*r17*	MR_3
MR_4	*r18*		*r18*	MR_4
MR_5	*r19*		*r19*	MR_5
MR_6	*r20*		*r20*	MR_6
MR_7	*r21*		*r21*	MR_7
MR_8	*r22*		*r22*	MR_8
MR_9	*r23*		*r23*	MR_9
–	*r24. . . r29*		*r24. . . r29*	∼
–	*r30, r31*		*r30, r31*	≡
–	*lr*		*lr*	∼
Ipc	*ctr*		*ctr*	∼
–	*cr*		*cr*	∼
–	*xer*		*xer*	∼

LIPC [Systemcall]

–	*r0*	**– Lipc →**	*r0*	∼
–	*r1*		*r1*	≡
–	*r2*		*r2*	≡
to	*r3*	bctrl	*r3*	*from*
FromSpecifier	*r4*		*r4*	∼
Timeouts	*r5*		*r5*	∼
–	*r6... r12*		*r6... r12*	∼
UTCB	*r13*		*r13*	*UTCB*
MR_0	*r14*		*r14*	MR_0
MR_1	*r15*		*r15*	MR_1
MR_2	*r16*		*r16*	MR_2
MR_3	*r17*		*r17*	MR_3
MR_4	*r18*		*r18*	MR_4
MR_5	*r19*		*r19*	MR_5
MR_6	*r20*		*r20*	MR_6
MR_7	*r21*		*r21*	MR_7
MR_8	*r22*		*r22*	MR_8
MR_9	*r23*		*r23*	MR_9
–	*r24... r29*		*r24... r29*	∼
–	*r30, r31*		*r30, r31*	≡
–	*lr*		*lr*	∼
Lipc	*ctr*		*ctr*	∼
–	*cr*		*cr*	∼
–	*xer*		*xer*	∼

UNMAP [Systemcall]

–	*r0*	**– Unmap →**	*r0*	∼
–	*r1*		*r1*	≡
–	*r2*		*r2*	≡
control	*r3*	bctrl	*r3*	∼
–	*r4... r12*		*r4... r12*	∼
UTCB	*r13*		*r13*	*UTCB*
–	*r14... r29*		*r14... r29*	∼
–	*r30, r31*		*r30, r31*	≡
–	*lr*		*lr*	∼
Unmap	*ctr*		*ctr*	∼
–	*cr*		*cr*	∼
–	*xer*		*xer*	∼

SPACECONTROL [Privileged Systemcall]

–	*r0*	**– Space Control →**	*r0*	∼
–	*r1*		*r1*	≡
–	*r2*		*r2*	≡
SpaceSpecifier	*r3*	bctrl	*r3*	*result*
control	*r4*		*r4*	*control*
KernelInterfacePageArea	*r5*		*r5*	∼
UtcbArea	*r6*		*r6*	∼
Redirector	*r7*		*r7*	∼
–	*r8... r12*		*r8... r12*	∼
UTCB	*r13*		*r13*	*UTCB*
–	*r14... r29*		*r14... r29*	∼
–	*r30, r31*		*r30, r31*	≡
–	*lr*		*lr*	∼
SpaceControl	*ctr*		*ctr*	∼
–	*cr*		*cr*	∼
–	*xer*		*xer*	∼

PROCESSORCONTROL [Privileged Systemcall]

–	*r0*	**– Processor Control →**	*r0*	∼
–	*r1*		*r1*	≡
–	*r2*		*r2*	≡
ProcessorNo	*r3*	bctrl	*r3*	*result*
InternalFreq	*r4*		*r4*	∼
ExternalFreq	*r5*		*r5*	∼
voltage	*r6*		*r6*	∼
–	*r7. . . r12*		*r7. . . r12*	∼
UTCB	*r13*		*r13*	*UTCB*
–	*r14. . . r29*		*r14. . . r29*	∼
–	*r30, r31*		*r30, r31*	≡
–	*lr*		*lr*	∼
ProcessorControl	*ctr*		*ctr*	∼
–	*cr*		*cr*	∼
–	*xer*		*xer*	∼

MEMORYCONTROL [Privileged Systemcall]

–	*r0*	**– Memory Control →**	*r0*	∼
–	*r1*		*r1*	≡
–	*r2*		*r2*	≡
control	*r3*	bctrl	*r3*	*result*
$attribute_0$	*r4*		*r4*	∼
$attribute_1$	*r5*		*r5*	∼
$attribute_2$	*r6*		*r6*	∼
$attribute_3$	*r7*		*r7*	∼
–	*r8. . . r12*		*r8. . . r12*	∼
UTCB	*r13*		*r13*	*UTCB*
–	*r14. . . r29*		*r14. . . r29*	∼
–	*r30, r31*		*r30, r31*	≡
–	*lr*		*lr*	∼
MemoryControl	*ctr*		*ctr*	∼
–	*cr*		*cr*	∼
–	*xer*		*xer*	∼

D.3 Memory Attributes [powerpc64]

The powerpc64 architecture supports the following memory/cache attribute values, to be used with the MEMORYCONTROL system-call:

attribute	value
Default	0
Uncached	1
Coherent	2

The default attributes depend on the platform and not all modes are defined for all processors.

D.4 Exception Message Format [powerpc64]

System Call Trap

System Call Trap Message to Exception Handler

<table>
<tr><td colspan="4">Flags $_{(64)}$</td><td>MR $_{12}$</td></tr>
<tr><td colspan="4">SP $_{(64)}$</td><td>MR $_{11}$</td></tr>
<tr><td colspan="4">IP $_{(64)}$</td><td>MR $_{10}$</td></tr>
<tr><td colspan="4">r0 $_{(64)}$</td><td>MR $_{9}$</td></tr>
<tr><td colspan="4">r10 $_{(64)}$</td><td>MR $_{8}$</td></tr>
<tr><td colspan="4">r9 $_{(64)}$</td><td>MR $_{7}$</td></tr>
<tr><td colspan="4">r8 $_{(64)}$</td><td>MR $_{6}$</td></tr>
<tr><td colspan="4">r7 $_{(64)}$</td><td>MR $_{5}$</td></tr>
<tr><td colspan="4">r6 $_{(64)}$</td><td>MR $_{4}$</td></tr>
<tr><td colspan="4">r5 $_{(64)}$</td><td>MR $_{3}$</td></tr>
<tr><td colspan="4">r4 $_{(64)}$</td><td>MR $_{2}$</td></tr>
<tr><td colspan="4">r3 $_{(64)}$</td><td>MR $_{1}$</td></tr>
<tr><td>-5 $_{(44)}$</td><td>0 $_{(4)}$</td><td>$t = 0$ $_{(6)}$</td><td>$u = 12$ $_{(6)}$</td><td>MR $_{0}$</td></tr>
</table>

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 a *sc* instruction. State is preserved similarly for the inclusive set of saved registers according the 64-bit PowerPC ELF ABI for function calls.

The non-volatile registers are: *r1, r2, r13 . . . r31, CR2 . . . CR4*

The volatile registers are: *r0, r3 . . . r12, LR, CTR, XER, CR0, CR1, CR5 . . . CR7*

Thread virtual registers may also be clobbered.

Generic Traps

Generic Trap Message To Exception Handler

<table>
<tr><td colspan="4">*ErrorAddress* $_{(64)}$</td><td>MR $_7$</td></tr>
<tr><td colspan="4">LocalID $_{(64)}$</td><td>MR $_6$</td></tr>
<tr><td colspan="4">ErrorCode $_{(64)}$</td><td>MR $_5$</td></tr>
<tr><td colspan="4">ExceptionNo $_{(64)}$</td><td>MR $_4$</td></tr>
<tr><td colspan="4">Flags $_{(64)}$</td><td>MR $_3$</td></tr>
<tr><td colspan="4">SP $_{(64)}$</td><td>MR $_2$</td></tr>
<tr><td colspan="4">IP $_{(64)}$</td><td>MR $_1$</td></tr>
<tr><td>-5 $_{(44)}$</td><td>0 $_{(4)}$</td><td>$t = 0$ $_{(6)}$</td><td>$u = 6/7$ $_{(6)}$</td><td>MR $_0$</td></tr>
</table>

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. Exceptions that provide an error address use the *ErrorAddress* register and specify 7 Untyped words, otherwise only 6 Untyped words will be sent. The kernel preserves all user state, including thread virtual registers.

For some exceptions, The following is a table of values for the Generic Trap *ExceptionNo*:

Exception	ExceptionNo	ErrorCode	Delivered	ErrorAddress
System Reset	0x100	-	No	-
Machine Check	0x200	-	No	-
DSI	0x300	DSISR	If not paging related	Yes
ISI	0x400	-	If not paging related	No
Interrupt	0x500	-	No	No
Alignment	0x600	DSISR	Yes	Yes
Program	0x700	-	Yes	Yes
FPU Unavailable	0x800	-	No	-
Decrementer	0x900	-	No	-
System Call	0xc00	-	No	-
Trace	0xd00	-	If kdb not using	No
FPU Assist	0xe00	-	Yes	No
Performance	0xf00	-	Yes	No
Breakpoint	0x1300	-	Yes	No
Soft Patch	0x1500	-	Yes	No
Maintenance	0x1600	-	Yes	No
Instrumentation	0x2000	-	Yes	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.

D.5 Booting [powerpc64]

IBM OpenFirmware 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 OpenPower 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.

First BootInfo Record	First Entry

~			Num Entries	+10 / +20
First Entry	Size	Version	Magic	BootInfo
+C / +18	+8 / +10	+4 / +8	+0	

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.

Offset Next	Version	Type
+8 / +10	+4 / +8	+0

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 72). 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 gains 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
ErrorCode () Word	*–none–*	69
ErrorCode () Word	*–none–*	75
ErrorCode () Word	*–none–*	77
ErrUtcbArea Word const	*–n/a–*	25
ErrUtcbArea Word const	*–n/a–*	47
ExceptionHandler () ThreadId	*–none–*	17
ExceptionHandler () ThreadId	*–none–*	72
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–*	41
ExternalFreq (ProcDesc& p) Word	*–none–*	10
Feature (void* KernelInterface, Word num) char*	*–none–*	9
Flush (Fpage f) Fpage	Unmap	44
Flush (Word n, Fpage& [n] fpages) void	Unmap	44
Fpage data type	*–n/a–*	40
$-$ (Fpage f, Word AccessRights) Fpage	*–none–*	41
$+$ (Fpage f, Word AccessRights) Fpage	*–none–*	41
FpageLog2 (Word BaseAddress, int Log2FpageSize $<$ 64) Fpage	*–none–*	41
Fpage (Word BaseAddress, int FpageSize $\geq$ 1K) Fpage	*–none–*	41
FPURegsCtrlXferItem data type	*–n/a–*	109
FullyAccessible Word const	*–n/a–*	41
Get (Msg& msg, Word t, CtrlXferItem& c) Word	*–none–*	54
Get (Msg& msg, Word t, GrantItem& g) Word	*–none–*	54
Get (Msg& msg, Word t, MapItem& m) Word	*–none–*	54
Get (Msg& msg, Word t, StringItem& s) Word	*–none–*	54
Get (Msg& msg, Word& ut, {MapItem, GrantItem, StringItem}& Items) void	*–none–*	53
Get (Msg& msg, Word u) Word	*–none–*	54
Get (Msg& msg, Word u, Word& w) void	*–none–*	54
GetStatus (Fpage f) Fpage	*–none–*	44
GlobalId (ThreadId t) ThreadId	ExchangeRegisters	15
GlobalId (ThreadId t) ThreadId	ExchangeRegisters	21
GlobalId (Word threadno, version) ThreadId	*–none–*	15
GlobalMemory Word const	*–n/a–*	144
GPRegsCtrlXferItem data type	*–n/a–*	109
GrantItem data type	*–n/a–*	57
GrantItem (Fpage f, Word SndBase) GrantItem	*–none–*	57
GrantItem (GrantItem g) Bool	*–none–*	57
GuardedMemory Word const	*–n/a–*	144
High (MemoryDesc& m) Word	*–none–*	9
IntendedReceiver () ThreadId	*–none–*	17
IntendedReceiver () ThreadId	*–none–*	69
InternalFreq (ProcDesc& p) Word	*–none–*	10
IoFpageLog2 (Word BaseAddress, int Log2FpageSize <= 16) Fpage	*–none–*	130
IoFpageLog2 (Word BasePort, int Log2FpageSize <= 16) Fpage	*–none–*	103
IoFpagePort (Fpage f) Word	*–none–*	103
IoFpageSize (Fpage f) Word	*–none–*	103
IoFpageSizeLog2 (Fpage f) Word	*–none–*	103
IoFpage (Word BaseAddress, int FpageSize) Fpage	*–none–*	130
IoFpage (Word BasePort, int FpageSize) Fpage	*–none–*	103
IpcFailed (MsgTag t) Bool	*–none–*	69
IpcPropagated (MsgTag t) Bool	*–none–*	69
IpcRedirected (MsgTag t) Bool	*–none–*	69
IpcSucceeded (MsgTag t) Bool	*–none–*	69
Ipc (ThreadId to, FromSpecifier, Word Timeouts, ThreadId& from) MsgTag	Ipc	68
IpcXcpu (MsgTag t) Bool	*–none–*	69
IsGlobalId (ThreadId t) Bool	*–none–*	15
IsIoFpage (Fpage f) Bool	*–none–*	103
IsLocalId (ThreadId t) Bool	*–none–*	15
IsNilFpage (Fpage f) Bool	*–none–*	41
IsNilThread (ThreadId t) Bool	*–none–*	15
IsVirtual (MemoryDesc& m) Bool	*–none–*	9
KernelGenDate (void* KernelInterface, Word& year, month, day) void	*–none–*	8

	used system call	page
Set_Rights (Fpage& f, Word AccessRights) void	*–none–*	41
Set_Timeslice (ThreadId dest, Time ts, Time tq) Word	*–none–*	35
Set_UserDefinedHandle (ThreadId t, Word handle) void	ExchangeRegisters	21
Set_UserDefinedHandle (Word NewValue) void	*–none–*	17
Set_VirtualSender (ThreadId t) void	*–none–*	17
Set_VirtualSender (ThreadId t) void	*–none–*	70
Set_XferTimeouts (Word NewValue) void	*–none–*	17
SharedMemoryType Word const	*–n/a–*	9
SimpleExec_BssPstart (BootRec* b) Word	*–none–*	171
SimpleExec_BssSize (BootRec* b) Word	*–none–*	172
SimpleExec_BssVstart (BootRec* b) Word	*–none–*	171
SimpleExec_Cmdline (BootRec* b) char*	*–none–*	172
SimpleExec_DataPstart (BootRec* b) Word	*–none–*	171
SimpleExec_DataSize (BootRec* b) Word	*–none–*	171
SimpleExec_DataVstart (BootRec* b) Word	*–none–*	171
SimpleExec_Flags (BootRec* b) Word	*–none–*	172
SimpleExec_InitialIP (BootRec* b) Word	*–none–*	172
SimpleExec_Label (BootRec* b) Word	*–none–*	172
SimpleExec_Set_Flags (BootRec* b, Word w) void	*–none–*	172
SimpleExec_Set_Label (BootRec* b, Word w) void	*–none–*	172
SimpleExec_TextPstart (BootRec* b) Word	*–none–*	171
SimpleExec_TextSize (BootRec* b) Word	*–none–*	171
SimpleExec_TextVstart (BootRec* b) Word	*–none–*	171
Size (Fpage f) Word	*–none–*	41
SizeLog2 (Fpage f) Word	*–none–*	41
Sleep (Time t) void	Ipc	69
SmallSpace (Word location, size) Word	*–none–*	105
SndBase (GrantItem g) Word	*–none–*	57
SndBase (MapItem m) Word	*–none–*	56
SndFpage (GrantItem g) Fpage	*–none–*	57
SndFpage (MapItem m) Fpage	*–none–*	56
SpaceControl (ThreadId SpaceSpecifier, Word control, Fpage KernelInterfacePageArea, UtcbArea, ThreadId Redirector, Word& old_Control) Word	SpaceControl	47
SpeculativeMemory Word const	*–n/a–*	144
Start (ThreadId t) void	ExchangeRegisters	22
Start (ThreadId t, Word sp, ip, flags) void	ExchangeRegisters	22
Start (ThreadId t, Word sp, ip) void	ExchangeRegisters	22
Stop (ThreadId t) ThreadState	ExchangeRegisters	22
Stop (ThreadId t, Word& sp, ip, flags) ThreadState	ExchangeRegisters	22
StoreBR (int i, Word& w) void	*–none–*	11
StoreBR (int i, Word& w) void	*–none–*	63
StoreBRs (int i, k, Word& [k]) void	*–none–*	11
StoreBRs (int i, k, Word& [k]) void	*–none–*	63
StoreMR (int i, Word& w) void	*–none–*	11
StoreMR (int i, Word& w) void	*–none–*	54
StoreMRs (int i, k, Word& [k] w) void	*–none–*	11
StoreMRs (int i, k, Word& [k] w) void	*–none–*	54
Store (MsgTag t, Msg& msg) void	*–none–*	53
StringItem data type	*–n/a–*	60
StringItem (int size, void* address) StringItem	*–none–*	60
StringItems (Acceptor a) Bool	*–none–*	63
StringItemsAcceptor Acceptor const	*–n/a–*	62
$+$ (StringItem s, CacheAllocationHint h) StringItem	*–none–*	61
StringItem (StringItem& s) Bool	*–none–*	60
Substrings (StringItem& s) Word	*–none–*	60
Substring (StringItem& s, Word n) void*	*–none–*	60
SystemClock () Clock	SystemClock	29
ThreadControl (ThreadId dest, SpaceSpecifier, Scheduler, Pager, void* UtcbLocation) Word	ThreadControl	25
ThreadIdBits (void* KernelInterface) Word	*–none–*	8
ThreadId data type	*–n/a–*	15
$==$ (ThreadId l, r) Bool	*–none–*	15
ThreadIdSystemBase (void* KernelInterface) Word	*–none–*	8

Index