# L4 Reference Manual

# Alpha 21x64

**Daniel Potts, Simon Winwood, Gernot Heiser**

UNSW CS&E Technical Report TR-0104
April 2001

disy@cse.unsw.edu.au
http://www.cse.unsw.edu.au/∼disy/
Operating Systems and Distributed Systems Group
School of Computer Science and Engineering
The University of New South Wales
UNSW Sydney 2052, Australia

## Note

This document describes release 2.0 of the L4 microkernel for the Alpha microprocessor family. It is based on the L4/x86 reference manual Version 2.0 by Jochen Liedtke and the L4/MIPS R4x00 reference manual Version 1.0 by Kevin Elphinstone, Gernot Heiser and Jochen Liedtke, modified as appropriate to describe the Alpha implementation.

Comments and critiques, as well as proposed additions and alternatives for future versions are most welcome.

The source code for L4/Alpha is available free of charge under the terms of the GNU General Public License. To obtain the source check `http://l4alpha.sourceforge.net` or `http://www.cse.unsw.edu.au/~disy/L4/`. Future versions of this document, as well as related documents and tools, will be available from the source distribution sites.

## How To Read This Manual

This reference manual consists of two parts, (1) a processor-independent description of the principles and mechanisms of L4, (2) a detailed Alpha processor-specific description.

Where L4/Alpha differs from L4/MIPS or L4/x86 significantly, or something is partially or completely unimplemented, then an implementation note appears as below. There is also a summary of various implementation details in section 2.1.

**Alpha Implementation Note:** This is what an implementation note looks like.

## Acknowledgements

The original L4 reference manual was written by Jochen Liedtke, who would like to thank many people for their helpful contributions for improving the reference manual and the L4 interface. Particular thanks go to Bryan Ford, Hermann Härtig, Michael Hohmuth, Sebastian Schönberg and Jean Wolter. For the MIPS version we would like to thank in particular Jerry Vochteloo for testing the kernel, Alan Au for contributions to the manual, as well as the 1997 class of UNSW COMP9242 "guinea pigs" who built their operating systems on top of the MIPS version of the kernel.

# Contents

# Chapter 1

# L4 in General

## 1.1 Basic Concepts

The following section contains excerpts from [Lie93b, Lie93a, Lie95].

We reason about the minimal concepts or "primitives" that a $\mu$-kernel should implement.[1] The determining criterion used is functionality, not performance. More precisely, a concept is tolerated inside the $\mu$-kernel only if moving it outside the kernel, i.e. permitting competing implementations, would prevent the implementation of the system's required functionality.

We assume that the target system has to support interactive and/or not completely trustworthy applications, i.e. it has to deal with protection. We further assume that the hardware implements page-based virtual memory.

One inevitable requirement for such a system is that a programmer must be able to implement an arbitrary subsystem $S$ in such a way that it cannot be disturbed or corrupted by other subsystems $S'$. This is the principle of independence: $S$ can give guarantees independent of $S'$. The second requirement is that other subsystems must be able to rely on these guarantees. This is the principle of integrity: there must be a way for $S_1$ to address $S_2$ and to establish a communication channel which can neither be corrupted nor eavesdropped by $S'$.

Provided hardware and kernel are trustworthy, further security services, like those described by [GGKL89], can be implemented by servers. Their integrity can be ensured by system administration or by user-level boot servers. For illustration: a key server should deliver public-secret RSA key pairs on demand. It should guarantee that each pair has the desired RSA property and that each pair is delivered only once and only to the demander. The key server can only be realized if there are mechanisms which (a) protect its code and data, (b) ensure that nobody else reads or modifies the key and (c) enable the demander to check whether the key comes from the key server. Finding the key server can be done by means of a name server and checked by public key based authentication.

### 1.1.1 Address Spaces

At the hardware level, an *address space* is a mapping which associates each virtual page to a physical page frame or marks it 'non-accessible'. For the sake of simplicity, we omit access attributes like read-only and read/write. The mapping is implemented by TLB hardware and page tables.

The $\mu$-kernel, the mandatory layer common to all subsystems, has to hide the hardware concept of address spaces, since otherwise, implementing protection would be impossible. The $\mu$-kernel concept of address spaces must be tamed, but must permit the implementation of arbitrary protection

[1] Proving minimality, necessarity and completeness would be nice but is impossible, since there is no agreed-upon metric and all is Turing-equivalent.

(and non-protection) schemes on top of the $\mu$-kernel. It should be simple and similar to the hardware concept.

The basic idea is to support recursive construction of address spaces outside the kernel. By magic, there is one address space $\sigma_0$ which essentially represents the physical memory and is controlled by the first subsystem $S_0$. At system start time, all other address spaces are empty. For constructing and maintaining further address spaces on top of $\sigma_0$, the $\mu$-kernel provides three operations:

**Grant.** The owner of an address space can *grant* any of its pages to another space, provided the recipient agrees. The granted page is removed from the granter's address space and included into the grantee's address space. The important restriction is that instead of physical page frames, the granter can only grant pages which are already accessible to itself.

**Map.** The owner of an address space can *map* any of its pages into another address space, provided the recipient agrees. Afterwards, the page can be accessed in both address spaces. In contrast to granting, the page is not removed from the mapper's address space. Comparable to the granting case, the mapper can only map pages which itself already can access.

**Flush.** The owner of an address space can *flush* any of its pages. The flushed page remains accessible in the flusher's address space, but is removed from all other address spaces which had received the page directly or indirectly from the flusher. Although explicit consent of the affected address-space owners is not required, the operation is safe, since it is restricted to own pages. The users of these pages already agreed to accept a potential flushing, when they received the pages by mapping or granting.

## Reasoning

The described address-space concept leaves memory management and paging outside the $\mu$-kernel; only the grant, map and flush operations are retained inside the kernel. Mapping and flushing are required to implement memory managers and pagers on top of the $\mu$-kernel.

The grant operation is required only in very special situations: consider a pager $F$ which combines two underlying file systems (implemented as pagers $f_1$ and $f_2$, operating on top of the standard pager) into one unified file system (see figure 1.1). In this example, $f_1$ maps one of its pages to $F$

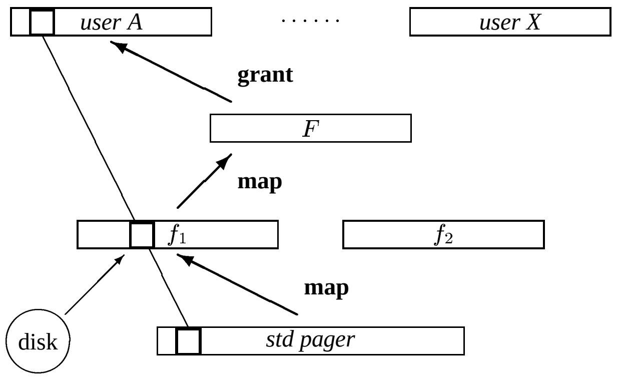


Figure 1.1: *A Granting Example.*

which grants the received page to *user A*. By granting, the page disappears from $F$ so that it is then available only in $f_1$ and *user A*; the resulting mappings are denoted by the thin line: the page is mapped in *user A*, $f_1$ and the standard pager. Flushing the page by the standard pager would affect $f_1$ and *user A*, flushing by $f_1$ only *user A*. $F$ is not affected by either flush (and cannot flush itself), since it used the page only transiently. If $F$ had used mapping instead of granting, it would have needed to replicate most of the bookkeeping which is already done in $f_1$ and $f_2$. Furthermore, granting avoids a potential address-space overflow of $F$.

In general, granting is used when page mappings should be passed through a controlling subsystem without burdening the controller's address space by all pages mapped through it.

The model can easily be extended to access rights on pages. Mapping and granting copy the source page's access right or a subset of them, i.e., can restrict the access but not widen it. Special flushing operations may remove only specified access rights.

## I/O

An address space is the natural abstraction for incorporating device ports. This is obvious for memory mapped I/O, but I/O ports can also be included. The granularity of control depends on the given processor. The 386 and its successors permit control per port (one very small page per port) but no mapping of port addresses (it enforces mappings with $v{=}v'$); the PowerPC uses pure memory mapped I/O, i.e., device ports can be controlled and mapped with 4K granularity. Controlling I/O rights and device drivers is thus also done by memory managers and pagers on top of the $\mu$-kernel.

## An Abstract Model of Address Spaces

We describe address spaces as mappings. $\sigma_0 : V \to R \cup \{\phi\}$ is the initial address space, where $V$ is the set of virtual pages, $R$ the set of available physical (real) pages and $\phi$ the nilpage which cannot be accessed. Further address spaces are defined recursively as mappings $\sigma : V \to (\Sigma \times V) \cup \{\phi\}$, where $\Sigma$ is the set of address spaces. It is convenient to regard each mapping as a one column table which contains $\sigma(v)$ for all $v{\in}V$ and can be indexed by $v$. We denote the elements of this table by $\sigma_v$.

All modifications of address spaces are based on the replacement operation: we write $\sigma_v \leftarrow x$ to describe a change of $\sigma$ at $v$, precisely:

$$\text{flush } (\sigma, v) \quad ; \quad \sigma_v := x \quad .$$

A page potentially mapped at $v$ in $\sigma$ is flushed, and the new value $x$ is copied into $\sigma_v$. This operation is internal to the $\mu$-kernel. We use it only for describing the three exported operations.

A subsystem $S$ with address space $\sigma$ can *grant* any of its pages $v$ to a subsystem $S'$ with address space $\sigma'$ provided $S'$ agrees:

$$\sigma'_{v'} \leftarrow \sigma_v \quad , \quad \sigma_v \leftarrow \phi \quad .$$

Note that $S$ determines which of its pages should be granted, whereas $S'$ determines at which virtual address the granted page should be mapped in $\sigma'$. The granted page is transferred to $\sigma'$ and removed from $\sigma$.

A subsystem $S$ with address space $\sigma$ can *map* any of its pages $v$ to a subsystem $S'$ with address space $\sigma'$ provided $S'$ agrees:

$$\sigma'_{v'} \leftarrow (\sigma, v) \quad .$$

In contrast to grant, the mapped page remains in the mapper's space $\sigma$ and *a link to the page in the mapper's address space* $(\sigma, v)$ *is stored in the receiving address space* $\sigma'$, instead of transferring the existing link from $\sigma_v$ to $\sigma'_{v'}$. This operation permits to construct address spaces recursively, i.e. new spaces based on existing ones.

Flushing, the reverse operation, can be executed without explicit agreement of the mappees, since they agreed implicitly when accepting the prior map operation. $S$ can *flush* any of its pages:

$$\forall_{\sigma'_{v'} = (\sigma, v)} : \sigma'_{v'} \leftarrow \phi \quad .$$

Note that $\leftarrow$ and *flush* are defined recursively. Flushing recursively affects also all mappings which are indirectly derived from $\sigma_v$.

No cycles can be established by these three operations, since $\leftarrow$ flushes the destination prior to copying.

### Implementing the Model

At a first glance, deriving the physical address of page $v$ in address space $\sigma$ seems to be rather complicated and expensive:

$$\sigma(v) = \begin{cases} \sigma'(v') & \text{if } \sigma_v{=}(\sigma', v') \\ r & \text{if } \sigma_v{=}r \\ \phi & \text{if } \sigma_v{=}\phi \end{cases}$$

Fortunately, a recursive evaluation of $\sigma(v)$ is never required. The three basic operations guarantee that the physical address of a virtual page will never change, except by flushing. For implementation, we therefore complement each $\sigma$ by an additional table $P$, where $P_v$ corresponds to $\sigma_v$ and holds either the physical address of $v$ or $\phi$. Mapping and granting then include

$$P'_{v'} \ := \ P_v$$

and each replacement $\sigma_v \leftarrow \phi$ invoked by a flush operation includes

$$P_v \ := \ \phi \quad .$$

$P_v$ can always be used instead of evaluating $\sigma(v)$. In fact, $P$ is equivalent to a hardware page table. $\mu$-kernel address spaces can be implemented straightforward by means of the hardware-address-translation facilities.

The recommended implementation of $\sigma$ is to use one mapping tree per physical page frame which describes all actual mappings of the frame. Each node contains $(P, v)$, where $v$ is the according virtual page in the address space which is implemented by the page table $P$.

Assume that a grant-, map- or flush-operation deals with a page $v$ in address space $\sigma$ to which the page table $P$ is associated. In a first step, the operation selects the according tree by $P_v$, the physical page. In the next step, it selects the node of the tree that contains $(P, v)$. (This selection can be done by parsing the tree or in a single step, if $P_v$ is extended by a link to the node.) Granting then simply replaces the values stored in the node and map creates a new child node for storing $(P', v')$. Flush lets the selected node unaffected but parses and erases the complete subtree, where $P'_v \ := \ \phi$ is executed for each node $(P', v')$ in the subtree.

## 1.1.2 Threads and IPC

A *thread* is an activity executing inside an address space. A thread $\tau$ is characterised by a set of registers, including at least an instruction pointer, a stack pointer and a state information. A thread's state also includes the address space $\sigma^{(\tau)}$ in which $\tau$ currently executes. This dynamic or static association to address spaces is the decisive reason for including the thread concept (or something equivalent) in the $\mu$-kernel. To prevent corruption of address spaces, all changes to a thread's address space ($\sigma^{(\tau)} \ := \ \sigma'$) must be controlled by the kernel. This implies that the $\mu$-kernel includes the notion of some $\tau$ that represents the above mentioned activity. In some operating systems, there may be additional reasons for introducing threads as a basic abstraction, e.g. preemption. Note that choosing a concrete thread concept remains subject to further OS-specific design decisions.

Consequently, cross-address-space communication, also called inter-process communication (IPC), must be supported by the $\mu$-kernel. The classical method is transferring messages between threads by the $\mu$-kernel.

IPC always enforces a certain agreement between both parties of a communication: the sender decides to send information and determines its contents; the receiver determines whether it is willing to receive information and is free to interpret the received message. Therefore, IPC is not only the basic concept for communication between subsystems but also, together with address spaces, the foundation of independence.

Other forms of communication, remote procedure call (RPC) or controlled thread migration between address spaces, can be constructed from message-transfer based IPC.

Note that the *grant* and *map* operations (section 1.1.1) need IPC, since they require an agreement between granter/mapper and recipient of the mapping.

### Interrupts

The natural abstraction for hardware interrupts is the IPC message. The hardware is regarded as a set of threads which have special thread ids and send empty messages (only consisting of the sender id) to associated software threads. A receiving thread concludes from the message source id, whether the message comes from a hardware interrupt and from which interrupt:

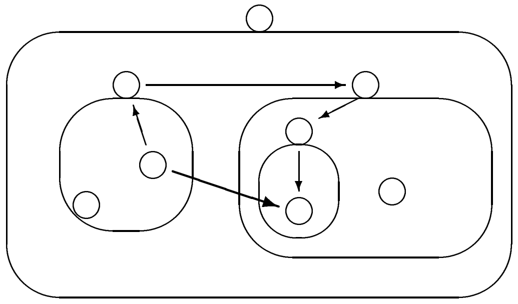
```
driver thread:
    do
        wait for (msg, sender) ;
        if sender = my hardware interrupt
            then   read/write io ports ;
                   reset hardware interrupt
            else   . . .
        fi
    od .
```

Transforming the interrupts into messages must be done by the kernel, but the $\mu$-kernel is not involved in device-specific interrupt handling. In particular, it does not know anything about the interrupt semantics. On some processors, resetting the interrupt is a device specific action which can be handled by drivers at user level. The `iret`-instruction then is used solely for popping status information from the stack and/or switching back to user mode and can be hidden by the kernel. However, if a processor requires a privileged operation for releasing an interrupt, the kernel executes this action implicitly when the driver issues the next IPC operation.

### 1.1.3 Clans & Chiefs

Within all systems based on direct message transfer, protection is essentially a matter of message control. Using access control lists (acl) this can be done at the server level, but maintenance of large distributed acls becomes hard when access rights change rapidly. So [HKK93] have proposed that object (passive entity) protection be complemented by subject (active entity) restrictions. In this approach the kernel is able to restrict the outgoing messages of a task (the subject) by means of a list of permitted receivers.

The clan concept [Lie92] is an algorithmic generalisation of this idea:



A *clan* (denoted as an oval) is a set of tasks (denoted as a circle) headed by a *chief* task. Inside the clan all messages are transferred freely and the kernel guarantees message integrity. But whenever a message tries to cross a clan's borderline, regardless of whether it is outgoing or incoming, it is redirected to the clan's chief. This chief may inspect the message (including the sender and receiver ids as well as the contents) and decide whether or not it should be passed to the destination to which it was addressed. As demonstrated in the figure above, these rules apply to nested clans as well. Obviously subject restrictions and local reference monitors can be implemented outside the kernel by means of clans. Since chiefs are tasks at user level, the clan concept allows more sophisticated and user definable checks as well as active control. Typical clan structures are

**Clan per machine:** In this simple model there is only one clan per machine covering all tasks. Local communication is handled directly by the kernel without incorporating a chief, whereas cross machine communication involves the chief of the sending and the receiving machine. Hence, the clan concept is used for implementing remote ipc by user level tasks.

**Clan per system version:** Sometimes chiefs are used for adapting different versions. The servers of the old or new versions are encapsulated by a clan so that its chief can translate the messages.

**Clan per user:** Surrounding the tasks of each user or user group by a clan is a typical method when building security systems. Then the chiefs are used to control and enforce the requested security policy.

**Clan per task:** In the extreme case there are single tasks each controlled by a specific chief. For example these one-task-clans are used for debugging and supervising suspicious programs.

In the case of intra-clan communication (no chief involved), the additional costs of the clan concept are negligible (below 1% of minimal ipc time). Inter-clan communication however multiplies the ipc operations by the number of chiefs involved. This can be tolerated, since (i) L4 ipc is very fast (see above) and (ii) crossing clan boundaries occurs seldom enough in practice. Note that many security policies can be implemented simply by checking the client id in the server and do not need clans.

### 1.1.4 Unique Identifiers

A $\mu$-kernel must supply unique identifiers (uid) for something, either for threads or tasks or communication channels. Uids are required for reliable and efficient local communication. If $S_1$ wants to send a message to $S_2$, it needs to specify the destination $S_2$ (or some channel leading to $S_2$). Therefore, the $\mu$-kernel must know which uid relates to $S_2$. On the other hand, the receiver $S_2$ wants to be sure that the message comes from $S_1$. Therefore the identifier must be unique, both in space and time.

In theory, cryptography could also be used. In practice, however, enciphering messages for local communication is far too expensive and the kernel must be trusted anyway. $S_2$ can also not rely on purely user-supplied capabilities, since $S_1$ or some other instance could duplicate and pass them to untrusted subsystems without control of $S_2$.

### 1.1.5 Flexibility

To illustrate the flexibility of the basic concepts, we sketch some applications which typically belong to the basic operating system but can easily be implemented on top of the $\mu$-kernel.

**Memory Manager.** A server managing the initial address space $\sigma_0$ is a classical main memory manager, but outside the $\mu$-kernel. Memory managers can easily be stacked: $M_0$ maps or grants parts of the physical memory ($\sigma_0$) to $\sigma_1$, controlled by $M_1$, other parts to $\sigma_2$, controlled by $M_2$. Now we have two coexisting main memory managers.

**Pager.** A Pager may be integrated with a memory manager or use a memory managing server. Pagers use the $\mu$-kernel's grant, map and flush primitives. The remaining interfaces, pager – client, pager – memory server and pager – device driver, are completely based on IPC and are user-level defined.

Pagers can be used to implement traditional paged virtual memory and file/database mapping into user address spaces as well as unpaged resident memory for device drivers and/or real time systems. Stacked pagers, i.e. multiple layers of pagers, can be used for combining access control with existing pagers or for combining various pagers (e.g. one per disk) into one composed object. User-supplied paging strategies [LCC94, CFL94] are handled at the user level and are in no way restricted by the $\mu$-kernel. Stacked file systems [KN93] can be realized accordingly.

**Multimedia Resource Allocation.** Multimedia and other real-time applications require memory resources to be allocated in a way that allows predictable execution times. The above mentioned user-level memory managers and pagers permit e.g. fixed allocation of physical memory for specific data or locking data in memory for a given time.

Note that resource allocators for multimedia and for timesharing can coexist. Managing allocation conflicts is part of the servers' jobs.

**Device Driver.** A device driver is a process which directly accesses hardware I/O ports mapped into its address space and receives messages from the hardware (interrupts) through the standard IPC mechanism. Device-specific memory, e.g. a screen, is handled by means of appropriate memory managers. Compared to other user-level processes, there is nothing special about a device driver. No device driver has to be integrated into the $\mu$-kernel.[2]

**Second Level Cache and TLB.** Improving the hit rates of a secondary cache by means of page allocation or reallocation [KH92, RLBC94] can be implemented by means of a pager which applies some cache-dependent (hopefully conflict reducing) policy when allocating virtual pages in physical memory.

In theory, even a software TLB handler could be implemented like this. In practice, the first-level TLB handler will be implemented in the hardware or in the $\mu$-kernel. However, a second-level

[2]In general, there is no reason for integrating boot drivers into the kernel. The booter, e.g. located in ROM, simply loads a bit image into memory that contains the micro-kernel and perhaps some set of initial pagers and drivers (running in user mode and *not* linked but simply appended to the kernel). Afterwards, the boot drivers are no longer used.

TLB handler, e.g. handling misses of a hashed page table, might be implemented as a user-level server.

**Remote Communication.** Remote IPC is implemented by communication servers which translate local messages to external communication protocols and vice versa. The communication hardware is accessed by device drivers. If special sharing of communication buffers and user address space is required, the communication server will also act as a special pager for the client. The $\mu$-kernel is not involved.

**Unix Server.** Unix[3] system calls are implemented by IPC. The Unix server can act as a pager for its clients and also use memory sharing for communicating with its clients. The Unix server itself can be page-able or resident.

**Conclusion.** A small set of $\mu$-kernel concepts lead to abstractions which stress flexibility, provided they perform well enough. The only thing which cannot be implemented on top of these abstractions is the processor architecture, registers, first-level caches and first-level TLBs.

[3]Unix is a registered trademark of UNIX System Laboratories.

## 1.2 Data Types

### 1.2.1 Unique Ids

Unique ids identify tasks, threads and hardware interrupts. They are also unique in time. Unique ids are 64-bit values.

### 1.2.2 User-Level Operations on Uids

| | |
|---|---|
| a = b : | a = b |
| task(a) = task (b) : | (a AND NOT lthread mask) = (b AND NOT lthread mask) |
| chief(a) = chief (b) : | (a AND NOT chief mask) = (b AND NOT chief mask) |
| site(a) = site (b) : | (a AND NOT site mask) = (b AND NOT site mask) |
| lthread no(a) : | (a AND lthread mask) SHR lthread shift<br>*extract lthread no from thread id* a |
| thread(a,n) : | (a AND NOT lthread mask) + (n SHL lthread shift)<br>*construct thread id from task id* a *and lthread no* n |
| task no(a) : | (a AND task mask) SHR task shift |
| chief no(a) : | (a AND chief mask) SHR chief shift |
| site no(a) : | (a AND site mask) SHR site shift |

### 1.2.3 Fpages

Fpages (Flexpages) are regions of the virtual address space. An fpage consists of all pages actually mapped in this region. The minimal fpage size is the minimal hardware-page size.

An fpage of size $2^s$ has a $2^s$-aligned base address $b$, i.e. $b \bmod 2^s{=}0$. An fpage with base address $b$ and size $2^s$ is denoted by the 64-bit value

$$b + 4s.$$

On R4x00 processors, the smallest possible value for $s$ is 12, since the hardware page size is 4K. On Alpha processors, the smallest possible value for $s$ is 13, since the default hardware page size is 8K.

### 1.2.4 Messages

S :: snd ; EMPTY .

R :: rcv ; EMPTY .

EMPTY :: .

| | | | |
|---|---|---|---|
| S R message: | rcv fpage option , | | |
| | size dope , | | |
| | S R msg dope , | | |
| | S R mwords , | | |
| | S R string dopes . | | |
| rcv fpage option: | rcv fpage:fpage ; | | |
| | zero:word. | | |
| size dope: | reserved:byte , | | |
| | string dope number:5bits , | $= S$ | |
| | mwords number:19bits . | $= W$ | |
| snd R msg dope: | undefined:byte , | | |
| | string dope number:5bits , | $= s$ | $s \leq S$ |
| | mwords number:19bits . | $= w$ | $w \leq W$ |
| rcv msg dope: | undefined:word . | | |
| snd R mwords: | $w \times$ send receive word , | | |
| | $(W - w) \times$ receive word ; | | |
| | $m \times$ snd fpage receive double word , | $2m \leq w$ | |
| | $w - 2m \times$ send receive words , | | |
| | $(W - w) \times$ receive word . | | |
| rcv mwords: | $W \times$ receive word . | | |
| snd R string dopes: | $s \times$ snd R string dope , | | |
| | $(S - s) \times$ R string dope . | | |
| rcv string dopes: | $S \times$ rcv string dope . | | |
| snd rcv string dope: | snd addr:word , | | |
| | snd size:word , | $\leq$ 4MB | |
| | rcv addr:word , | | |
| | rcv size:word . | $\leq$ 4MB | |
| snd string dope: | snd addr:word , | | |
| | snd size:word , | $\leq$ 4MB | |
| | undefined:word , | | |
| | undefined:word . | | |
| rcv string dope: | undefined:word , | | |
| | undefined:word , | | |
| | rcv addr:word , | | |
| | rcv size:word . | $= s_r$ | $s_r \leq$ 4MB |

```
snd map fpage:      grant flag:1bit ,
                    write flag:1bit ,
                    snd base:30bits ,
                    snd fpage:fpage .
```

## 1.3 $\mu$-Kernel Calls

| | |
|---|---|
| **ipc** | (dest option, snd descriptor option, rcv descriptor option, timeouts) $\longrightarrow$ (source option, result code) |
| CALL | (dest, snd descriptor, closed rcv descriptor, timeouts) $\longrightarrow$ (dest option, result code) |
| REPLY&WAIT | (dest, snd descriptor, open rcv descriptor, timeouts) $\longrightarrow$ (source option, result code) |
| SEND | (dest, snd descriptor, –nil– , timeouts) $\longrightarrow$ ($\sim$, result code) |
| RECEIVE FROM | (source, –nil– , closed rcv descriptor, timeouts) $\longrightarrow$ (source option, result code) |
| WAIT | ($\sim$, –nil– , open rcv descriptor, timeouts) $\longrightarrow$ (source option, result code) |
| RECEIVE INTR | (intr, –nil– , closed rcv descriptor, timeouts) $\longrightarrow$ (source option, result code) |
| SLEEP | (–nil– , –nil– , closed rcv descriptor, timeouts) $\longrightarrow$ ($\sim$, result code) |
| **id_nearest** | (dest id) $\longrightarrow$ (nearest id) |

| | |
|---|---|
| **fpage_unmap** | (fpage, map mask) $\rightarrow$ () |
| **thread_switch** | (dest) $\rightarrow$ () |
| **lthread_ex_regs** | (lthread no, SP, IP, excpt, pager) $\rightarrow$ (FLAGS, SP, IP, excpt, pager) |
| **thread_schedule** | (dest, prio, timeslice, ext preempter) $\rightarrow$ (prio, timeslice, state, ext preempter, partner, time) |
| **task_new** | (dest task id, mcp/new chief, SP, IP, pager id, excpt id) $\rightarrow$ (new task id) |

# Chapter 2

# L4/Alpha

## L4/Alpha 21064, 21164, 21264

## 2.1 Implementation Notes

What follows is a list of implementation details of the current L4/Alpha implementation. It is here to serve as a quick reference as to what may or may not be implemented for those that are familiar with L4/x86 or L4/MIPS.

### 2.1.1 IPC

- Granting is not supported.
- Qwords sent in memory based messages are 64-bit, not 32-bit as in L4/x86. This allows sending direct messages of up to 8MB in size.
- Indirect strings are not currently supported.

### 2.1.2 Scheduling

- The current scheduler uses a multi-level round robin scheme with absolute priorities (0-255).
- Internal and external preempters are not supported.
- Constant interrupts will prevent other threads from running.

### 2.1.3 $\sigma_0$

- Multiple mappings of the same physical frame is not supported.
- The RPC protocol is slightly different, see section 2.8 for details.
- $\sigma_0$is runs in user mode rather than kernel mode.

### 2.1.4 Exceptions

Exceptions are handled using an interrupt descriptor table (IDT). Each thread may have it's own IDT, see section 2.5 for details.

### 2.1.5 Interrupts

Interrupts are handled using IPC. Each level of interrupt may have an associated handler thread, see the ipc part of section 2.4 for how this is done. When an interrupt occurs, the associate handler (if any) is notified via ipc.

## 2.2 Notational conventions

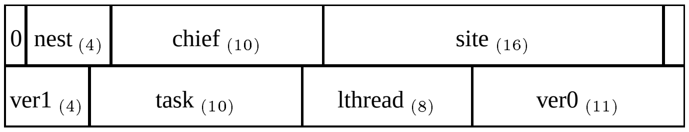
| | |
|---|---|
| $\sim$ | If this refers to an input parameter, its value is meaningless. If it refers to an output parameter, its value is undefined. |
| **a0**,**a1**... | denote the processor's general registers. |

## 2.3 Data Types

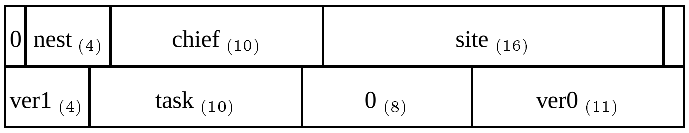
### 2.3.1 Unique Ids

Unique ids identify tasks, threads and hardware interrupts. Each unique id is a 64-bit value which is unique in time. A unique id consists of a single 64-bit word:

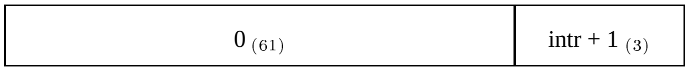
*thread id*



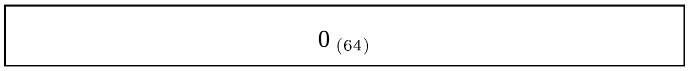
*task id*



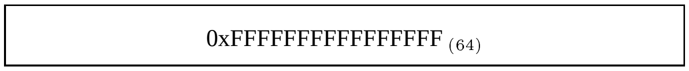
*interrupt id*



*nil id*



*invalid id*



### 2.3.2 Fpages

Fpages (Flexpages) are regions of the virtual address space. An fpage designates all pages actually mapped in this region. The minimal fpage size is 8 K, the minimal hardware-page size.

An fpage of size $2^s$ has a $2^s$-aligned base address $b$, i.e. $b \bmod 2^s = 0$. On the Alpha processors, the smallest possible value for $s$ is 13, since hardware pages are at least 8K. The complete user address space (base address 0, size $2^{64} - K$, where $K$ is the size of the kernel area) is denoted by $b = 0, s = 64$. An fpage with base address $b$ and size $2^s$ is denoted by a 64-bit word:

| | | | | | |
|---|---|---|---|---|---|
| fpage$(b, 2^s)$ | $b/8192$ $_{(51)}$ | 0 $_{(5)}$ | $s$ $_{(6)}$ | ~ | ~ |
| fpage$(0, 2^{64} - K)$ | 0 $_{(51)}$ | 0 $_{(5)}$ | 63 $_{(6)}$ | ~ | ~ |

**Alpha Implementation Note:** The user address space on the Alpha 21064 and 21164 is upto eight terabytes ($2^{43}$) beginning at 0x0. Newer processors such as the 21264 support upto 256 terabytes ($2^{48}$). L4/Alpha will detect and use the maximum available virtual address space size.

### 2.3.3 Messages

A message contains between $2^6$ and $2^{22} + 2^6$ bytes of in-line data (**mwords**). The first 64 bytes (eight qwords) are transfered via registers, the (optional) remainder is contained in a qword-aligned memory buffer pointed to by a *message descriptor*. Every successful IPC operation will always copy at least eight qwords to the receiver.

The buffer pointed to by the optional message descriptor contains a 3 qword message header, followed by a number of mwords, followed by a number of *string dopes*. The number of mwords (in 64-bit qwords, excluding those copied in registers) and string dopes is specified in the message header.

| | |
|---|---|
| | string dopes |
| | mwords |
| message: | msg header |

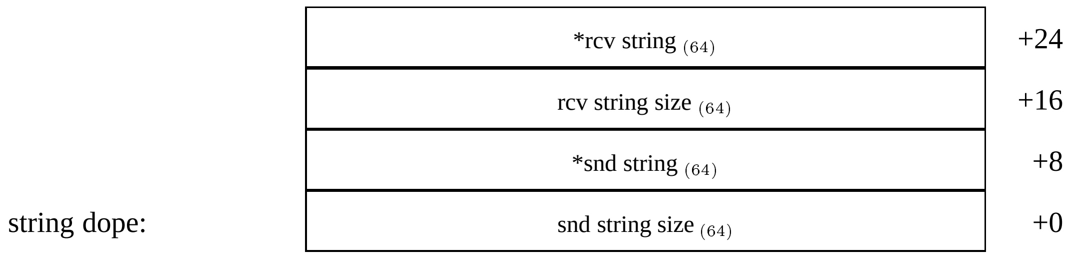
The beginning of the message buffer has the following format:

| | | | | | |
|---|---|---|---|---|---|
| | ⋮ | | | | |
| | qword 1 $_{(64)}$ | | | | +32 |
| mwords: | qword 0 $_{(64)}$ | | | | +24 |
| msg snd dope: | 0 $_{(32)}$ | mwords $_{(19)}$ | strings $_{(5)}$ | ~ $_{(8)}$ | +16 |
| msg size dope: | 0 $_{(32)}$ | mwords $_{(19)}$ | strings $_{(5)}$ | ~ $_{(8)}$ | +8 |
| msg rcv fpage option: | fpage $_{(64)}$ | | | | +0 |

The *receive fpage* describes the address range in which the caller is willing to accept fpage mappings or grants in the receive part (if any) of the IPC. The *size dope* defines the size (in dwords) of the mword buffer (and hence the offset of the string dopes from the end of the header), and the number of string dopes.

The *send dope* specifies how many dwords and strings are actually to be sent. (Specifying send dope values less than the size dope values makes sense when the caller is willing to receive more data than sending.)

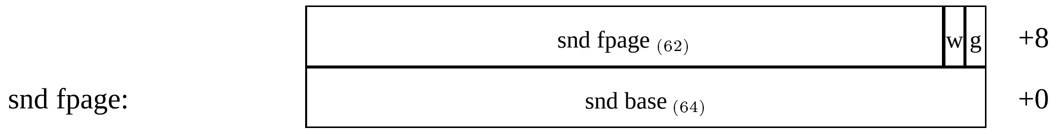
*Strings* are out-of-line by-value data. Their size and location is specified by the corresponding string dopes. The string dope format is:



The first part of the string dope specifies the size and location of the string the caller wants sent to the destination, while the second part specifies the size and location of a buffer where the caller is willing to receive a string. **Note** that strings do not have to be aligned, and that their size is specified in *bytes*.

The in-line part of the message consists of the eight qwords passed in registers followed by any dwords specified by the message descriptor. This part consists of optional *fpage descriptors* followed by by-value data. If the receiver of an IPC has specified a valid *receive fpage*, the kernel will interpret each pair of dwords of the in-line part as fpage descriptors, until an invalid descriptor is encountered. This and any further dwords are then passed by value.

The format of an fpage descriptor is:



The first word contains the address of the *hot spot*, while the second word describes the sender's fpage in the format given in Sect. 2.3.2. The $g$-bit, if set, indicates that the fpage is to be *granted* to the receiver, otherwise it is just *mapped*. The $w$-bit indicates whether the receiver will be given write or read-only access to the address-space region.

Each fpage specified by the sender is mapped individually into the address-space window specified by the receiver's *receive fpage*. If the sender and receiver specify different fpage sizes, the hot-spot specification is used to determine how the mapping between the two different size fpages occurs: If $2^s$ is the size of the larger, and $2^t$ the size of the smaller fpage, then the larger fpage can be thought as being tiled by $2^{s-t}$ fpages of the smaller size. One of these is uniquely identified as containing the hot spot address (mod $2^s$). This is the fpage which will actually be mapped.

The kernel refuses to map or grant a page over an existing mapping in the receiver's address space; an attempt to perform such a mapping will be treated as a no-op. There is one exception: If the receiver's page is already mapped to the same page as the present IPC attempts to map it (i.e. the mapping operation would not change the association of the receiver's page with physical memory) then the write permission on the receiver's page is set according to the $w$-bit of the sender's fpage descriptor. In other words, such an operation can be used to change a mapping from R/O to writable or vice versa.

If the sender provides several fpage descriptors which attempt to define conflicting mappings for one of the receiver's pages, the result is undefined.

### 2.3.4 Timeouts

Timeouts are used to control ipc operations. The *send timeout* determines how long ipc should try to send a message. If the specified period is exhausted without that message transfer could start, ipc fails. The *receive timeout* specifies how long ipc should wait for an incoming message. Both timouts specify the maximum period of time *before message transfer starts*. Once started, message transfer is no longer influenced by send or receive timeout.

Pagefaults occuring during ipc are controlled by *send* and *receive pagefault timeout*. A pagefault is translated to an RPC by the kernel. In the case of a pagefault in the receiver's address space, the corresponding RPC to the pager uses *send pagefault timeout* (specified by the sender) for both send and receive timeout. In the case of a pagefault in the sender's address space, *receive pagefault timeout* specified by the receiver is taken.

Besides the special timeouts 0 (do not wait at all) and $\infty$ (wait forever), periods from 1 $\mu$s up to approximately 19 hours can be specified. The complete quadruple is packed into one 64-bit word:

| | | | | | |
|---|---|---|---|---|---|
| $m_s$ $_{(16)}$ | $p_s$ $_{(8)}$ | $e_s$ $_{(8)}$ | $m_r$ $_{(16)}$ | $p_r$ $_{(8)}$ | $e_r$ $_{(8)}$ |

$$\text{snd timeout} = \begin{cases} \infty & \text{if} \quad e_s{=}0 \\ 4^{15-e_s} m_s\ \mu s & \text{if} \quad e_s{>}0 \\ 0 & \text{if} \quad m_s{=}0, e_s{\neq}0 \end{cases}$$

$$\text{rcv timeout} = \begin{cases} \infty & \text{if} \quad e_r{=}0 \\ 4^{15-e_r} m_r\ \mu s & \text{if} \quad e_r{>}0 \\ 0 & \text{if} \quad m_r{=}0, e_r{\neq}0 \end{cases}$$

$$\text{snd pagefault timeout} = \begin{cases} \infty & \text{if} \quad p_s{=}0 \\ 4^{15-p_s}\mu s & \text{if} \quad 0{<}p_s{<}15 \\ 0 & \text{if} \quad p_s{=}15 \end{cases}$$

$$\text{rcv pagefault timeout} = \begin{cases} \infty & \text{if} \quad p_r{=}0 \\ 4^{15-p_r}\mu s & \text{if} \quad 0{<}p_r{<}15 \\ 0 & \text{if} \quad p_r{=}15 \end{cases}$$

| approximate timeout ranges | | |
|---|---|---|
| $e_s, e_r, p_s, p_r$ | snd/rcv timeout | pf timeout |
| 0 | $\infty$ | $\infty$ |
| 1 | 256 s … 19 h | 256 s |
| 2 | 64 s … 55 h | 64 s |
| 3 | 16 s … 71 m | 16 s |
| 4 | 4 s … 17 m | 4 s |
| 5 | 1 s … 4 m | 1 s |
| 6 | 262 ms … 67 s | 256 ms |
| 7 | 65 ms … 17 s | 64 ms |
| 8 | 16 ms … 4 s | 16 ms |
| 9 | 4 ms … 1 s | 4 ms |
| 10 | 1 ms … 261 ms | 1 ms |
| 11 | 256 $\mu$s … 65 ms | 256 $\mu$s |
| 12 | 64 $\mu$s … 16 ms | 64 $\mu$s |
| 13 | 16 $\mu$s … 4 ms | 16 $\mu$s |
| 14 | 4 $\mu$s … 1 ms | 4 $\mu$s |
| 15 | 1 $\mu$s … 255 $\mu$s | 0 |
| $m=0, e>0$ | 0 | — |

**Alpha Implementation Note:** Timeouts presently have millisecond granularity. Specified microsecond timeouts are therefore rounded down (truncated) to the nearest millisecond, say $x$. Actual timeout will then occur anywhere in the interval $[ct + x - 1, ct + x]$, where $ct$ is the current time in milliseconds.

## 2.4 $\mu$-Kernel Calls

System calls are implemented as separate Alpha PALcode entry points known as `PALcall`. All registers, unless otherwise stated, are returned undefined after the system call except for the stack pointer **sp**.

This section describes the 7 system calls of L4:

| | |
|---|---|
| • ipc | **PALcall** 0xB0 |
| • id_nearest | **PALcall** 0xB2 |
| • fpage_unmap | **PALcall** 0xB1 |
| • thread_switch | **PALcall** 0xB3 |
| • thread_schedule | **PALcall** 0xB4 |
| • lthread_ex_regs | **PALcall** 0xB5 |
| • task_new | **PALcall** 0xB6 |

For exception handling on L4/Alpha, these 2 system calls are described in this section also:

| | |
|---|---|
| • l4_config | **PALcall** 0xB7 |
| • l4_return | **PALcall** 0xB8 |

The following system calls have been added for use in SMP systems.

| | |
|---|---|
| • l4_migrate | **PALcall** 0xAC |
| • l4_whoami | **PALcall** 0xAB |

Miscellaneous system calls:

| | |
|---|---|
| • sys_time | **PALcall** 0xB9 |

## ipc

| | | | | | |
|---|---|---|---|---|---|
| *dest id / wait for id* | **a0** | | **a0** | ∼ | |
| *snd descriptor* | **a1** | | **a1** | ∼ | |
| *rcv descriptor* | **a2** | | **a2** | ∼ | |
| *timeouts* | **a3** | — **PALcall** 0xB0⟶ | **a3** | ∼ | |
| *virtual sender id / ∼* | **a4** | | **a4** | *virtual sender id / ∼* | |
| *dest id / ∼* | **a5** | | **a5** | *real dest id* | |
| *msg.w0* | **t0** | | **t0** | *msg.w0* | / ∼ |
| *msg.w1* | **t1** | | **t1** | *msg.w1* | / ∼ |
| *msg.w2* | **t2** | | **t2** | *msg.w2* | / ∼ |
| *msg.w3* | **t3** | | **t3** | *msg.w3* | / ∼ |
| *msg.w4* | **t4** | | **t4** | *msg.w4* | / ∼ |
| *msg.w5* | **t5** | | **t5** | *msg.w5* | / ∼ |
| *msg.w6* | **t6** | | **t6** | *msg.w6* | / ∼ |
| *msg.w7* | **t8** | | **t8** | *msg.w7* | / ∼ |
| ∼ | **v0** | | **v0** | *msgdope + cc* | / *cc* |
| ∼ | **pv** | | **pv** | *source id* | |

This is the basic system call for inter-process communication and synchronisation. It may be used for intra- as 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 a period specified by the sender elapsed without that the destination became 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 section 2.3.3.

64-byte messages (plus 64-bit sender id) can be transferred solely via the registers and are thus specially optimised. If possible, short messages should therefore be reduced to 64-byte messages.

A single ipc call combines an optional send operation with an optional receive operation. Whether it includes a send and/or a receive is determined by the actual parameters. If the send or receive address is specified as $nil$ (0xFFFFFFFFFFFFFFFF), the corresponding operation is skipped.

No time is required for the transition between send and receive phase of one ipc operation (i.e., the destination can reply with a timeout of zero).

### Parameters

*snd descriptor*

*“nil”*

| |
|---|
| 0xFFFFFFFFFFFFFFFF $_{(64)}$ |

Ipc does not include a send operation.

*“mem”*

| | | |
|---|---|---|
| *snd msg/4 $_{(62)}$ | *m* | *d* |

Ipc includes sending a message to the destination specified by *dest id*. *snd msg must point to a valid message. The first 8 64-bit words of the message (*msg.w0* to *msg.w7*) are *not* taken from the message data structure but must be contained in registers **t0** through **t6**, **t8**.

*snd descriptor* *“reg”*

| | | |
|---|---|---|
| 0 $_{(62)}$ | $m$ | $d$ |

Ipc includes sending a message to the destination specified by *dest id*. The message consists solely of the 8 64-bit words *msg.w0* to *msg.w7* in registers **t0** through **t6**, **t8**.

$m$=0 Value-copying send operation; the qwords of the message are simply copied to the recipient.

$m$=1 Fpage-mapping send operation. The qwords of the message to be sent are treated as ’send fpages’. The described fpages are mapped or granted (depending on the $g$ bit in the fpage descriptor) /cbend into the recipient’s address space. Mapping/granting stops when either the end of the qwords is reached or when an invalid fpage denoter is found, in particular 0. The send fpage descriptors and all potentially following words are also transferred by simple copy to the recipient. Thus a message may contain some fpages and additional value parameters. The recipient can use the received fpage descriptors to determine what has been mapped or granted into its address space, including location and access rights.

$d$=0 Normal send operation. The recipient gets the true sender id.

$d$=1 Deceiving send operation. A sender can specify the *virtual sender id* which the recipient should get instead of the real sender’s id. The *virtual sender id* parameter contained in **a6** is only required if $d$=1. Recall that deceiving is secure, since only *direction-preserving deceit* is possible, see Section 1.1.3, page 12. (Note that “direction-preserving” relates to the task structure, not to threads within tasks. If a message can be sent to or from a particular thread in a task, it can also be sent to or from any other thread of the same task, and deceiving is always possible if it only changes the thread number while leaving the task ID unchanged.) If the specified *virtual-sender id* does not fulfil this constraint, the send operation works like $d$=0.

*rcv descriptor*

*“nil”*

| |
|---|
| 0xFFFFFFFFFFFFFFFF $_{(64)}$ |

Ipc does not include a receive operation.

*“mem”*

| | | |
|---|---|---|
| *rcv msg/4 $_{(62)}$ | 0 | $o$ |

Ipc includes receiving a message or waiting to receive a message. *rcv msg must point to a valid message. The 8 64-bit words of the received message (*msg.w0* to *msg.w7*) are *not* stored in the message data structure but are returned in registers **t0** through **t6**, **t8**.

*“reg”*

| | | |
|---|---|---|
| 0 $_{(62)}$ | 0 | $o$ |

Ipc includes receiving a message or waiting to receive a message. However, only messages up to 8 64-bit words *msg.w0* to *msg.w7* are accepted. The received message is returned in registers **s0** through **s7**.

*rcv descriptor*

*“rmap”*

| | | |
|---|---|---|
| rcv fpage $_{(62)}$ | 1 | $o$ |

Ipc includes receiving a message or waiting to receive a message. However, only send-fpage messages or up to 8 64-bit words *msg.w0* to *msg.w7* are accepted. The received message is returned in registers **t0** through **t6**, **t8**. If a map message is received, “rcv fpage” describes the receive fpage (instead of “rcv fpage option” in a memory message buffer). Thus fpages can also be received without a message buffer in memory.

$o$=0 Only messages from the thread specified as *dest id* are accepted (“closed wait”). Any send operation from a different thread (or hardware interrupt) will be handled exactly as if the actual thread would be busy.

$o$=1 Messages from any thread will be accepted (“open wait”). If the actual thread is associated to a hardware interrupt, also messages from this hardware interrupt can arrive.

*dest id*

Sending is directed to the specified thread, if it resides in the sender’s clan. If the destination is outside the sender’s clan, the message is sent to the sender’s chief. If the destination is in an inner clan (a clan whose chief resides in the sender’s clan), it is redirected to that chief. (See also ‘id nearest’ operation, page 38.)
This parameter is irrelevant if the ipc does not contain a send part.

| | | |
|---|---|---|
| *waiting for id* | $\neq 0, \neq nil$ | Closed receive: receiving from the specified thread (in the case of a hardware interrupt the interrupt id). This parameter is irrelevant if the ipc does not contain a receive part. |
| | $=0$ | Open receive: receiving from any sender. |
| | $=nil$ | Although specifying $nil$ as the destination for a send operation is illegal (error: ‘destination not existent’), it can be legally specified as the source of a receive-only operation. In this case, ipc will not receive any message but will wait the specified *rcv timeout* and then terminate with error code ‘receive timeout’. **Alpha Implementation Note:** “Open” and “closed” waits are specified in the *rcv descriptor* as in L4/x86. This is different to L4/MIPS. |
| *virtual sender id* | | If deceiving (*snd descriptor.*$d = 1$) this is the *source id* delivered to the receiver. The parameter is irrelevant if the IPC does not include a deveiving send part. |
| *real dest id* | | If a message was received, this is the id of the intended recipient of the message (which is different from the actual recipient if redirection took place). The parameter is undefined if no message was received. |
| *source id* | | If a message was received this is the id of its sender (or the *virtual sender id* if the message received used a deceiving send operation). If a hardware interrupt was received this is the interrupt id. The parameter is undefined if no message was received. |
| *msg.w0 . . . w7* | *“snd”* | First 8 64-bit words of message to be sent. These message words are taken directly from registers **s0** through **s7**. *They are not read from the message data structure.* |
| | *“rcv”* | First 8 64-bit words of received message, undefined if no message was received. *These message words are available only in registers **s0** through **s7**.* The $\mu$-kernel does not store it in the receive message buffer. The user program may store it or use it directly in the registers. |

*msg.dope + cc*

| | | | |
|---|---|---|---|
| 0 $_{(32)}$ | mwords $_{(19)}$ | strings $_{(5)}$ | $cc$ $_{(8)}$ |

Message dope describing received message. If no message was received, only $cc$ is delivered. *The dope word of the received message is available only in register **v0**.* The $\mu$-kernel does not store it in the receive message buffer. The user program may store it or use it directly in the register. (Note that the lowermost 8 bits of msg dope and size dope in the message data structure are undefined. So it is legal to store **v0** in the msg-dope field, even if $cc \neq 0$.)

*cc*

|  |  |  |  |  |
|---|---|---|---|---|
| $ec_{(4)}$ | $i$ | $r$ | $m$ | $d$ |

|  |  |
|---|---|
| $d$=0 | The received message is transferred directly ("undeceived") from *source id*. |
| $d$=1 | The received message is "deceived" by a chief. *source id* is the virtual source id which was specified by the sending chief. |
| $m$=0 | The received message did not contain fpages. |
| $m$=1 | The sender mapped or granted fpages. The sender's fpage descriptors were also (besides mapping/granting) transferred as mwords. |
| $r$=0 | The received message was directed to the actual recipient, not redirected to a chief. I.e. sender and receiver a part of the same clan. The $i$-bit has no meaning in this case and is zero. |
| $r$=1 | The received message was redirected to the chief which was next on the path to the true destination. Sender and addressed recipient belong to different clans. |
| $i$=0 | If $r$=1: the received message comes from outside the own clan. |
| $i$=1 | If $r$=1: the received message comes from an inner clan. |

*ec*

|  |  |
|---|---|
| = 0 | *ok*: the optional send operation was successful, and if a receive operation was also specified (*rcv descriptor* $\neq nil$) a message was also received correctly. |
| $\neq$ 0 | If ipc fails the completion code is in the range 0x10…0xF0. If the send operation fails, ipc is terminated without attemtping any receive operation. $s$ specifies whether the error occurred during the receive $(s = 0)$ operation or during the send $(s = 1)$ operation: |
| 1 | *Non-existing* destination or source. |
| $2 + s$ | *Timeout*. |
| $4 + s$ | *Cancelled* by another thread (system call lthread_ex_regs). |
| $6 + s$ | *Map failed* due to a shortage of page tables. |
| $8 + s$ | *Send pagefault timeout*. |
| $A + s$ | *Receive pagefault timeout*. |
| $C + s$ | *Aborted* by another thread (system call lthread_ex_regs or task_new). |
| $E + s$ | *Cut* message. Potential reasons are (a) the recipient's mword buffer is too small; (b) the recipient does not accept enough strings; (c) at least one of the recipient's string buffers is too small. |
| 1…5 | The respective operation was terminated before a real message transfer started. No partner was directly involved. |
| 6…F | The respective operation was terminated while a message transfer was running. The message transfer was aborted. The current partner (sender or receiver) was involved and received the corresponding error code. It is not defined which parts of the message are already transferred and which parts are not yet transferred. The source id returned to the receiver is also undefined. |

| | | |
|---|---|---|
| *timeouts* | | This 64-bit word specifies all 4 timeouts, the quadruple (*snd, rcv, snd pf, rcv pf*). For A detailed description see section 2.3.4. |
| | *“snd”* | If the required send operation cannot start transfer data within the specified time, ipc is terminated and fails with completion code ‘send timeout’ (0x18). If ipc does not include a send operation, this parameter is meaningless. |
| | *“rcv”* | If ipc includes a receive operation and no message transfer starts within the specified time, ipc is terminated and fails with completion code ‘receive timeout’ (0x20). If ipc does not include a receive operation, this parameter is meaningless. |
| | *“spf”* | If during sending data a pagefault *in the receiver’s address space* occurs, *snd pf* specified by the sender is used as send and receive timeout for the pagefault RPC. |
| | *“rpf”* | If during receiving data a pagefault *in the sender’s address space* occurs, *rcv pf* specified by the receiver is used as send and receive timeout for the pagefault RPC. |

## Basic Ipc Types

CALL

| | | | | |
|---|---|---|---|---|
| *dest id* | **a0** | | **a0** | ~ |
| **snd msg / 0* | **a1** | | **a1** | ~ |
| **rcv msg / 0* | **a2** | | **a2** | ~ |
| *timeouts* | **a3** | − **PALcall** 0xB0$\longrightarrow$ | **a3** | ~ |
| *virt sndr / ~* | **a4** | | **a4** | *virt sndr / ~* |
| *dest id* | **a5** | | **a5** | *real dest id* |
| *msg.w0* | **t0** | | **t0** | *msg.w0* |
| *msg.w1* | **t1** | | **t1** | *msg.w1* |
| *msg.w2* | **t2** | | **t2** | *msg.w2* |
| *msg.w3* | **t3** | | **t3** | *msg.w3* |
| *msg.w4* | **t4** | | **t4** | *msg.w4* |
| *msg.w5* | **t5** | | **t5** | *msg.w5* |
| *msg.w6* | **t6** | | **t6** | *msg.w6* |
| *msg.w7* | **t8** | | **t8** | *msg.w7* |
| ~ | **v0** | | **v0** | *msgdope + cc* |
| ~ | **pv** | | **pv** | *source id* |

This is the usual blocking RPC. *snd msg* is sent to *dest id* and the invoker waits for a reply from *dest id*. Messages from other sources are not accepted. Note that since the send/receive transition needs no time, the destination can reply with *snd timeout* = 0.

This operation can also be used for a server with one dedicated client. It sends the reply to the client and waits for the client's next order.

REPLY&WAIT

| | | | | |
|---|---|---|---|---|
| *dest id* | **a0** | | **a0** | ~ |
| **snd msg / 0* | **a1** | | **a1** | ~ |
| **rcv msg / 0* | **a2** | | **a2** | ~ |
| *timeouts* | **a3** | — **PALcall** 0xB0⟶ | **a3** | ~ |
| *virt sndr / ~* | **a4** | | **a4** | *virt sndr / ~* |
| *dest id* | **a5** | | **a5** | *real dest id* |
| *msg.w0* | **t0** | | **t0** | *msg.w0* |
| *msg.w1* | **t1** | | **t1** | *msg.w1* |
| *msg.w2* | **t2** | | **t2** | *msg.w2* |
| *msg.w3* | **t3** | | **t3** | *msg.w3* |
| *msg.w4* | **t4** | | **t4** | *msg.w4* |
| *msg.w5* | **t5** | | **t5** | *msg.w5* |
| *msg.w6* | **t6** | | **t6** | *msg.w6* |
| *msg.w7* | **t8** | | **t8** | *msg.w7* |
| ~ | **v0** | | **v0** | *msgdope + cc* |
| ~ | **pv** | | **pv** | *source id* |

*snd msg* is sent to *dest id* and the invoker waits for a reply from any source. This is the standard server operation: it sends a reply to the actual client and waits for the next order which may come from a different client.

SEND

| | | | | |
|---|---|---|---|---|
| *dest id* | **a0** | | **a0** | ~ |
| **snd msg / 0* | **a1** | | **a1** | ~ |
| *0xFFFFFFFFFFFFFFFF* | **a2** | | **a2** | ~ |
| *timeouts* | **a3** | — **PALcall** 0xB0⟶ | **a3** | ~ |
| *virt sndr / ~* | **a4** | | **a4** | ~ |
| *dest id* | **a5** | | **a5** | ~ |
| *msg.w0* | **t0** | | **t0** | ~ |
| *msg.w1* | **t1** | | **t1** | ~ |
| *msg.w2* | **t2** | | **t2** | ~ |
| *msg.w3* | **t3** | | **t3** | ~ |
| *msg.w4* | **t4** | | **t4** | ~ |
| *msg.w5* | **t5** | | **t5** | ~ |
| *msg.w6* | **t6** | | **t6** | ~ |
| *msg.w7* | **t8** | | **t8** | ~ |
| ~ | **v0** | | **v0** | *msgdope + cc* |
| ~ | **pv** | | **pv** | ~ |

*snd msg* is sent to *dest id*. There is no receive phase included. The invoker continues working after sending the message.

RECEIVE FROM

| | | | | |
|---|---|---|---|---|
| *source id* | **a0** | | **a0** | ~ |
| *0xFFFFFFFFFFFFFFFF* | **a1** | | **a1** | ~ |
| **rcv msg / 0* | **a2** | | **a2** | ~ |
| *timeouts* | **a3** | – **PALcall** 0xB0⟶ | **a3** | ~ |
| ~ | **a4** | | **a4** | ~ |
| ~ | **a5** | | **a5** | *real dest id* |
| ~ | **t0** | | **t0** | *msg.w0* |
| ~ | **t1** | | **t1** | *msg.w1* |
| ~ | **t2** | | **t2** | *msg.w2* |
| ~ | **t3** | | **t3** | *msg.w3* |
| ~ | **t4** | | **t4** | *msg.w4* |
| ~ | **t5** | | **t5** | *msg.w5* |
| ~ | **t6** | | **t6** | *msg.w6* |
| ~ | **t8** | | **t8** | *msg.w7* |
| ~ | **v0** | | **v0** | *msgdope + cc* |
| ~ | **pv** | | **pv** | ~ |

This operation includes no send phase. The invoker waits for a message from *source id*. Messages from other sources are not accepted. Note that also a hardware interrupt might be specified as source.

WAIT

| | | | | |
|---|---|---|---|---|
| ~ | **a0** | | **a0** | ~ |
| *0xFFFFFFFFFFFFFFFF* | **a1** | | **a1** | ~ |
| **rcv msg / 0* | **a2** | | **a2** | ~ |
| *timeouts* | **a3** | – **PALcall** 0xB0⟶ | **a3** | ~ |
| ~ | **a4** | | **a4** | *virt sndr / ~* |
| ~ | **a5** | | **a5** | *real dest id* |
| ~ | **t0** | | **t0** | *msg.w0* |
| ~ | **t1** | | **t1** | *msg.w1* |
| ~ | **t2** | | **t2** | *msg.w2* |
| ~ | **t3** | | **t3** | *msg.w3* |
| ~ | **t4** | | **t4** | *msg.w4* |
| ~ | **t5** | | **t5** | *msg.w5* |
| ~ | **t6** | | **t6** | *msg.w6* |
| ~ | **t8** | | **t8** | *msg.w7* |
| ~ | **v0** | | **v0** | *msgdope + cc* |
| ~ | **pv** | | **pv** | *source id* |

This operation includes no send phase. The invoker waits for a message from any source (including a hardware interrupt).

RECEIVE INTR

| | | | | |
|---|---|---|---|---|
| *intr + 1* | **a0** | | **a0** | ~ |
| *0xFFFFFFFFFFFFFFFF* | **a1** | | **a1** | ~ |
| **rcv msg / 0* | **a2** | | **a2** | ~ |
| *timeouts* | **a3** | − **PALcall** 0xB0→ | **a3** | ~ |
| ~ | **a4** | | **a4** | ~ |
| ~ | **a5** | | **a5** | ~ |
| ~ | **t0** | | **t0** | ~ |
| ~ | **t1** | | **t1** | ~ |
| ~ | **t2** | | **t2** | ~ |
| ~ | **t3** | | **t3** | ~ |
| ~ | **t4** | | **t4** | ~ |
| ~ | **t5** | | **t5** | ~ |
| ~ | **t6** | | **t6** | ~ |
| ~ | **t8** | | **t8** | ~ |
| ~ | **v0** | | **v0** | *msgdope + cc* |
| ~ | **pv** | | **pv** | ~ |

This operation includes no send phase. The invoker waits for an interrupt message coming from interrupt source *intr*. Note that interrupt messages come *only* from the interrupt which is currently associated with this thread and the interrupt is disabled until the next receive by the handles.

The *intr* parameter is only evaluated if *rcv timeout* = 0 is specified, see ‘associate intr’.

The interrupts available to the system vary from with processor version.

ASSOCIATE INTR

| | | | | |
|---|---|---|---|---|
| *intr + 1 \| 0x80* | **a0** | | **a0** | ~ |
| *0xFFFFFFFFFFFFFFFF* | **a1** | | **a1** | ~ |
| **rcv msg / 0* | **a2** | | **a2** | ~ |
| *timeouts* | **a3** | − **PALcall** 0xB0→ | **a3** | ~ |
| ~ | **a4** | | **a4** | ~ |
| ~ | **a5** | | **a5** | ~ |
| ~ | **t0** | | **t0** | ~ |
| ~ | **t1** | | **t1** | ~ |
| ~ | **t2** | | **t2** | ~ |
| ~ | **t3** | | **t3** | ~ |
| ~ | **t4** | | **t4** | ~ |
| ~ | **t5** | | **t5** | ~ |
| ~ | **t6** | | **t6** | ~ |
| ~ | **t8** | | **t8** | ~ |
| ~ | **v0** | | **v0** | *msgdope + cc* |
| ~ | **pv** | | **pv** | ~ |

If no (currently associated) interrupt is pending, the current thread is (1) detached from its currently associated interrupt (if any) and (2) associated to the specified interrupt provided that this one is free, i.e. not associated to another thread. If the association succeeds, the completion code is *receive timeout* (0x20) and no interrupt is received. Note that to associate an interrupt *IRQ ATTACH*

(0x80) must be or'ed with the interrupt number.

If an interrupt from the currently associated interrupt was pending, this one is delivered together with completion code *ok* (0x00); the interrupt association is *not* modified. If the requested new interrupt is already associated to another thread or is not existing, completion code *non existing* (0x10) is delivered and the interrupt association is not modified.

Dissociating an interrupt without associating a new one is done by issuing a receive from *nilthread* (0) with *rcv timeout* = 0.

SLEEP

| | | | | |
|---|---|---|---|---|
| *l4_myself()* | **a0** | | **a0** | ~ |
| *0xFFFFFFFFFFFFFFFF* | **a1** | | **a1** | ~ |
| *0* | **a2** | | **a2** | ~ |
| *timeouts* | **a3** | — **PALcall** 0xB0 ⟶ | **a3** | ~ |
| ~ | **a4** | | **a4** | ~ |
| ~ | **a5** | | **a5** | ~ |
| ~ | **t0** | | **t0** | ~ |
| ~ | **t1** | | **t1** | ~ |
| ~ | **t2** | | **t2** | ~ |
| ~ | **t3** | | **t3** | ~ |
| ~ | **t4** | | **t4** | ~ |
| ~ | **t5** | | **t5** | ~ |
| ~ | **t6** | | **t6** | ~ |
| ~ | **t8** | | **t8** | ~ |
| ~ | **v0** | | **v0** | *cc = 0x20* |
| ~ | **pv** | | **pv** | ~ |

This operation includes no send phase. Since l4_myself() is specified as source, no message can arrive and the ipc will be terminated with 'receive timeout' after the time specified by the *rcv-timeout* parameter is elapsed.

**Alpha Implementation Note:** L4/MIPS allows $invalid$ (-1) to be specified as source, however L4/Alpha at this stage does not.

## id_nearest

|  |  |  |  |  |
|---|---|---|---|---|
| *dest id* | **a0** |  | **a0** | *nearest id* |
| $\sim$ | **v0** |  | **v0** | *type* |
|  |  | — **PALcall** 0xB2$\longrightarrow$ |  |  |

If *nil* is specified as destination, the system call delivers the uid of the current thread. Otherwise, it delivers the nearest partner which would be engaged when sending a message to the specified destination. If the destination does not belong to the invoker's clan, this call delivers the chief that is nearest to the invoker on the path from the invoker to the destination.

- If the destination resides outside the invoker's clan, it delivers the invoker's own chief.
- If the destination is inside a clan or a clan nesting whose chief $C$ is direct member of the invoker's clan, the call delivers $C$.
- If the destination is a direct member of the invoker's clan, the call delivers the destination itself.
- If the destination is *nil*, the call delivers the current thread's id.

Concluding: id nearest (*dest id* ≠ *nil*) delivers exactly that partner to which the kernel would physically send a message which is targeted to *dest id*. On the other hand, a message from *dest id* would physically come from exactly this partner.

### Parameters

|  |  |  |
|---|---|---|
| *dest id* |  | Id of the destination. |
| *type* |  | Note that the *type* values correspond exactly to the completion codes of ipc. |
|  | =0 | Destination resides in the same clan. *dest id* is delivered as *nearest id*. |
|  | =$C$ | Destination is in an inner clan. The chief of this clan or clan nesting is delivered as *nearest id*. |
|  | =4 | Destination is outside the invoker's clan. The invoker's chief is delivered as *nearest id*. |
| *nearest id* |  | Either the current thread's id or the id of the nearest partner towards *dest id*. |

## fpage_unmap

| | | | | |
|---|---|---|---|---|
| *fpage + map mask* | **a0** | — **PALcall** 0xB1$\longrightarrow$ | **a0** | $\sim$ |

The specified *fpage* is unmapped in all address spaces into which the invoker mapped it directly or indirectly.

### Parameters

| | |
|---|---|
| *fpage* | Fpage to be unmapped. |
| *map mask* | see bit field below |

| | | | |
|---|---|---|---|
| $f$ | 0 $_{(61)}$ | $w$ | 0 |

| | |
|---|---|
| $w$=0 | Fpage will be partially unmapped. Already read/write mapped parts will be set to read only. Read only mapped parts are not affected. |
| $w$=1 | Fpage will be completely unmapped. |
| $f$=0 | Unmapping happens in all address spaces into which pages of the specified fpage have been mapped directly or indirectly. The *original* pages in the own task remain mapped. |
| $f$=1 | Additionally, also the original pages in the own task are unmapped (flushing). |

**SMP Implementation Note:** Flush not fully functional for SMP.

## thread_switch

*dest id* **a0** | — **PALcall** 0xB3$\longrightarrow$ | **a0** $\sim$

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

### Parameters

| | | |
|---|---|---|
| *dest id* | $= nil$ | (=0) 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. |
| | $\neq nil$ | If *dest id* 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 additionally to its ordinary scheduled timeslices.)<br>If the destination thread is not ready, the system call operates as described for *dest id* $= nil$. |

$\boxed{\textbf{SMP Implementation Note:}}$ Executing this syscall on a thread on a remote CPU acts only as a timeslice yield operation. No timeslice donation actually occurs.

## thread_schedule

| | | | | |
|---|---|---|---|---|
| *dest id* | **a0** | | **a0** | $\sim$ |
| *ext preempter* | **a1** | | **a1** | $\sim$ |
| *param word* | **a2** | | **a2** | $\sim$ |
| $\sim$ | **t0** | — **PALcall** 0xB4$\longrightarrow$ | **t0** | *partner* |
| $\sim$ | **t1** | | **t1** | *old parameter word* |
| $\sim$ | **t2** | | **t2** | *old preempter* |
| $\sim$ | **t3** | | **t3** | *current CPU* |
| $\sim$ | **v0** | | **v0** | *time* |

The system call can be used by schedulers to define the *priority*, *timeslice length* and *external preempter* of other threads. Furthermore, it delivers thread states. Note that due to security reasons, thread state information must be retrieved through the appropriate scheduler.

The system call is only effective if the current (and the new) priority of the specified destination is less or equal than the current task's *maximum controlled priority (mcp)*.

**SMP Implementation Note:** This system call affects threads on the same CPU only.

### Parameters

*dest id*

Destination thread id. The destination thread must currently exist and run on a priority level less than or equal to the current thread's *mcp*. Otherwise, the destination thread is not affected by this system call and all result parameters except *old param word* are undefined.

*param word*

| | | | | |
|---|---|---|---|---|
| valid | $m_t$ $_{(8)}$ | $e_t$ $_{(4)}$ | 0 $_{(12)}$ | prio $_{(8)}$ |

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

$m_t, e_t$ New timeslice length for the destination thread. The timeslice quantum is encoded like a timeout: $16^{e_t} m_t\ \mu s$.
The kernel rounds this value up towards the nearest possible value. Thus the timeslice granularity can be determined by trying to set the timeslice to 1 $\mu$s. Note, however, that the timeslice granularity may depend on the priority.
Timeslice length 0 ($m_t = 0, e_t \neq 0$) is always a possible value. It means that the thread will get no ordinary timeslice, i.e. is blocked. However, even a blocked thread may execute in a timeslice donated to it by ipc.

*"inv"* (0xFFFFFFFF) The current priority and timeslice length of the thread is not modified.

*ext preempter*

valid — Defines the external preempter for the destination thread. (Nilthread is a valid id.)

*“inv”* — (0xFFFFFFFF,∼) The current external preempter of the thread is not changed.

**Alpha Implementation Note:** External preempters are currently not implemented in L4/Alpha.

*old param word*

valid

| | | | | |
|---|---|---|---|---|
| $m_t$ (8) | $e_t$ (4) | $ts$ (4) | ∼ (8) | prio (8) |

*prio* — Old priority of destination thread.

$m_t, e_t$ — Old timeslice length of the destination thread: $4^{15-e_t} m_t\ \mu s$.

$ts =$ — Thread state:

| | |
|---|---|
| $0 + k$ | *Running.* The thread is ready to execute at user-level. |
| $4 + k$ | *Sending.* A user-invoked ipc send operation currently transfers an outgoing message. |
| $8 + k$ | *Receiving.* A user-invoked ipc receive operation currently receives an incoming message. |
| C | *Waiting* for receive. A user-invoked ipc receive operation currently waits for an incoming message. |
| D | *Pending* send. A user-invoked ipc send operation currently waits for the destination (recipient) to become ready to receive. |
| E | Reserved. |
| F | *Dead.* The thread is unable to execute. |

| | |
|---|---|
| $k = 0$ | *Kernel inactive.* The kernel does not execute an automatic RPC for the thread. |
| 1 | *Pager*. The kernel executes a pagefault RPC to the thread’s pager. |
| 2 | *Internal preempter.* The kernel executes a preemption RPC to the thread’s internal preempter. |
| 3 | *External preempter.* The kernel executes a preemption RPC to the thread’s external preempter. |

**Alpha Implementation Note:** Presently, $k$ is always zero in L4/Alpha.

*“inv”* — (0xFFFFFFFF) The addressed thread does either not exist or has a priority which exceeds the current thread’s *mcp*. All other return parameters are undefined (∼).

*old ext preempter*

**Alpha Implementation Note:** External preempters are currently not implemented in L4/Alpha.
Old external preempter of the destination thread.

*partner*

Partner of an active user-invoked ipc operation. This parameter is only valid if the thread’s user state is *sending*, *receiving*, *pending* or *waiting* (4. . . D). An invalid thread id (0xFFFFFFFF,∼) is delivered if there is no specific partner, i.e. if the thread is in an open receive state.

*time*

| | | | |
|---|---|---|---|
| $m_{w\ (8)}$ | $e_{w\ (4)}$ | $e_{p\ (4)}$ | $T_{high\ (16)}$ |
| $T_{low\ (32)}$ | | | |

| | |
|---|---|
| $T$ | Cpu time (48-bit value) in microseconds which has been consumed by the destination thread. |
| $m_w, e_w$ | Current user-level wakeup of the destination thread, encoded like a timeout. The value denotes the remaining timeout interval. Valid only if the user state is *waiting* (C) or *pending* (D). |
| $e_p$ | Effective pagefault wakeup of the destination thread, encoded like a 4-bit pagefault timeout. The value denotes the remaining timeout interval. Valid only if the kernel state is *pager* ($k = 1$). |

## lthread_ex_regs

| | | | | |
|---|---|---|---|---|
| *lthread no* | **a0** | | **a0** | ~ |
| *pager id* | **a1** | | **a1** | *old pager id* |
| *IP* | **a2** | | **a2** | *old IP* |
| *SP* | **a3** | — **PALcall** 0xB5⟶ | **a3** | *old SP* |

This function reads and writes some register values of a thread in the current task.

It also creates threads. Conceptually, creating a task includes creating all of its threads. Except lthread 0, all these threads run an idle loop. Of course, the kernel does neither allocate control blocks nor time slices etc. to them. Setting stack and instruction pointer of such a thread to valid values then really generates the thread.

Note that this operation reads and writes the *user-level* registers (SP and IP). Ongoing kernel activities are not affected. However an ipc operation is cancelled or aborted. If the thread is either waiting to send a message or waiting to receive a message, i.e. a message transfer is not yet running, ipc is cancelled (completion code 0x40 or 0x50). If a message transfer is currently running, ipc is aborted (completion code 0xC0 or 0xD0).

### Parameters

*lthread no*

| | |
|---|---|
| 0 $_{(57)}$ | lthread $_{(7)}$ |

Number of addressed lthread (0...127) inside the current task.

| | | |
|---|---|---|
| *SP* | valid | New stack pointer (SP) for the thread. It must point into the user-accessible part of the address space. |
| | *"inv"* | (0xFFFFFFFFFFFFFFFF) The existing stack pointer is not modified. |
| *IP* | valid | New instruction pointer (IP) for the thread. It must point into the user-accessible part of the address space. |
| | *"inv"* | (0xFFFFFFFFFFFFFFFF) The existing instruction pointer is not modified. |
| *pager* | valid | Defines the pager used by the thread. |
| | *"inv"* | (0xFFFFFFFFFFFFFFFF) The existing pager id is not modified. |
| *old SP* | | Old stack pointer (SP) of the thread. |
| *old SP* | | Old instruction pointer (IP) of the thread. |
| *old pager* | | Id of the thread's old pager. |

### Example

Signalling can be implemented as follows:

```
signal (lthread) :
    sp := receive signal stack ;
    ip := receive signal ;
    mem [sp – –] := 0 ;
    lthread ex regs (lthread, sp, ip, –, –) ;
    mem [sp – –] := ip ;
    mem [idle stack – wordlength] := sp .

receive signal :
    push all regs ;
    **while** mem [sp + 8 × wordlength] = 0 **do**
        thread switch (nilthread)
    **od** ;
    pop all regs ;
    pop (sp) ;
    jmp (signal ip) .
```

## task_new

| | | | | |
|---|---|---|---|---|
| *dest task* | **a0** | | **a0** | ~ |
| *mcp / new chief* | **a1** | | **a1** | ~ |
| *pager* | **a2** | | **a2** | ~ |
| *IP* | **a3** | — **PALcall** 0xB6⟶ | **a3** | ~ |
| *SP* | **a4** | | **a4** | ~ |
| *CPU* | **a5** | | **a5** | ~ |
| ~ | **v0** | | **v0** | *new task id* |

This function deletes and/or creates a task. Deletion of a task means that the address space of the task and all threads of the task disappear. The cputime of all deleted threads is added to the cputime of the deleting thread. If the deleted task was chief of a clan, all tasks of the clan are deleted as well.

Tasks may be created as *active* or *inactive*, as defined by the *pager* attribute. For an active task, a new address space is created together with 256 threads. Lthread 0 is started, the other ones wait for a "real" creation by lthread_ex_regs. An inactive task is empty. It occupies no resources, has no address space and no threads. Communication with inactive tasks is not possible. Loosely speaking, inactive tasks are not really existing but represent only the right to create an active task.

A newly created task gets the creator as its chief, i.e. it is created inside the creator's clan. A task can only be deleted either directly by its chief (its creator) or indirectly by (a higher-level chief) deleting the task's chief.

$\boxed{\textbf{Alpha Implementation Note:}}$ The task delete component of task_new is not currently implemented in this version of L4/Alpha. This will be fixed shortly.

### Parameters

| | | |
|---|---|---|
| *dest task* | | Task id of an *existing* task (active or inactive) whose chief is the current task. If one of these preconditions is not fulfilled, the system call has no effect. Otherwise, *dest task* is deleted and simultaneously a new task is created *with the same task number*. The new task may be active or inactive (see next parameter). |
| *pager* | ≠ *nil* | The new task is created as *active*. The specified pager is associated with lthread 0. |
| | = *nil* | (0) The new task is created as *inactive*. Lthread 0 is not created. |
| *SP* | | Initial stack pointer for lthread 0 if the new task is created as an active one. Ignored otherwise. |
| *IP* | | Initial instruction pointer for lthread 0 if the new task is created as an active one. Ignored otherwise. |
| *mcp* | | Maximum controlled priority (mcp) defines the highest priority which can be ruled by the new task acting as a scheduler. The new task's effective mcp is the minimum of the creator's mcp and the specified mcp.<br>**a1** contains this parameter, if the newly generated task is an *active* task, i.e. has a pager and at least lthread 0. |

| | | |
|---|---|---|
| *new chief* | | Specifies the chief of the new inactive task. This mechanism permits to transfer inactive (“empty”) tasks to other tasks. Transferring an inactive task to the specified chief means to transfer the related right to create a task. Note that the task number remains unchanged.<br>**a1** contains this parameter, if the newly generated task is an *inactive* task, i.e. has no pager and no threads.<br>The lthread no of the chief id is ignored, the effective chief (as far as ipc delivery is concerned) is the chief tasks’ lthread 0. |
| *new task id* | $\neq nil$ | Task creation succeeded. If the new task is active, the new task id will have a new version number so that it differs from all task ids used earlier. Chief and task number are the same as in *dest task*.<br>However, if the new task is created inactive, the chief is taken from the *chief* parameter and the task number remains unchanged. The version is in this case undefined so that the new task id might be identical with a formerly (but not currently and not in future) valid task id. This is safe since communication with inactive tasks is impossible. |
| | $= nil$ | (0) The task creation failed. |
| *CPU* | $\neq -1$ | Specifies the CPU on which lthread 0 is to be started, |
| | $= -1$ | Specifies the current CPU for lthread 0. |

## l4_config

| | | | | |
|---|---|---|---|---|
| *type* | **a0** | | **a0** | ~ |
| *IDT address* | **a1** | | **a1** | ~ |
| | | — **PALcall** 0xB7$\longrightarrow$ | | |

This function sets the address of the threads Interrupt Descriptor Table, which contains an array of entry points. When an exception occurs, the threads IP will be forced to the corresponding entry in the IDT (See Section 2.5).

### Parameters

| | |
|---|---|
| *type* | This is the type of configuration request. Currently only 0 is used. |
| *IDT address* | This is the address of the IDT table. |

## l4_return

— **PALcall** 0xB8⟶

This function is used to return to the address on the top of the user stack. This is used for returning from exception handling.

### Parameters

## l4_migrate

| | | | | |
|---|---|---|---|---|
| *thread id* | **a0** | | **a0** | ~ |
| *destination CPU* | **a1** | | **a1** | ~ |
| ~ | **v0** | | **v0** | *return code* |
| | | — **PALcall** 0xAC→ | | |

This function migrates a thread from one CPU to another.

### Parameters

| | | |
|---|---|---|
| *thread id* | | This is the thread to be migrated. |
| *destination CPU* | | This is the CPU to which the thread is to be migrated. |
| *return code* | =0 | This indicates that the migration was successful. |
| | =1 | This indicates that the migration was unsuccessful. |

## l4_whoami

|  |  |  |  |  |
|---|---|---|---|---|
| $\sim$ | **v0** |  | **v0** | *current CPU* |
|  |  | — **PALcall** 0xAB$\longrightarrow$ |  |  |

This function returns the CPU id of the current thread.

### Parameters

|  |  |
|---|---|
| *current CPU* | This is the CPU on which the thread is currently running. |

## sys_time

| | | | |
|---|---|---|---|
| ∼ | **v0** | | **v0** *ticks* |
| | | — **PALcall** 0xB9$\longrightarrow$ | |

This function returns the number of ticks since boot time for the processor the syscall is executed on.

### Parameters

| | |
|---|---|
| *ticks* | Number of ticks of run time since boot time. |

## 2.5 Exception Handling

Exceptions in L4/Alpha are handled via a per thread interrupt descriptor table (IDT).

The IDT is a structure defined as follows:

```
typedef struct {
        void (*idt_memory)(void);
        void (*idt_instruction)(void);
        void (*idt_arithmetic)(void);
        void (*idt_unaligned)(void);
        void (*idt_mchk)(void);
        void (*idt_break)(void);
} l4_alphaidt_t;
```

Each thread must call *l4_set_idt()* to register an IDT.

When a thread raises one of the above exceptions in L4/Alpha, the kernel will attempt to jump to the appropriate entry point. Before doing so, the PC is pushed onto the user's stack.

To exit from exception handling, *l4_return()* call should be used.

## 2.6 The Kernel-Info Page

The kernel-info page contains kernel-version data, memory descriptors *and the clock*. The remainder of the page is undefined. The kernel-info page is mapped *read-only* in the $\sigma_0$-address space. $\sigma_0$ can use the memory descriptors for its memory management. $\sigma_0$ can map the page read-only to other address spaces.

The kernel information page contains information useful for the initial servers to find out about the environment they were started in. Its layout is as follows.

| | | | |
|---|---|---|---|
| kernel data $_{(64)}$ | | | +40 |
| dit header $_{(64)}$ | | | +32 |
| kernel $_{(64)}$ | | | +24 |
| memory size $_{(64)}$ | | | +16 |
| clock $_{(64)}$ | | | +8 |
| build $_{(16)}$ | version $_{(16)}$ | "L4uK" $_{(32)}$ | +0 |

| | |
|---|---|
| *version* | L4 version number. |
| *build* | L4 build number of above version |
| *clock* | Number of 1 milliseconds ticks since L4 booted. |
| *memory size* | The amount of RAM installed on machine L4 is running on. |
| *kernel* | The address + 1 of last byte reserved by the kernel of low physical memory. |
| *dit header* | The address of the DIT header which maps out what was loaded with the kernel image. |
| *kernel data* | The address of the start of kernel reserved memory in the upper physical memory region. |

The physical memory initially available for applications lies between *kernel* and *kernel data*.

**Alpha Implementation Note:** *clock* field is not used on Alpha.

## 2.7 Page-Fault and Preemption RPC

### Page Fault RPC

**kernel sends:**

| | | | |
|---|---|---|---|
| w0 (**t0**) | fault address / 4 $_{(62)}$ | $w$ | ~ |
| w1 (**t1**) | faulting user-level IP $_{(64)}$ | | |

$w = 0$ Read page fault.
$w = 1$ Write page fault.

**kernel receives:** The kernel provides a receive fpage covering the complete user address space. The kernel accepts mappings or grants into this region. Only a short (i.e., register-only) message is accepted, and its contents are ignored.

| timeouts used for pagefault RPC | PF at user level | PF at ipc in receiver's space | PF at ipc in sender's space |
|---|---|---|---|
| snd | $\infty$ | sender's snd pf | receiver's rcv pf |
| rcv | $\infty$ | sender's snd pf | receiver's rcv pf |
| snd pf | $\infty$ | sender's snd pf | receiver's rcv pf |
| rcv pf | $\infty$ | sender's snd pf | receiver's rcv pf |

### Preemption RPC

**Alpha Implementation Note:** Preemption RPC is yet to be implemented in L4/Alpha.

**kernel sends:**

| | |
|---|---|
| w0 | user-level ESP $_{(32)}$ |
| w1 | user-level EIP $_{(32)}$ |

ESP and EIP are the exception stack pointer and exception instruction pointer registers, respectively.

**kernel receives:** The kernel only accepts a short (in-register) message, whose contents are ignored.

| timeouts used for preemption RPC | |
|---|---|
| snd | $\infty$ |
| rcv | $\infty$ |
| snd pf | $\infty$ |
| rcv pf | $\infty$ |

## 2.8 $\sigma_0$ RPC protocol

$\sigma_0$ is the initial address space. Although $\sigma_0$ may not be part of the kernel its basic protocol is defined by the $\mu$-kernel. Special $\sigma_0$ implementations may extend this protocol.

The address space $\sigma_0$ is idempotent, i.e. all virtual addresses in this address space are identical to the corresponding physical address. Note that pages requested from $\sigma_0$ continue to be mapped idempotently if the receiver specifies its complete address space as receive fpage.

$\sigma_0$ 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 $\sigma_0$ has spent all its memory. Further requests will then automatically denied (by sending a null reply).

### General Memory Mapping

#### Physical memory

| | |
|---|---|
| *msg.w0* (**s0**) | address $_{(64)}$ |
| *msg.w1* (**s1**) | $\sim$ $_{(64)}$ |

If *address* is in the available memory range and not previously mapped, $\sigma_0$ sends a writable mapping to the requester.

Unlike L4/x86, multiple mappings of the same physical frame is not supported, any frame is only mapped once.

#### Kernel information page

| | |
|---|---|
| *msg.w0* (**s0**) | 0xFFFFFFFFFFFFFFFD $_{(64)}$ |
| *msg.w1* (**s1**) | $\sim$ $_{(64)}$ |

Maps the kernel info page to the requester read-only. The requester receives the address of the info page in **s0** assuming a one to one mapping. Multiple mappings to multiple requesters are supported. Note that the address of the dit header can be found in the kernel information page (Sect. 2.6).

#### Dit header page

| | |
|---|---|
| *msg.w0* (**s0**) | dit header address $_{(64)}$ |
| *msg.w1* (**s1**) | $\sim$ $_{(64)}$ |

Maps the dit header page read-only. Multiple mappings to multiple requesters are supported.

## 2.9 DIT header

DIT is the tool used to build kernel images for download. Like the L4 kernel itself, it has an information page describing the layout of various programs and data that were part of the downloaded kernel image. It consists of two parts, the initial header followed by zero of more file headers as specified by the initial header. The initial headers format follows below.

| | |
|---|---|
| vaddr end $_{(32)}$ | +20 |
| file end $_{(32)}$ | +16 |
| phdr num $_{(32)}$ | +12 |
| phdr size $_{(32)}$ | +8 |
| phdr off $_{(32)}$ | +4 |
| “dhdr” $_{(32)}$ | +0 |

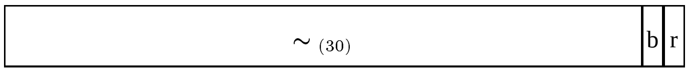
| | |
|---|---|
| *phdr off* | The offset from the beginning of this header to where the file headers start. |
| *phdr size* | The size of each of the file headers. |
| *phdr num* | The number of file headers that follow. |
| *file end* | The offset to the end of the kernel image file. For DIT internal use only. |
| *vaddr end* | The end of currently used physical memory space. This includes the L4 kernel and all other programs and data in the downloaded kernel image. |

Each of the file headers is laid out as follows.

| | |
|---|---|
| flags $_{(32)}$ | +28 |
| entry $_{(32)}$ | +24 |
| size $_{(32)}$ | +20 |
| base $_{(32)}$ | +16 |
| name string $_{(32)}$ | +12 |
| name string $_{(32)}$ | +8 |
| name string $_{(32)}$ | +4 |
| name string $_{(32)}$ | +0 |

| | |
|---|---|
| *name string* | Null terminated string containing name of program or data file (truncated to 16 characters). |

| | |
|---|---|
| *base* | The base address of the program or data file . |
| *size* | The size of the program or data. |
| *entry* | The start address of the program if it is executable, zero otherwise. |
| *flags* | Miscellaneous flags defined below. |



| | |
|---|---|
| *r* | If set the kernel runs this program as part of the initial servers upon startup. If not set, the program or data has simply been loaded into memory and has not been invoked. |
| *b* | If set this program is designated as the resource manager. In this case, the initial server is given rights to creation of new tasks. |

# Appendix A

# DITE

## A.1 Name

dite — tool for creating multi-object images

## A.2 Synopsis

**dite** [**-hVvqdl**] [[**-Rxr**] [**-b** *address*]] [**-E** *address*] *filename*] **-o** *target*

## A.3 Description

**DITE** (*Downloadable Image Tool Extended*) is a tool for combining several object files (usually executables, but it isn't picky) into a single downloadable image. This image is usually appended to a bootloader, or downloaded directly to the target machine.

**DITE** currently understands ELF and raw formats, although other formats can be added with relative ease.

## A.4 Options

**DITE** options may be either the traditional POSIX one letter options, or the GNU style long options. POSIX style options start with a single "-", while GNU long options start with "--".

### A.4.1 General Options

**`-h, --help`** Prints out a list of options and some usage examples.

**`-V, --version`** Prints out the name and version.

**`-v, --verbose`** Increases the verbosity level. For example, displays the address and offset of images in the output image.

**`-q, --quiet`** Disables any non-error output.

### A.4.2 Target Options

**`-n, --nodit`** Do not include a DIT header in the output image.

**`-t format, --target format`** Specifies the format of the output image. This takes the form *object-architecture-endianness*.

Note that partial target formats are allowed; the other information will be inferred from the first non-raw input file. If this option is not included, the output format is that of the first non-raw input file. For a complete list of available formats, execute `dite -h`.

**`-o filename, --output filename`** This specifies the output file.

**`-D segment, --dit-segment segment`** This specifies the segment in which the DIT header will be placed (starting from 0). The default segment is the last segment.

**`-O offset, --offset offset`**[1] This specifies the offset by which each segment will be moved. This does not effect any data in the DIT header.

**`-d, --dump`** This prints the contents of the DIT segment to stdout.

**`-l, --link`** Prints the next free address in the image.

**`-u, --update-sigma0`**[2] This updates the kernel info page in the second segment of the output image. This is used with `--raw` so that sigma0 knows where to find the DIT page.

### A.4.3 Source File Options

**`-a pad, --align pad`** This pads the input image with zeroes up to the specified alignment.

**`-R. --raw`** Specifies that the file will not be included in the output image as-is.

**`-e address, --entry address`** Specifies the entry point for the image. This is only required for those input formats which do not specify this information (i.e. raw).

**`-b address, --base address`** This specifies the base of the object. This is different to `--offset` in that it *does* affect the data in the DIT header. This option is primarily used with the `--raw` option.

**`-x, --execute`** This sets the execute flag in the DIT header.

**`-r, --resource`** This sets the resource flag in the DIT header.

## A.5 Notes

For output formats which contain an entry point, the entry point of the first source file is used, if the target does not exist. If the target does exist, its entry point is used.

## A.6 Examples

The following are examples of the usage of `dite`.

This is an example of how to create a file called *kernel* from a file called *kernel64*.

Note the use of the `-O` flag; the PMON bootloader uses the virtual address field rather than the physical address field to load the image. The `-O` flag specifies that the image needs to be loaded at 0xffffffff80000000 plus the normal addresses.

```
dite -t elf32-mips64-big -D 2 -O 0xffffffff80000000 -o ker-
nel kernel64
```

This command takes the file *kernel* generated in the previous example and adds the file *serial* specifying it as a initial server.

```
dite -o kernel-serial -O 0x8000000 kernel -x serial
```

The following example creates an image called *kernel-serial* containing the raw file *l4pal* with an entry address of 0x200000 and a base address of 0x20000 (i.e. it constitutes 1 segment in the output file at address 0x200000), the file *sigma0* and the file *serial*, which is an initial server and the resource manager.

```
dite -R -e 0x200000 -b 0x200000 l4pal sigma0 -xr serial -o kernel-
serial
```

## A.7 Bugs

The `-O` flag is an abomination.

On 32 bit architectures, any address larger than 0xffffffff may be incorrectly read (this is due to the use of *strtoul* to convert the string).

DITE will silently allow overlapping images.

## A.8 Authors

Simon Winwood (`sjw@cse.unsw.edu.au`)

The latest version of dite may be found at `http://www.cse.unsw.edu.au/~sjw/code/dite/`.

## A.9 See Also

`ld(1)`

# Appendix B

# L4/Alpha Kernel Debugger

The L4/Alpha kernel debugger, as it's name suggests, is used for debugging the L4 kernel itself. It is not intended to be used for debugging applications, though it can be used to if one understands enough of the internals of L4. For user level debugging, L4/Alpha supports the use of GDB.

The debugger is very primitive in functionality. It allows basic exploration of kernel data (both global and task specific data), memory, and general purpose registers.

A description of the current list of commands follows.

**`?`** Print out a short help message.

**`SPC`** Reshow entire current TCB.

**`t`** Select thread id.

**`n`** Next from present queue.

**`p`** Previous from present queue.

**`d`** Data translation buffer.

**`i`** Instruction translation buffer.

**`a`** Translate address in current thread context.

**`m`** Dump 256 bytes from current address space.

**`M`** Show mapping-tree from current task.

**`T`** Display mailboxes.

**`q`** Show associated interrupts.

**`g`** Continue at interrupt instruction.

**`h`** halt and go to console.

# Appendix C

# L4/Alpha C Library Headers

The following are the self documented header files for the C library interface to L4 on the Alpha platform. The headers contain useful constants, macros, and function prototypes for programming in the L4 environment.

## C.1 types.h

```
 /*
 * $Id: types.h,v 1.2 2000/11/29 03:50:34 danielp Exp $
 */

#ifndef __L4_TYPES_H__
#define __L4_TYPES_H__


#include <l4/compiler.h>

typedef unsigned char       byte_t;
typedef unsigned short int  word_t;
typedef unsigned int        dword_t;
typedef unsigned long long  qword_t;
typedef long long     cpu_time_t;



/*
 * L4 unique identifiers
 */

typedef struct {
  unsigned version_low:11 PACK;
  unsigned lthread:8   PACK;
  unsigned task:10        PACK;
  unsigned version_high:4 PACK;
  unsigned site:16        PACK;
  unsigned chief:10       PACK;
  unsigned nest:4         PACK; // code assumes on this order.
} l4_threadid_struct_t ;

typedef union {
  qword_t thread_id;
    qword_t ID; // Compatibility with L4/MIPS
  l4_threadid_struct_t id;
```

```
} l4_threadid_t;

typedef l4_threadid_t l4_taskid_t;

typedef unsigned long l4_cpuid_t;

typedef struct {
  unsigned MBZ1:11 PACK;
  unsigned intr:8 PACK;
  long MBZ2:45 PACK;
} l4_intrid_struct_t;

typedef union {
  qword_t intrid;
  l4_intrid_struct_t id;
} l4_intrid_t;

#define L4_NIL_ID  ((l4_threadid_t){thread_id:0x0000000000000000})
#define L4_INVALID_ID ((l4_threadid_t){thread_id:0xffffffffffffffff})

/*
 * L4 flex pages
 */

typedef struct {
  unsigned grant:1 PACK;
  unsigned write:1 PACK;
  unsigned size:6 PACK;
  unsigned zero:5 PACK;
  qword_t page:51 PACK;
} l4_fpage_struct_t;

typedef union {
  qword_t fpage;
  l4_fpage_struct_t fp;
} l4_fpage_t;

#define L4_PAGESIZE  (0x2000)
#define L4_PAGEMASK (~(L4_PAGESIZE - 1))
#define L4_LOG2_PAGESIZE (13)
#define L4_WHOLE_ADDRESS_SPACE (63)
#define L4_FPAGE_RO 0
#define L4_FPAGE_RW 1
#define L4_FPAGE_MAP 0
#define L4_FPAGE_GRANT 1

typedef struct {
  qword_t snd_base;
  l4_fpage_t fpage;
} l4_snd_fpage_t;

/*
 * L4 message dopes
 */

typedef struct {
  unsigned msg_deceited:1 PACK;
  unsigned fpage_received:1 PACK;
  unsigned msg_redirected:1 PACK;
  unsigned src_inside:1 PACK;
  unsigned snd_error:1 PACK;
  unsigned error_code:3 PACK;
  unsigned strings:5 PACK;
  qword_t dwords:51 PACK;
} l4_msgdope_struct_t;

typedef union {
  qword_t msgdope;
  l4_msgdope_struct_t md;
} l4_msgdope_t;

/*
 * L4 string dopes
 */

typedef struct {
  qword_t snd_size;
  qword_t snd_str;
  qword_t rcv_size;
  qword_t rcv_str;
} l4_strdope_t;

/*
 * L4 message header - currently unsupported.
 */

typedef struct {
    l4_fpage_t rcv_fpage;
    l4_msgdope_t size_dope;
    l4_msgdope_t snd_dope;
} l4_msghdr_t;


/*
 * L4 timeouts
 */

typedef struct {
  unsigned rcv_exp:8 PACK;
  unsigned rcv_pfault:8 PACK;
  unsigned rcv_man:16 PACK;

  unsigned snd_exp:8 PACK;
  unsigned snd_pfault:8 PACK;
  unsigned snd_man:16 PACK;

} l4_timeout_struct_t;

typedef union {
  qword_t timeout;
  l4_timeout_struct_t to;
} l4_timeout_t;

/*
 * l4_schedule param word
 */

typedef struct {
  unsigned prio:8 PACK;
  unsigned small:8 PACK;
  unsigned zero:4 PACK;
  unsigned time_exp:4 PACK;
  unsigned time_man:8 PACK;
} l4_sched_param_struct_t;

typedef union {
  qword_t sched_param;
  l4_sched_param_struct_t sp;
} l4_sched_param_t;

#define L4_SMALL_SPACE(size_mb, nr) ((size_mb >> 2) + nr * (size_mb >> 1))

/*
 * Some useful operations and test functions for id's and types
 */
#if 1
L4_INLINE int l4_is_nil_id(l4_threadid_t id);
L4_INLINE int l4_is_invalid_id(l4_threadid_t id);
L4_INLINE l4_fpage_t l4_fpage(unsigned long address, unsigned int size,
  unsigned char write, unsigned char grant);
L4_INLINE l4_threadid_t get_taskid(l4_threadid_t t);
L4_INLINE int thread_equal(l4_threadid_t t1,l4_threadid_t t2);
L4_INLINE int task_equal(l4_threadid_t t1,l4_threadid_t t2);


L4_INLINE int l4_is_nil_id(l4_threadid_t id)
{
  return id.thread_id == 0;
}

L4_INLINE int l4_is_invalid_id(l4_threadid_t id)
{
  return id.thread_id == 0xffffffffffffffff;
}

L4_INLINE l4_fpage_t l4_fpage(unsigned long address, unsigned int size,
  unsigned char write, unsigned char grant)
{
  return ((l4_fpage_t){fp:{grant, write, size, 0,
     (address & L4_PAGEMASK) >> L4_LOG2_PAGESIZE }});
}

L4_INLINE l4_threadid_t
get_taskid(l4_threadid_t t)
{
  t.id.lthread = 0;
  return t;
}

L4_INLINE int
thread_equal(l4_threadid_t t1,l4_threadid_t t2)
{
  return ((t1.thread_id == t2.thread_id));
}


L4_INLINE int
task_equal(l4_threadid_t t1,l4_threadid_t t2)
{
  return ((t1.thread_id & ~0x3ffff) ==
  (t2.thread_id & ~0x3ffff));
}

#endif
#endif /* __L4TYPES_H__ */
```

## C.2 syscalls.h

```
/*
 * $Id: syscalls.h,v 1.4.2.2 2001/03/08 02:33:38 danielp Exp $
 */

#ifndef __L4_SYSCALLS_H__
#define __L4_SYSCALLS_H__

#include <l4/compiler.h>
#include <l4/types.h>

/* For GDB */
#ifdef GDB
#ifndef BREAKDEFINE
#define BREAKDEFINE
#define breakpoint() __asm__ __volatile__ (".long 0x80") /* FIXME - use a define
  for pal 0x80 */
#endif
#endif

#define L4_FP_REMAP_PAGE 0x02 /* Page is set to read only */
#define L4_FP_FLUSH_PAGE 0x00 /* Page is flushed completly */
#define L4_FP_OTHER_SPACES 0x00 /* Page is flushed in all other */
/* address spaces */
#define L4_FP_ALL_SPACES 0x01
/* Page is flushed in own address */
/* space too */

#define L4_NC_SAME_CLAN 0x00 /* destination resides within the */
/* same clan */
#define L4_NC_INNER_CLAN 0x0C /* destination is in an inner clan */
#define L4_NC_OUTER_CLAN 0x04 /* destination is outside the */
/* invoker's clan */

#define L4_CT_LIMITED_IO 0
#define L4_CT_UNLIMITED_IO 1
#define L4_CT_DI_FORBIDDEN 0
#define L4_CT_DI_ALLOWED 1

/*
 * prototypes
 */
L4_INLINE void
l4_fpage_unmap(l4_fpage_t fpage, dword_t map_mask);

L4_INLINE void
l4_fpage_unmap_easy(qword_t VAddr, qword_t ld_pages, int bInclud-
ing, int bReadOnly);

L4_INLINE l4_threadid_t
l4_myself(void);

L4_INLINE l4_threadid_t
l4_nchief(l4_threadid_t destination, l4_threadid_t *next_chief);


L4_INLINE int
l4_id_nearest(l4_threadid_t destination, l4_threadid_t *next_chief);
```

```
L4_INLINE l4_threadid_t
l4_thread_ex_regs(l4_threadid_t destination, qword_t pc, qword_t sp,
  l4_threadid_t *preempter, l4_threadid_t *pager,
  qword_t *old_pc, qword_t *old_sp);

#ifdef GDB
L4_INLINE void
l4_thread_ex_regs_gdb(l4_threadid_t destination, qword_t pc, qword_t sp,
      l4_threadid_t *preempter, l4_threadid_t *pager,
      qword_t *old_pc, qword_t *old_sp);
#endif

L4_INLINE void
l4_thread_switch(l4_threadid_t destination);

L4_INLINE cpu_time_t
l4_thread_schedule(l4_threadid_t dest, l4_sched_param_t param,
   l4_threadid_t *ext_preempter, l4_threadid_t *partner,
   l4_sched_param_t *old_param);

L4_INLINE cpu_time_t
l4_smp_thread_schedule(l4_threadid_t dest, l4_sched_param_t param,
   l4_threadid_t *ext_preempter, l4_threadid_t *partner,
   l4_cpuid_t *cpuid, l4_sched_param_t *old_param);

L4_INLINE void
l4_schedule (l4_threadid_t dest, l4_threadid_t preempter, qword_t par2);

L4_INLINE l4_taskid_t
l4_task_new(l4_taskid_t destination, qword_t mcp_or_new_chief,
    qword_t sp, qword_t pc, l4_threadid_t pager);

L4_INLINE l4_taskid_t
l4_smp_task_new(l4_taskid_t destination, qword_t mcp_or_new_chief,
qword_t sp, qword_t pc, l4_threadid_t pager, l4_cpuid_t cpuid);

L4_INLINE void
l4_imb(void); /* John wants this, but he should just use imb() */

typedef struct {
    void (*idt_memory)(void);
    void (*idt_instruction)(void);
    void (*idt_arithmetic)(void);
    void (*idt_unaligned)(void);
    void (*idt_mchk)(void);
    void (*idt_break)(void);
} l4_alphaidt_t;


L4_INLINE void
l4_set_idt(l4_alphaidt_t *idt);

L4_INLINE unsigned long
l4_sys_time(void);

#endif
```

## C.3 ipc.h

```
/*
 * $Id: ipc.h,v 1.1.1.1.4.1 2001/03/08 02:33:38 danielp Exp $
 */

#ifndef __L4_IPC_H__
#define __L4_IPC_H__

/*
 * L4 ipc
 */

#include <l4/compiler.h>
#include <l4/types.h>

/*
 * IPC parameters
 */

/*
 * Structure used to describe destination and true source if a chief
 * wants to deceit
 */

typedef struct {
    l4_threadid_t dest, true_src;
} l4_ipc_deceit_ids_t;


#define L4_IPC_MAX_REG_MSG 8

/* lets not use l4_ipcregs_t in future
 */
typedef union {
    qword_t val[L4_IPC_MAX_REG_MSG];
    qword_t reg[L4_IPC_MAX_REG_MSG];  // compatibility with L4/MIPS
} l4_ipcregs_t;


/*typedef struct {
  qword_t reg[L4_IPC_MAX_REG_MSG];
} l4_ipc_reg_msg_t;
*/

typedef l4_ipcregs_t l4_ipc_reg_msg_t;  // compatibility with L4/MIPS

/*
 * Defines used for Parameters
 */

#define L4_IPC_SHORT_MSG  0

/*
 * Defines used to build Parameters
 */

#define L4_IPC_STRING_SHIFT 8
#define L4_IPC_DWORD_SHIFT 13
#define L4_IPC_SHORT_FPAGE ((void *)2)
```

```
#define L4_IPC_DOPE(dwords, strings) \
( (l4_msgdope_t) {md: {0, 0, 0, 0, 0, 0, strings, dwords }})


#define L4_IPC_TIMEOUT(snd_man, snd_exp, rcv_man, rcv_exp, snd_pflt, rcv_pflt)\
     ( (l4_timeout_t) \
       {to: { rcv_exp, rcv_pflt, rcv_man, snd_exp, snd_pflt, snd_man} } )

#define L4_IPC_NEVER ((l4_timeout_t) {timeout : 0})
#define L4_IPC_MAPMSG(address, size)  \
     ((void *)(qword_t)( ((address) & L4_PAGEMASK) | ((size) << 2) \
 | (unsigned long)L4_IPC_SHORT_FPAGE))

/*
 * Some macros to make result checking easier
 */

#define L4_IPC_ERROR_MASK  0xF0
#define L4_IPC_DECEIT_MASK 0x01
#define L4_IPC_FPAGE_MASK 0x02
#define L4_IPC_REDIRECT_MASK 0x04
#define L4_IPC_SRC_MASK 0x08
#define L4_IPC_SND_ERR_MASK 0x10

#define L4_IPC_IS_ERROR(x) (((x).msgdope) & L4_IPC_ERROR_MASK)
#define L4_IPC_MSG_DECEITED(x)  (((x).msgdope) & L4_IPC_DECEIT_MASK)
#define L4_IPC_MSG_REDIRECTED(x) (((x).msgdope) & L4_IPC_REDIRECT_MASK)
#define L4_IPC_SRC_INSIDE(x) (((x).msgdope) & L4_IPC_SRC_MASK)
#define L4_IPC_SND_ERROR(x) (((x).msgdope) & L4_IPC_SND_ERR_MASK)
#define L4_IPC_MSG_TRANSFER_STARTED \
((((x).msgdope) & L4_IPC_ERROR_MASK) < 5)


#define push_most \
"subq $30,8*8,$30;"\
"stq  $1, 0x00($30);stq $2, 0x08($30);stq $3, 0x10($30);stq $4, 0x18($30);"\
"stq  $5, 0x20($30);stq $6, 0x28($30);stq $7, 0x30($30);stq $27, 0x38($30);"

#define pop_most \
"ldq  $1, 0x00($30);ldq $2, 0x08($30);ldq $3, 0x10($30);ldq $4, 0x18($30);"\
"ldq  $5, 0x20($30);ldq $6, 0x28($30);ldq $7, 0x30($30);ldq $27, 0x38($30);"\
"addq $30,8*8,$30;"



/*
 * Prototypes
 */


L4_INLINE int
l4_alpha_ipc_call(l4_threadid_t dest,
  const void *snd_msg,
  const l4_ipcregs_t *snd_regs,
  void *rcv_msg,
  l4_ipcregs_t *rcv_regs,
  l4_timeout_t timeout,
       l4_msgdope_t *result);
```

```
L4_INLINE int
l4_alpha_ipc_reply_and_wait(l4_threadid_t dest,
    const void *snd_msg,
    const l4_ipcregs_t *snd_regs,
    l4_threadid_t *src,
    void *rcv_msg,
    l4_ipcregs_t *rcv_regs,
    l4_timeout_t timeout,
         l4_msgdope_t *result);

L4_INLINE int
l4_alpha_ipc_reply_deceiting_and_wait(l4_threadid_t dest,
              l4_threadid_t vsend,
      const void *snd_msg,
      l4_ipcregs_t *snd_reg,
      l4_threadid_t *src,
      void *rcv_msg,
      l4_ipcregs_t *rcv_reg,
      l4_timeout_t timeout,
      l4_msgdope_t *result);

L4_INLINE int
l4_alpha_ipc_send(l4_threadid_t dest,
  const void *snd_msg,
  const l4_ipcregs_t *snd_regs,
  l4_timeout_t timeout, l4_msgdope_t *result);

L4_INLINE int
l4_alpha_ipc_send_deceiting(l4_threadid_t dest,
                            l4_threadid_t vsend,
                            const void *snd_msg,
                            l4_ipcregs_t *snd_reg,
                            l4_timeout_t timeout,
                            l4_msgdope_t *result);

L4_INLINE int
l4_alpha_ipc_wait(l4_threadid_t *src,
  void *rcv_msg,
  l4_ipcregs_t *rcv_regs,
  l4_timeout_t timeout,
       l4_msgdope_t *result);

L4_INLINE int
l4_alpha_ipc_receive(l4_threadid_t src,
     void *rcv_msg,
     l4_ipcregs_t *rcv_regs,
     l4_timeout_t timeout,
          l4_msgdope_t *result);

/* new API calls to support knowledge of real_dst on deceit calls */
L4_INLINE int
l4_alpha_ipc_chief_wait(l4_threadid_t *src,
       l4_threadid_t *real_dst,
         void *rcv_msg,
         l4_ipc_reg_msg_t *rcv_reg,
         l4_timeout_t timeout,
         l4_msgdope_t *result);

L4_INLINE int
l4_alpha_ipc_chief_receive(l4_threadid_t src,
    l4_threadid_t *real_dst,
    void *rcv_msg,
    l4_ipc_reg_msg_t *rcv_reg,
    l4_timeout_t timeout,
    l4_msgdope_t *result);

L4_INLINE int
l4_alpha_ipc_chief_call(l4_threadid_t dest,
         l4_threadid_t vsend,
         const void *snd_msg,
         l4_ipc_reg_msg_t *snd_reg,
         l4_threadid_t *real_dst,
         void *rcv_msg,
         l4_ipc_reg_msg_t *rcv_reg,
         l4_timeout_t timeout,
         l4_msgdope_t *result);

L4_INLINE int
l4_alpha_ipc_chief_reply_and_wait(l4_threadid_t dest,
   l4_threadid_t vsend,
   const void *snd_msg,
   l4_ipc_reg_msg_t *snd_reg,
   l4_threadid_t *src,
   l4_threadid_t *real_dst,
   void *rcv_msg,
   l4_ipc_reg_msg_t *rcv_reg,
   l4_timeout_t timeout,
   l4_msgdope_t *result);

L4_INLINE int
l4_alpha_ipc_chief_send(l4_threadid_t dest,
         l4_threadid_t vsend,
         const void *snd_msg,
         l4_ipc_reg_msg_t *snd_reg,
         l4_timeout_t timeout,
         l4_msgdope_t *result);

#if 0
L4_INLINE int l4_ipc_fpage_received(l4_msgdope_t msgdope);
L4_INLINE int l4_ipc_is_fpage_granted(l4_fpage_t fp);
L4_INLINE int l4_ipc_is_fpage_writable(l4_fpage_t fp);



/*
 *
 */

L4_INLINE int l4_ipc_fpage_received(l4_msgdope_t msgdope)
{
  return msgdope.md.fpage_received != 0;
}
L4_INLINE int l4_ipc_is_fpage_granted(l4_fpage_t fp)
{
  return fp.fp.grant != 0;
}
L4_INLINE int l4_ipc_is_fpage_writable(l4_fpage_t fp)
{
  return fp.fp.write != 0;
}
```

```
/*
 * IPC results
 */

#endif

#define L4_IPC_ERROR(x) (((x).msgdope) & L4_IPC_ERROR_MASK)
#define L4_IPC_ENOT_EXISTENT 0x10
#define L4_IPC_RETIMEOUT 0x20
#define L4_IPC_SETIMEOUT 0x30
#define L4_IPC_RECANCELED 0x40
#define L4_IPC_SECANCELED 0x50
#define L4_IPC_REMAPFAILED 0x60
#define L4_IPC_SEMAPFAILED 0x70
#define L4_IPC_RESNDPFTO L4_IPC_ENTRY
#define L4_IPC_SERCVPFTO 0x90
#define L4_IPC_REABORTED 0xA0
#define L4_IPC_SEABORTED 0xB0
#define L4_IPC_REMSGCUT 0xE0
#define L4_IPC_SEMSGCUT 0xF0


/*
 * Internal defines used to build IPC parameters for the L4 kernel
 */

#define L4_IPC_NIL_DESCRIPTOR  (-1)
#define L4_IPC_DECEIT  1
#define L4_IPC_OPEN_IPC  1

#endif /* __L4_IPC__ */
```

## C.4 sigma0.h

```
/*
 * $Id: sigma0.h,v 1.2 2000/11/16 01:11:06 sjw Exp $
 *
 * Proportions of this file are taken from L4/MIPS sigma0.h. L4 on Alpha
 * attempts to make use of the same structures to make programming
 * easier.
 */

#ifndef ALPHA_SIGMA0_H
#define ALPHA_SIGMA0_H 1

#include <l4/types.h>

#define DIT_HEADER_OFFSET 0x100

/* FIXME */
#ifdef __GNUC__

#if 1
#define SIGMA0_KERNEL_INFO_MAP (0xfffffffffffffffeULL)
#else /* FIXME - currently mapped 1 - 1 */
#define SIGMA0_KERNEL_INFO_MAP  (0x222000ULL)
#endif

#define SIGMA0_DEBUG_LEVEL (0xfffffffffffffffcULL)

#define SIGMA0_TID ((l4_threadid_t) {ID: (0x80001ULL)})

#endif

typedef struct {
    dword_t magic; /* L4uK */
    word_t version;
    word_t build;
    qword_t clock;
    qword_t memory_size;
    qword_t kernel;
    qword_t dit_hdr;
    qword_t kernel_data;
} l4_kernel_info;

#endif
```

## C.5 dit.h

```
/*
 * $Id: dit.h,v 1.1.1.1 2000/10/18 05:29:15 sjw Exp $
 *
 * Proportions of this file are taken from L4/MIPS dit.h. L4 on Alpha
 * attempts to make use of the same structures to make programming
 * easier.
 */

#ifndef ALPHA_DIT_H
#define ALPHA_DIT_H

#include <l4/types.h>

#define DIT_RUN         0x01
#define DIT_MANAGER     0x02

#define DIT_NIDENT      4
#define DIT_NPNAME      16

typedef unsigned int Dit_uint;

typedef struct {
  char d_ident[DIT_NIDENT];
  Dit_uint d_phoff;
  Dit_uint d_phsize;
  Dit_uint d_phnum;
  Dit_uint d_fileend;
  Dit_uint d_vaddrend;
} Dit_Dhdr;             /* L4 DIT Header */

typedef struct {
  Dit_uint p_base;
  Dit_uint p_size;
  Dit_uint p_entry;
  Dit_uint p_flags;
  char p_name[DIT_NPNAME];
} Dit_Phdr;             /* L4 DIT File entry */

#endif
```

# Bibliography

[CFL94] P. Cao, E. W. Felton, and K. Li. Implementation and performance of application-controlled file caching. In *1st USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 165–178, Monterey, CA, 1994.

[GGKL89] M. Gasser, A. Goldstein, C. Kaufmann, and B. Lampson. The Digital distributed system security architecture. In *12th National Computer Security Conference (NIST/NCSC)*, pages 305–319, Baltimore, 1989.

[HKK93] H. Härtig, O. Kowalski, and W. Kühnhauser. The Birlix security architecture. *Journal of Computer Security*, 2(1):5–21, 1993.

[KH92] R. Kessler and M. D. Hill. Page placement algorithms for large real-indexed caches. *ACM Transactions on Computer Systems*, 10(4):11–22, November 1992.

[KN93] Y. A. Khalidi and M. N. Nelson. Extensible file systems in Spring. In *14th ACM Symposium on Operating System Principles (SOSP)*, pages 1–14, Asheville, NC, 1993.

[LCC94] C. H. Lee, M. C. Chen, and R. C. Chang. HiPEC: high performance external virtual memory caching. In *1st USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 153–164, Monterey, CA, 1994.

[Lie92] J. Liedtke. Clans & chiefs. In *12. GI/ITG-Fachtagung Architektur von Rechensystemen*, pages 294–305, Kiel, 1992. Springer.

[Lie93a] J. Liedtke. Improving IPC by kernel design. In *14th ACM Symposium on Operating System Principles (SOSP)*, pages 175–188, Asheville, NC, 1993.

[Lie93b] J. Liedtke. A persistent system in real use – experiences of the first 13 years. In *3rd International Workshop on Object Orientation in Operating Systems (IWOOOS)*, pages 2–11, Asheville, NC, 1993.

[Lie95] J. Liedtke. On $\mu$-kernel construction. In *15th ACM Symposium on Operating System Principles (SOSP)*, pages 237–250, Copper Mountain Resort, CO, 1995.

[RLBC94] T. H. Romer, D. L. Lee, B. N. Bershad, and B. Chen. Dynamic page mapping policies for cache conflict resolution on standard hardware. In *1st USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 153–164, Monterey, CA, 1994.