

# Iguana Programming Manual

---

Open Kernel Labs

DRAFT

**Document Number:** OK 40008:2007 (revision 4)  
**Software Version:** 2.1  
**Date:** April 15, 2008

**Copyright © 2007–2008 Open Kernel Labs, Inc.**

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

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

This document may not be redistributed outside your organization without prior permission.

**Contact Details:**

Open Kernel Labs Pty Ltd  
Attention: Open Kernel Labs

Suite 3, 540 Botany Road  
Alexandria, NSW 2015  
Australia

email: [enquiries@ok-labs.com](mailto:enquiries@ok-labs.com)

web: <http://www.ok-labs.com/>

# Contents

---

<b>Preface</b> . . . . .	<b>9</b>
<b>1 Introduction</b> . . . . .	<b>11</b>

## PART A — IGUANA API

<b>A-1 Iguana API</b> . . . . .	<b>15</b>
A-1.1 Memory Sections . . . . .	16
<i>A-1.1.1 Memory Types</i> . . . . .	16
<i>A-1.1.2 Memory Attributes</i> . . . . .	17
A-1.2 Threads . . . . .	17
A-1.3 Capabilities . . . . .	18
A-1.4 Protection Domains . . . . .	19
<i>A-1.4.1 Protection Domain Extensions</i> . . . . .	19
A-1.5 Zones . . . . .	20
A-1.6 External Address Spaces . . . . .	20
A-1.7 Memory Pools . . . . .	21
A-1.8 Phymem . . . . .	22
<b>A-2 Protection Management</b> . . . . .	<b>23</b>
A-2.1 Capability Lists . . . . .	23
<b>A-3 System Initialization</b> . . . . .	<b>25</b>
A-3.1 Merge command options . . . . .	25
<i>A-3.1.1 Output</i> . . . . .	25
<i>A-3.1.2 Memory Map</i> . . . . .	25
A-3.2 The Configuration File . . . . .	25
<i>A-3.2.1 The Root Program Element</i> . . . . .	26
<i>A-3.2.2 The Extension Element</i> . . . . .	26
<i>A-3.2.3 The Program Element</i> . . . . .	27
<i>A-3.2.4 The Thread Element</i> . . . . .	28
<i>A-3.2.5 The Protection Domain Element</i> . . . . .	28
<i>A-3.2.6 The Memory Section Element</i> . . . . .	29
<i>A-3.2.7 The Capability Element</i> . . . . .	31
<i>A-3.2.8 Memory Pools</i> . . . . .	31

## PART B — THE IGUANA LIBRARY

<b>B-1 The Iguana Library</b> . . . . .	<b>35</b>
<b>B-2 The <code>&lt;cap.h&gt;</code> Header</b> . . . . .	<b>36</b>
B-2.1 <code>iguana_get_cap()</code> . . . . .	36
B-2.2 <code>cap_matches()</code> . . . . .	36
B-2.3 <code>clist_create()</code> . . . . .	36
B-2.4 <code>pd_create_clist()</code> . . . . .	36
B-2.5 <code>clist_delete()</code> . . . . .	37
B-2.6 <code>clist_insert()</code> . . . . .	37
B-2.7 <code>clist_lookup</code> . . . . .	37
B-2.8 <code>clist_remove()</code> . . . . .	37

<b>B-3</b>	<b>The &lt;eas.h&gt; Header</b>	<b>38</b>
B-3.1	eas_create()	38
B-3.2	eas_create()	38
B-3.3	eas_delete()	38
B-3.4	eas_create_thread()	39
B-3.5	eas_delete_thread()	39
B-3.6	eas_map()	39
B-3.7	eas_unmap()	39
B-3.8	eas_share_domain()	40
B-3.9	eas_unshare_domain()	40
B-3.10	eas_modify()	40
<b>B-4</b>	<b>The &lt;env.h&gt; Header</b>	<b>41</b>
B-4.1	The Environment Item Type	41
B-4.2	iguana_getenv()	41
B-4.3	env_get_next()	41
B-4.4	env_name()	41
B-4.5	env_type()	42
B-4.6	env_const()	42
B-4.7	env_memsection()	42
B-4.8	env_memsection_base()	42
B-4.9	env_memsection_size()	42
B-4.10	env_thread()	43
B-4.11	env_thread_id()	43
B-4.12	env_virtpool()	43
B-4.13	env_physpool()	43
B-4.14	env_clist()	43
B-4.15	env_elf_segment_paddr()	43
B-4.16	env_elf_segment_offset()	44
B-4.17	env_elf_segment_filesz()	44
B-4.18	env_elf_segment_memsz()	44
B-4.19	env_elf_segment_vaddr()	44
B-4.20	env_elf_segment_flags()	44
B-4.21	env_elf_file_type()	45
B-4.22	env_elf_file_entry()	45
B-4.23	env_zone()	45
<b>B-5</b>	<b>The &lt;hardware.h&gt; Header</b>	<b>46</b>
B-5.1	hardware_register_interrupt()	46
B-5.2	hardware_back_memsection()	46
<b>B-6</b>	<b>The &lt;memsection.h&gt; Header</b>	<b>47</b>
B-6.1	Memory Types	47
B-6.2	memsection_create()	47
B-6.3	memsection_create_with_pools()	47
B-6.4	memsection_create_user()	48
B-6.5	memsection_create_dma()	48
B-6.6	memsection_create_direct()	48
B-6.7	memsection_create_fixed()	49
B-6.8	memsection_create_fixed_user()	49
B-6.9	memsection_create_in_zone()	49
B-6.10	memsection_register_server()	49
B-6.11	memsection_virt_to_phys()	50

B-6.12	<code>memsection_lookup()</code>	50
B-6.13	<code>memsection_delete()</code>	50
B-6.14	<code>memsection_set_attributes()</code>	50
B-6.15	<code>memsection_base()</code>	51
B-6.16	<code>memsection_size()</code>	51
B-6.17	<code>memsection_map()</code>	51
B-6.18	<code>memsection_unmap()</code>	51
B-6.19	<code>memsection_page_map()</code>	52
B-6.20	<code>memsection_page_unmap()</code>	52
B-6.21	<code>memsection_back_range()</code>	52
<b>B-7 The <code>mutex.h</code> Header</b>		<b>53</b>
B-7.1	<code>okl4_mutex_allocate()</code>	53
B-7.2	<code>okl4_mutex_deallocate()</code>	53
<b>B-8 The <code>&lt;pd.h&gt;</code> Header</b>		<b>54</b>
B-8.1	<code>pd_myself()</code>	54
B-8.2	<code>pd_create()</code>	54
B-8.3	<code>pd_create_pd()</code>	54
B-8.4	<code>pd_create_restricted()</code>	54
B-8.5	<code>pd_create_restricted_in_pd()</code>	55
B-8.6	<code>pd_delete()</code>	55
B-8.7	<code>pd_create_thread()</code>	55
B-8.8	<code>pd_create_thread_with_priority()</code>	55
B-8.9	<code>pd_create_memsection()</code>	56
B-8.10	<code>pd_create_memsection_user()</code>	56
B-8.11	<code>pd_create_memsection_fixed_user()</code>	56
B-8.12	<code>pd_add_clist()</code>	57
B-8.13	<code>pd_release_clist()</code>	57
B-8.14	<code>pd_attach()</code>	57
B-8.15	<code>pd_detach()</code>	57
B-8.16	<code>pd_attach_zone()</code>	58
B-8.17	<code>pd_detach_zone()</code>	58
B-8.18	<code>pd_extension_activate()</code>	58
B-8.19	<code>pd_extension_deactivate()</code>	58
B-8.20	<code>pd_l4id()</code>	59
<b>B-9 The <code>&lt;physmem.h&gt;</code> Header</b>		<b>60</b>
B-9.1	<code>physmem_delete()</code>	60
B-9.2	<code>physmem_info()</code>	60
<b>B-10 The <code>&lt;physpool.h&gt;</code> Header</b>		<b>61</b>
B-10.1	<code>physpool_alloc()</code>	61
B-10.2	<code>physpool_alloc_fixed()</code>	61
<b>B-11 The <code>&lt;segment_info.h&gt;</code> Header</b>		<b>62</b>
B-11.1	<code>get_elf_info()</code>	62
B-11.2	<code>get_segment_info()</code>	62
<b>B-12 The <code>&lt;thread.h&gt;</code> Header</b>		<b>63</b>
B-12.1	<code>thread_create()</code>	63
B-12.2	<code>thread_create_priority()</code>	63
B-12.3	<code>thread_create_simple()</code>	63

B-12.4	<code>thread_start()</code>	64
B-12.5	<code>thread_myself()</code>	64
B-12.6	<code>thread_l4tid()</code>	64
B-12.7	<code>thread_id()</code>	64
B-12.8	<code>thread_delete()</code>	64
B-12.9	<code>thread_delete_self()</code>	65
<b>B-13</b>	<b>The <code>&lt;tls.h&gt;</code> Header</b>	<b>66</b>
<b>B-14</b>	<b>The <code>&lt;zone.h&gt;</code> Header</b>	<b>68</b>
B-14.1	<code>zone_create()</code>	68
B-14.2	<code>zone_delete()</code>	68

## PART C — APPENDIX

<b>C-1</b>	<b>Iguana IDL Description</b>	<b>71</b>
C-1.1	INTERFACE_IGUANA_PD_UUID	72
C-1.1.1	<code>mypd()</code>	72
C-1.1.2	<code>create_memsection()</code>	72
C-1.1.3	<code>create_pd()</code>	72
C-1.1.4	<code>create_thread()</code>	72
C-1.1.5	<code>create_eas()</code>	72
C-1.1.6	<code>create_eas()</code>	72
C-1.1.7	<code>set_callback()</code>	72
C-1.1.8	<code>add_clist()</code>	73
C-1.1.9	<code>release_clist()</code>	73
C-1.1.10	<code>delete()</code>	73
C-1.1.11	<code>create_restricted_pd()</code>	73
C-1.1.12	<code>attach()</code>	73
C-1.1.13	<code>detach()</code>	73
C-1.2	INTERFACE_IGUANA_EAS_UUID	74
C-1.2.1	<code>create_thread()</code>	74
C-1.2.2	<code>delete()</code>	74
C-1.2.3	<code>map()</code>	74
C-1.2.4	<code>unmap()</code>	74
C-1.3	INTERFACE_IGUANA_THREAD_UUID	75
C-1.3.1	<code>id()</code>	75
C-1.3.2	<code>l4id()</code>	75
C-1.3.3	<code>start()</code>	75
C-1.3.4	<code>delete()</code>	75
C-1.3.5	<code>set_exception()</code>	75
C-1.4	INTERFACE_IGUANA_HARDWARE_UUID	76
C-1.4.1	<code>register_interrupt()</code>	76
C-1.4.2	<code>back_memsection()</code>	76
C-1.5	INTERFACE_IGUANA_MEMSECTION_UUID	77
C-1.5.1	<code>register_server()</code>	77
C-1.5.2	<code>lookup()</code>	77
C-1.5.3	<code>info()</code>	77
C-1.5.4	<code>delete()</code>	77
C-1.5.5	<code>set_attributes()</code>	77
C-1.5.6	<code>map()</code>	77
C-1.5.7	<code>unmap()</code>	77
C-1.5.8	<code>page_map()</code>	78
C-1.5.9	<code>page_unmap()</code>	78

---

C-1.5.10	<i>virt_to_phys()</i> . . . . .	78
C-1.6	INTERFACE_IGUANA_PHYSPOOL_UUID . . . . .	79
C-1.6.1	<i>alloc()</i> . . . . .	79
C-1.6.2	<i>alloc_fixed()</i> . . . . .	79
C-1.7	INTERFACE_IGUANA_PHYSMEM_UUID . . . . .	80
C-1.7.1	<i>delete()</i> . . . . .	80
C-1.7.2	<i>info()</i> . . . . .	80
C-1.8	INTERFACE_IGUANA_CLIST_UUID . . . . .	81
C-1.8.1	<i>delete()</i> . . . . .	81
C-1.8.2	<i>insert()</i> . . . . .	81
C-1.8.3	<i>lookup()</i> . . . . .	81
C-1.8.4	<i>remove()</i> . . . . .	81
<b>C-2</b>	<b>Example Configuration File</b> . . . . .	<b>82</b>



---

# Preface

---

The *Iguana Programming Manual* describes the Iguana embedded operating system. For convenience this manual has been divided into two parts. Part A of this manual commences with an overview the main components of the Iguana API. The Iguana protection management system and the initialization process are also outlined in Part A of this manual. Part B of this manual provides a brief description of the main header files and functions available in the Iguana library.

It should be noted that Iguana is a currently a work in progress and is due to undergo substantial change. As such, this manual provides an overview of the current Iguana API and the Iguana library and indicates the areas mostly likely to change in the near future.



# 1 Introduction

---

Iguana, designed specifically for embedded systems, provides the operating system functionality on top of the OKL4 microkernel. Iguana provides a number of services including virtual memory management, a protection framework for the management of access rights, protection domain or process management and thread management as well as a convenient method of using OKL4 primitives.

As Iguana has been designed specifically for embedded systems, both the memory and cache footprints are kept to a minimum. Iguana also attempts to lower the overhead associated with sharing data. Iguana strives to provide the best performance on typical embedded processors and supports the separation and translation feature of the ARM 9 cores by encouraging a non-overlapping address space layout.

To support the minimization of both the memory and cache footprint as well as minimizing the overhead associated with sharing data, Iguana is designed as a single address space system, containing only the *Iguana Address Space (IAS)*. Though Iguana is a single address space system, applications still have the option of creating a separate address space, known as an *External Address Space (EAS)*. External Address Spaces are further described in Section A-1.6.

Iguana uses a capability based *protection management* system to regulate access to objects. The elements used to create the protection management framework in Iguana, as well as an overview of *capability lists* and the implementation of several protection models are detailed in Chapter A-2. Iguana uses the concept of *protection domains* to manage access to objects that exist within the Iguana Address Space. Protection domains are further described in Section A-1.4. A protection domain may contain zero or more threads and zero or more memory sections. Threads and memory sections are described in Sections A-1.2 and A-1.1, respectively.

Iguana allows the user to specify *Physmem* objects and *memory pools*. Physmem objects allow for a specific range of physical memory to be referred to directly, a feature useful for device drivers. Physmem objects are further described in Section A-1.8. Memory pools are used to allow both physical and virtual memory to be cleanly partitioned between different applications as well as allowing users to specify when memory with specific properties, such as TCM, is used. Memory pools allow both physical and virtual memory to be grouped into multiple pools rather than being grouped together in a single global memory pool and are further described in Section A-1.7.

The initial state of Iguana may be configured using *Elfweaver*, a tool that is used to manipulate ELF files. An overview of the main functionality of Elfweaver that is relevant to specifying the initial state of Iguana, is provided in Chapter A-3. The Elfweaver tool is described in greater detail in the *Elfweaver User Manual*.

For convenience, this manual has been divided into three parts. Part A of this manual presents an overview of the main components of the Iguana API, as well as protection management and a brief description of the initialization process. Part B of this manual provides a brief overview of the functions available in the Iguana library. Lastly, a brief overview of the Iguana IDL description and an example configuration file is provided in Part C.



# **PART A**

**Iguana API**



---

## A-1 Iguana API

---

This chapter introduces the *Iguana API* and presents a brief overview of the main concepts in Iguana. Iguana uses *memory sections* to specify units of virtual memory allocation and protection, described in Section A-1.1. Threads are described in Section A-1.2

The capability based system of *protection management*, used by Iguana, is presented in Section A-1.3. The main players in the Iguana protection management system, *protection domains* and *external address spaces (EAS)* are outlined in Sections A-1.4 and A-1.6, respectively. Iguana allows the user to specify *Phymem* objects, allowing the user to refer directly to a specific range of physical memory. Iguana also allows virtual and physical memory to be grouped into *memory pools*, allowing memory to be cleanly partitioned between applications as well as allowing the user to specify when memory with specific characteristics is used. Phymem objects and memory pools are described in Sections A-1.8 and A-1.7, respectively.

Some APIs are considered optional and are only available when the OKL4 system profile *extra* is specified, these include *protection domain extentions*, *zones* and *external address-spaces* documented in Sections A-1.4.1, A-1.5, and A-1.6.

## A-1.1 Memory Sections

Iguana uses *memory sections* to specify units of virtual memory allocation and protection. A memory section consists of a contiguous range of virtual pages and is a multiple of page size, usually 4KB. Protection is uniform to a memory section, that is a thread with a particular set of permissions for a particular area of the memory section has the same permissions for all other areas of the memory section.

Memory sections may be created at boot time, using the `memsection` element in the *configuration file*, or alternatively at runtime, using the `memsection_create()`, library function. The `memsection` element and configuration files are further described in Section A-3.2.6. The Iguana library functions associated with memory sections are outlined in Chapter B-6.

Each memory section is associated with two bit-masks, *flags* and *attributes*, which are used to indicate the memory type and memory attributes associated with the memory section. Flags and attributes that may be associated with a memory section are further described in Sections A-1.1.1 and A-1.1.2, below.

Memory sections may contain application-defined methods of implementing arbitrary functionality. Therefore, memory sections play an important role in the provision of services in Iguana. In order to use a memory section to provide a service, a server thread for the memory section must first be registered with Iguana. This may be achieved at boot time using the `server` attribute of the `program` element, or at run time, using the `memsection_register_server()` library function described in Section B-6.10. The `program` element is further described in Section A-3.2.6. Some memory sections may also serve a special purpose such as backing *capability lists* or and acting as callback buffers.

The `memsection_delete()` function described in Chapter B-6, is used to remove memory sections of all memory types from a protection domain.

### A-1.1.1 Memory Types

Each memory section is associated with a bit-mask, *flags*, used to specify the *memory type* of the memory section. The memory types available in Iguana, together with the corresponding flag and a short description are listed below.

Memory Types		
Memory Type :	Flag:	Description:
NORMAL	0x1	<i>Default memory type in Iguana</i>
FIXED	0x2	<i>Memory section fixed at a particular virtual address</i>
DIRECT	0x4	<i>Memory section has same virtual and physical address</i>
USER	0x10	<i>Memory section mapped by a user pager</i>

The memory type of a memory section is specified at the time of its creation. The memory type of a memory section may be specified at boot time using the *Elfweaver* tool. The `FIXED`, `DIRECT` and `USER` memory types may be specified using the `virt_addr`, `direct` and `pager` attributes of the `memsection` element, respectively. The `memsection` element is further described in Section A-3.2.6. Alternatively, the memory type of memory sections created at run time may be determined by selecting the appropriate `create_memsection()` function from the functions outlined in Chapter B-6.

### A-1.1.2 Memory Attributes

Each memory section is associated with a bit-mask, *attributes*, used to specify the memory attributes of the memory section. The memory attributes available in Iguana, as well as the corresponding *flag* and a short description are listed below.

<b>Memory Attributes</b>		
<b>Memory Attributes :</b>	<b>Flag:</b>	<b>Description:</b>
L4_DefaultMemory	0	<i>Architecture-specific default policy</i>
L4_CachedMemory	1	<i>Architecture-specific caching policy</i>
L4_UncachedMemory	2	<i>Caching disabled</i>
L4_WriteBackMemory	3	<i>Write-back caching policy</i>
L4_WriteThroughMemory	4	<i>Write through caching policy</i>
L4_CoherentMemory	5	<i>Coherent caching policy</i>
L4_IOMemory	6	<i>Architecture specific policy for device memory</i>
L4_IOCombineMemory	7	<i>Uncached, write combining policy</i>

The attributes of a memory section may be specified when a memory section is created at boot time using *Elfweaver* or at run time using the `memsection_set_attributes()` function described in Section B-6.14. The memory attributes are controlled by the physical memory pool used to allocate the memory section. Both the physical memory pool and memory section elements are further described in Sections A-3.2.8.2 and A-3.2.6, respectively. Unless otherwise specified by the user, all newly created memory sections have their attributes set to `L4_DefaultMemory`.

## A-1.2 Threads

Iguana threads are essentially OKL4 threads that may be manipulated directly using the OKL4 system calls. Some OKL4 operations are restricted to privileged threads. These restricted operations are subject to the Iguana protection model. Threads in Iguana may be created at boot time using `thread` element provided by the *Elfweaver* tool, or at run time using an appropriate `thread_create()` library function described in Chapter B-12. The `thread` element is a sub element of the `program` element. The `thread` and `program` elements are further described in Sections A-3.2.3 and A-3.2.4, respectively.

Each thread is associated with a priority level that is determined at the time of its creation. Iguana uses the priority level to determine when a particular thread should run. Priority levels in Iguana range between 0 and 255, inclusive, with 255 being the highest priority level. If the thread is created at boot time its priority level may be specified using the `priority` attribute of the `thread` element. Failing to specify the `priority` attribute or setting it to 0, results in the priority level of the thread being set to the default priority level in Iguana. Alternatively, if the thread is created at run time, using a function that does not request a priority level as an argument will result in the priority level of the new thread being set to the default priority level in Iguana.

Each thread in Iguana is associated with both an `L4_ThreadId` and an Iguana thread reference. When a thread is created using an appropriated `thread_create()` library function, both the Iguana thread reference and the `L4_ThreadId` of the newly created thread are returned to the caller. The Iguana thread reference and the `L4_ThreadId` are interchangeable, that is one can be obtained using the other. The Iguana thread reference of a thread may be obtained by calling the `thread_id()` function with its `L4_ThreadId`. Similarly, the `L4_ThreadId` of a thread may be obtained by calling the `thread_l4tid()` function with the Iguana thread reference of the thread. The thread library functions are further described in Chapter B-12.

### A-1.3 Capabilities

Iguana uses *capabilities* to control access to data or objects. A capability may be viewed as a right to perform a particular operation on a particular object. Capabilities may be implemented using several different methods. The method used by Iguana in implementing capabilities is that of password capabilities. A capability in Iguana consists of a *reference* to the object, an *interface identifier* and a *password*.



Figure A-1.1: *Structure of a capability*

The reference is used to uniquely identify the object to which the capability belongs and the interface identifier is used to specify a value used to identify a particular interface consisting of a set of methods. The holder of the capability may only access the object using the methods prescribed by the particular interface. The password component of a capability specifies a password generated by Iguana and is used by Iguana to verify that the capability is genuine. A capability may only be exercised if the holder of the capability also holds the correct password.

Iguana uses two types of capabilities, *master capability* and *invocation capability*. When an object is created, the creator receives a master capability for the object. Master capability gives the holder the right to invoke any method on the object, including methods not registered at the time at which the master capability was granted. A master capability has an interface identifier of 0. The holder may subsequently choose to pass on the master capability, that is, Iguana does not restrict the master capability to a particular process or user. Alternatively, the holder may choose to pass on a subset of the master capability, that is access only to a particular interface, known as an invocation capability. An invocation capability gives the holder the right to invoke any method from a specified interface on the corresponding object.

Capabilities in Iguana are only valid if they are stored in a *capability list* or *Clist*. A Clist may be viewed as an array of *slots* or storage spaces for capabilities. Each slot specifies the Clist and the index at which the capability is stored. Each protection domain may contain multiple capability lists. Capability lists are further described in Section A-2.1.

Once a capability is passed on to another protection domain, the protection domain that held the original capability is no longer in control of access to the object. This is due to the fact that the protection domain receiving the capability is currently not restricted by Iguana from making copies of the capability as well as passing it on to other protection domains.

There are two main solutions to this issue. The first alternative is for Iguana to invalidate the old password and issue a new password to the owning protection domain. This would hand control back to the protection domain that created the object. The second alternative is to add the capability directly to the capability list of the recipient whilst denying the recipient both read and write permissions on the clist, thus ensuring that capabilities are not copied or passed on.

Currently, this system of protection is not fully implemented in Iguana. However, the basic framework required for its implementation is already present in Iguana.

## A-1.4 Protection Domains

*Protection Domains* or *PDs* are used by Iguana to manage access to objects. Though Iguana has been designed as a single address space system, consisting of only the *Iguana Address Space (IAS)*, it allows applications to create separate address spaces known as *External Address Spaces (EAS)*, described in Section A-1.6. It should be noted that external address spaces are limited to a restricted API and cannot contain protection domains within them. Therefore all protection domains reside within the Iguana Address Space.

Each protection domain may contain zero or more *threads*, zero or more *memory sections* and one or more *capability lists* or *Clists*. Threads residing within the same protection domain have unrestricted access to all memory sections belonging to that protection domain. Threads in one protection domain may however, access the memory of a thread belonging to a different protection domain if the thread possesses the appropriate capabilities. Threads and memory sections are further described in Sections A-1.2 and A-1.1, respectively. Capabilities and capability lists are further described in Sections A-1.3 and A-2.1, respectively.

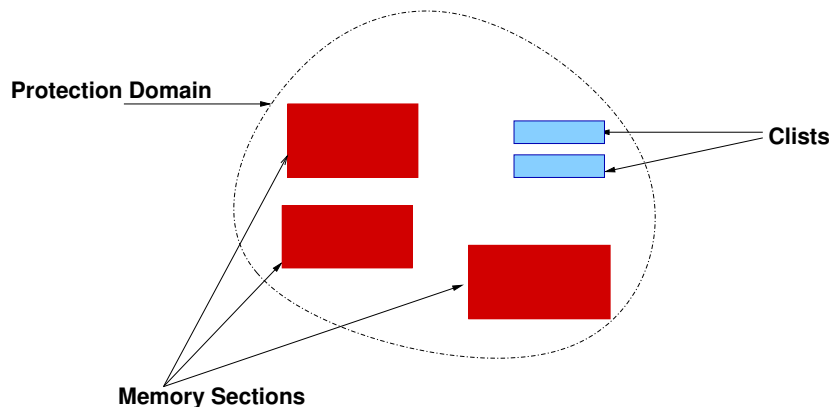


Figure A-1.2: *An example protection domain*

Threads residing in a particular protection domain and holding the requisite capabilities may create a new protection domain or external address space. A new protection domain or a new external address space may be created at run time using the Iguana library functions, `pd_create()` or `eas_create()`, respectively. The newly created protection domain or external address space is said to be owned by the creating protection domain. A protection domain may be subsequently removed from the Iguana Address Space using the Iguana library function, `pd_delete()`. It should be noted that when a protection domain is deleted, all protection domains and external address spaces owned by that protection domain are also deleted. The Iguana library functions provided for protection domains and external address spaces are described in Part B of this manual in Chapters B-8 and B-3, respectively.

Alternatively, new protection domains may be created at boot time using the `pd` element provided by the *Elfweaver* tool. The user may subsequently create new memory sections and threads belonging to the newly created protection domain using the elements `memsection` and `thread`, respectively. A description of those aspects of *Elfweaver* relevant to the initialization of Iguana together with a detailed description of `pd` element is provided in Chapter A-3 and Section A-3.2.5, respectively. For additional information regarding the *Elfweaver* tool, please refer to the *Elfweaver User Manual*.

### A-1.4.1 Protection Domain Extensions

One method of increasing the robustness of a large system is to split it into smaller sub-systems and place each sub-system in a separate protection domain. However, it may not be possible to encapsulate these subsystems in their entirety. Iguana implements *protected libraries*, which are implemented as extensions to protection domains.

Currently a system may only contain a single protection domain extension. In addition, only a single thread may access the protection domain extension at one time. In order to access the protection domain extension,

the thread must explicitly activate it using the function `pd_extension_activate()`. Once the extension is activated, it is visible to and only to the thread calling the function. The thread may then deactivate the extension using the function `pd_extension_deactivate()`. These functions are both further described in Chapter B-8.

## A-1.5 Zones

A *Zone* is a collection of memory sections that are grouped into one of more *windows*. Each window is *1MB* in size and aligned on a *1MB* boundary in the virtual address space. It should be noted that on ARM9 machines, zones may be shared between protection domains with greater efficiency than normal memory sections by exploiting the *Fast Address Space Switching* (FASS) capabilities of the processor.

When a particular zone is attached to a protection domain, all memory section residing within the zone are also attached to that protection domain. Similarly, when a zone is detached from a protection domain, all memory sections residing within that zone are also detached from the protection domain.

It should be noted that all memory sections belonging to a particular zone are all attached with the same access permissions. All memory sections within a particular window belong to a single zone. It is not possible to share all or part of the *1MB* region of a window with other zones or with memory sections that are not part of the zone.

## A-1.6 External Address Spaces

*External Address Spaces* (EAS) are raw OKL4 address spaces and are not part of the *Iguana Address Space*. External address spaces are primarily supported to allow native Linux applications to run in full binary compatibility mode. External address spaces also provide support for legacy applications as well as applications that are too large to share the Iguana address space.

An external address space can be created by a protection domain belonging to the Iguana Address Space. An EAS may contain zero or more threads and is considered to be owned by the creating protection domain. If the protection domain owning the external address space is deleted, the external address space will also be deleted. An existing external address space cannot create a new EAS or a new protection domain.

External address spaces are limited to a restricted API and have limited access to Iguana services. External address spaces do not have access to *capabilities* or *capability lists*, making them dependent on the owning protection domain with regards to accessing other objects. Capabilities and capability lists are described in Sections A-1.3 and A-2.1, respectively. Iguana supports only a limited range of operations on external address spaces. Supported operations include the creation and deletion of external address spaces using the Iguana library functions, `eas_create()` and `eas_delete()`, respectively. Iguana also allows external address spaces to create and delete threads as required, via the use of the Iguana library functions, `eas_create_thread()` and `eas_delete_thread()`.

An EAS cannot directly access memory belonging to another EAS or another protection domain by default. If access to memory belonging to another EAS or protection domain is required, the EAS must request its owning protection domain to have the memory directly mapped into its address space using the `eas_map()` Iguana library function. These mappings may be subsequently unmapped using the Iguana library function, `eas_unmap()`.

External address spaces cannot communicate with other external address spaces or with other protection domains by default. The Iguana library functions relating to external address spaces are described in Part B of this manual in Chapter B-3.

## A-1.7 Memory Pools

Iguana allows both virtual and physical memory to be grouped into separate *memory pools*. This allows both virtual and physical memory to be cleanly partitioned between different applications and allows the user to specify when memory with specific characteristics such as TCM is used. Each system contains a single default physical and virtual memory pool.

The default virtual and physical memory pools must be specified using the `virtpool` and `physpool` attributes, respectively, of the `rootprogram` element. Other virtual and physical memory pools may be specified using the `virt_pool` and `phys_pool` elements, respectively. The aspects of Elfweaver relevant to the initialization of Iguana are described in Chapter A-3. For a more detailed description of the Elfweaver tool, please refer to the *Elfweaver User Manual*.

Currently, Iguana does not facilitate the dynamic creation of physical or virtual memory pools. However, it is expected that future versions of Iguana will allow users to create both virtual and physical memory pools dynamically, as required.

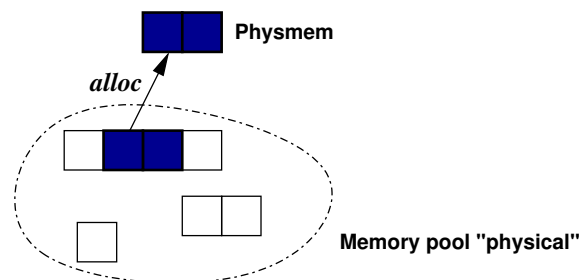


Figure A-1.3: *Physmem object allocated from Memory Pool*

It is important to note that a physical memory pool does not directly refer to physical memory. This role is performed by *Physmem* objects described in Section A-1.8. Each physical memory pool object manages a set of physical pages. Once established, memory pools may be used to create *Physmem* objects described in Section A-1.8. Figure A-1.3 provides an example of a *Physmem* object consisting of two physical pages, which has been allocated from the default physical memory pool, "physical". Similarly, it should be noted that virtual memory pools do not directly refer to virtual memory. This role is performed by memory sections described in Section A-1.1. Both virtual and physical memory pools in Iguana are not limited to a particular size and may contain as few or as many pages as required. In Figure A-1.3, the example memory pool, "physical" consists of group of seven physical pages.

Certain Iguana library functions such as `memsection_create_with_pools()`, allow the user to specify a reference to a particular physical and/or virtual memory pool, which is then used for the memory allocation of the newly created memory section. Similarly, all Iguana library functions requiring the allocation of virtual or physical memory from a specific virtual or physical memory pool require the caller to specify a reference to the memory pool from which the memory is to be allocated.

It is expected that a majority of applications will not use user defined virtual and physical memory pools in favour of the default global virtual and physical memory pools, created and managed by Iguana. The Iguana library functions are described in Part B of this manual.

## A-1.8 Phymem

*Phymem* objects are first class objects in Iguana and represents a contiguous range of pages in physical memory. *Phymem* objects may be mapped into memory sections using the `Map` system call.

*Phymem* objects may be created at runtime by allocating a physical page from a *memory pool* as described in Section A-1.7. Iguana facilitates the creation of *Phymem* objects at run time by providing the Iguana library functions `physpool_alloc()` and `physpool_alloc_fixed()`, described in Chapter B-10. The physical address of the *Phymem* object and its size in bytes may be obtained using the `phymem_info()` Iguana library function. *Phymem* objects may be subsequently deleted using the Iguana library function, `phymem_delete()`. Both functions are further described in Chapter B-9.

Currently, *Phymem* objects cannot be created at boot time using *Elfweaver*. It is expected that future versions of Iguana will allow users to specify *Phymem* objects at boot time via the use of an appropriate XML tag in the *configuration file*.

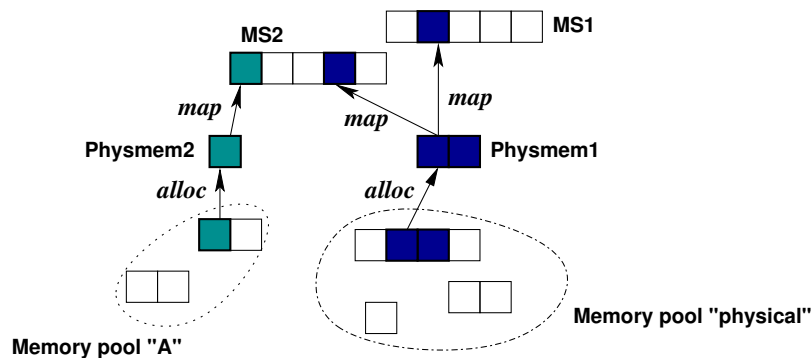


Figure A-1.4: *Memory sections backed by Phymem objects*

Figure A-1.4 above, presents an example that consists of two memory sections, MS1 and MS2, two *Phymem* objects, *Phymem1* and *Phymem2*, two memory pools, "physical" and "A". *Phymem1* is allocated out of the default physical memory pool, "physical" with *Phymem2* being allocated out of a user defined physical memory pool, "A". A single physical page of *Phymem1* has been mapped to a page in both memory sections, MS1 and MS2. *Phymem2* is mapped to a single page in memory section, MS2. Memory sections and memory pools are further described in Sections A-1.1 and A-1.7, respectively.

*Phymem* objects were introduced to meet the requirements of device drivers and Wombat which required direct access to specific ranges of physical memory, such as device registers. This requirement was previously met using the `hardware_back_memsection()` Iguana library function, which compromised security and did not comfortably fit the Iguana object model.

It is expected that most applications will not deal directly with *Phymem* objects and as such it is not expected to have a large impact on existing applications.

## A-2 Protection Management

Iguana uses a capability based protection management system that has been designed to be as unobtrusive as possible. Access to objects is managed by restricting it to holders of the appropriate *capability*. A capability may be viewed as a right to perform a particular operation on a particular object and is further described in Section A-1.3.

This chapter examines the elements used to create the protection management framework in Iguana and provides an overview of capability lists. It should be noted that Iguana does not currently, fully enforce the system of protection management outlined in this chapter. However, the basic framework for its implementation is currently in place. It is expected that this protection management system will be fully implemented in future versions of Iguana.

### A-2.1 Capability Lists

Iguana uses a capability based protection management system. When an object is created, its creator receives a *master capability*, entitling the holder to perform any operation on the newly created object. The holder of this capability may subsequently pass on the master capability or a subset of the master capability to another entity.

Iguana password capabilities are stored in data structures called *capability lists* or *clists*. A Clist may be viewed as an array containing capabilities. Each protection domain may contain zero or more Clists as required, that is, newly created protection domains do not contain Clists by default. Provided that an application uses the Iguana library functions described in Part B of this manual, Clists are managed internally by Iguana. Alternatively, where applications utilize the raw Iguana functions described in Chapter C-1, Clists must be managed by the application itself.

Figure A-2.1 below, provides an illustration of Clists in Iguana. Each protection domain descriptor contains an array of references to associated Clists. In order to enforce system-wide access control policies, Iguana allows an extra level of indirection, thus allowing security managers and policy managers to be interposed.

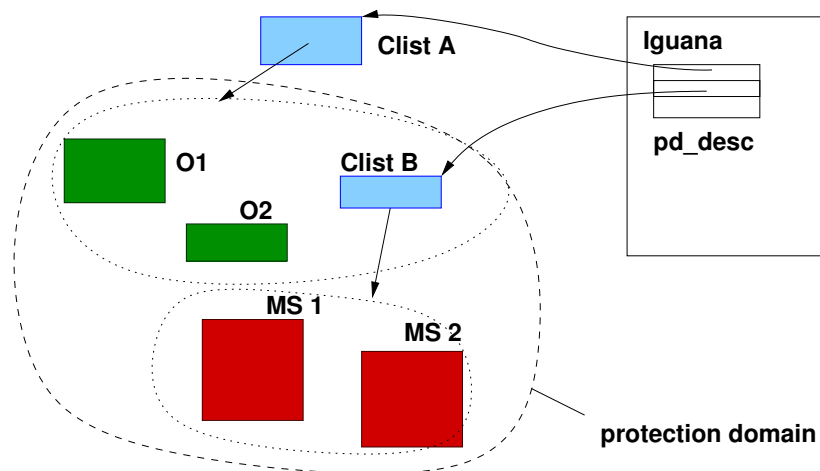


Figure A-2.1: *Capability lists in Iguana*

As illustrated in Figure A-2.1, Iguana stores capabilities in a two-level data structure. Each protection domain descriptor contains an array of references to Clist, where each Clist contains an array of capabilities. Access to Clists are also managed using capabilities stored in a separate Clist. For example in Figure A-2.1, the access to Clist B is controlled by a capability stored in Clist A. Capabilities stored in Clist A also control access to the objects O1 and O2. Access to memory sections MS1 and MS2 are in turn managed by a capability stored in Clist B.

When validating a method invocation, Iguana searches the Clists of the protection domain for a matching valid capability corresponding to the particular method. Clists may be shared between protection domains by expressly adding the Clist to a different protection domain using the `pd_add_clist()` Iguana library function, described in Part B of this manual in Section B-8.12. Clists expressly added to a protection domain, as described above, may be subsequently removed using the `pd_release_clist()` Iguana library function. Clists cannot be shared with external address spaces as they operate on a restricted API that does not include access to capabilities.

## A-3 System Initialization

---

The initial state of Iguana may be configured using *Elfweaver*, a tool that is used to manipulate ELF files. Elfweaver provides a number of other utilities described in the *Elfweaver User Manual*. The functionality of Elfweaver, that is most relevant in determining the initial state of Iguana, is the *merge* function. The *merge* command in Elfweaver may be used to merge multiple ELF files to create a single bootimage and has the following usage:

```
Usage: elfweaver merge [options] specfile
```

The main functionality of the *merge* command is utilized using the *specfile* argument, which is used to specify a *configuration file*. The aspects of the configuration file and the command options of the *merge* command, relevant to the initialisation of Iguana, are described below. Elfweaver and its functionality are further described in the *Elfweaver User Manual*.

### A-3.1 Merge command options

The options that are most relevant in determining the initial state of Iguana are the *output* and *map* options, described in Sections A-3.1.1 and A-3.1.2, below.

#### A-3.1.1 Output

The output option, `-o` or `--output`, is used to specify the name of the file to which the merged file is written. An output file must be specified, failing to specify an output file will result in Elfweaver issuing an error message.

The following command may be used to merge a set of ELF files as specified by the configuration file, `config.xml`, storing the resulting file in the file `image.elf`.

```
elfweaver merge config.xml --output=image.elf
```

#### A-3.1.2 Memory Map

The `--map` option is used to obtain a memory map of the built image and has the following usage:

```
Example Usage: elfweaver merge --map config.xml
```

### A-3.2 The Configuration File

The *configuration file* is supplied as an argument to the *merge* command and is used to specify the various aspects of the intended layout of the bootimage. The configuration file is specified in well formed XML and uses XML elements to specify the various aspects of the bootimage. The elements that are relevant in determining the initial state of Iguana are examined below, descriptions of the other elements that are also present in the configuration file are described in the *Elfweaver User Manual*.

Compound elements are XML elements that are internally expanded by Elfweaver. These elements require the user to specify fewer elements than would otherwise be necessary. The expanded compound elements in a particular configuration file may be obtained using the *map* option described in Section A-3.1.2.

### A-3.2.1 The Root Program Element

Root Program Element Attributes		
Attribute:	Type:	Description:
<code>file</code>	<i>required</i>	The path or location of the root server in the ELF file.
<code>virtpool</code>	<i>required</i>	The name of the default virtual memory pool.
<code>physpool</code>	<i>required</i>	The name of the default physical memory pool.
<code>direct</code>	<i>optional</i>	A Boolean value.
<code>pager</code>	<i>optional</i>	The default pager for the root program.

**Example: Root Program Element**

```
<rootprogram file="/path/to/iguana_server" virtpool="main_virt"
  physpool="main_phys">
  <segment name="data" physpool="somore" />

  <extension name="library" file="/path/to/extension" >
  </extension>
</rootprogram>
```

The `rootprogram` element is used to specify the program that is first started by OKL4. The `rootprogram` element allows the user the flexibility of either specifying Iguana or an alternative root server of their choice.

The `virtpool` and `physpool` attributes are used to specify the default virtual and physical memory pools, respectively. Where the `rootprogram` element fails to specify valid virtual and physical memory pools using the `virtpool` and `physpool` attributes, an error will be raised by Elfweaver. The `direct` attribute may be set to `true` or `false` by the user and is used to specify whether the segments are to be placed at the same physical address as the virtual address specified in the ELF program header. This attribute takes priority over the `physpool` attribute of the `rootprogram` element.

### A-3.2.2 The Extension Element

Extension Element Attributes		
Attribute:	Type:	Description:
<code>name</code>	<i>required</i>	The name of the extension.
<code>file</code>	<i>optional</i>	The path or location of the extension.
<code>physpool</code>	<i>optional</i>	Physical memory pool used for all segments belonging to the extension.
<code>pager</code>	<i>optional</i>	The default pager used for all segments in the extension's ELF file.
<code>direct</code>	<i>optional</i>	A Boolean value.
<code>start</code>	<i>optional</i>	A Boolean value.

**Example: Extension Element**

```
<extension name="library" file="/path/to/extension" >
</extension>
```

The `extension` element is used to describe a Root Program Extension Library, which allows arbitrary code to be loaded into the root program at boot time without the need to rebuild the main root program binary.

The root program will call the entry point of each extension library prior to entering the main message processing loop. The entry point is called with no arguments and the return value is not checked. Extensions are initialized in order of appearance in the configuration file.

The `file` attribute is used to specify the path, or location of the extension. Where the `file` attribute is omitted it is assumed that the extension has been linked in to the root program. The physical memory pool specified by the `physpool` attribute of the `extension` element takes precedence over the `physpool` attribute specified by the `rootprogram` element. Where the `physpool` attribute is left unspecified, the physical memory pool specified by the `physpool` attribute of the `rootprogram` element will be used as the default physical memory pool to be for the allocation of all segments belonging to the extension.

The `direct` attribute which may be set to either `true` or `false` by the user and is used to specify whether segments belonging to the extension are to be placed at the same physical address as the virtual address specified in the ELF program header. This attribute takes precedence over the `physpool` attribute of the `extension` element. Where the `direct` attribute is left unspecified, Elfweaver will set this value to the default value of `false`. The `start` attribute is used to indicate the start of the function used to initialize the extension. If left unspecified, Elfweaver will use the entry point of the ELF file as the initializing function.

### A-3.2.3 The Program Element

<b>Program Element Attributes</b>		
<b>Attribute:</b>	<b>Type:</b>	<b>Description:</b>
<code>name</code>	<i>required</i>	Name of the program.
<code>file</code>	<i>required</i>	The location or path of the specified program.
<code>priority</code>	<i>optional</i>	The priority level of the specified thread.
<code>virtpool</code>	<i>optional</i>	Virtual memory pool used for the program.
<code>physpool</code>	<i>optional</i>	Physical memory pool used for the program.
<code>pager</code>	<i>optional</i>	Default pager to be used for all segments in the program's ELF file.
<code>direct</code>	<i>optional</i>	A Boolean value.
<code>server</code>	<i>optional</i>	Key in the object environment to look up a particular memory section.

**Example: Program Element**

```
<program name="demo" file="path/to/file" priority="110">
  <stack size="0x4000" />
  <heap size="0x10000" />

  <commandline>
    <arg value="demo" />
    <arg value="second_arg" />
  </ commandline>
</ program>
```

The `program` element is used to specify a program started in the traditional manner. Where Elfweaver is provided with an ELF file containing a program it will create a new protection domain for the program, attach all segments of the ELF file into that protection domain and create a stack and a heap for the program. In addition, Elfweaver will start a single thread, the main thread, commencing at the entry point of the ELF file and pass the command line arguments and the object environment to this thread, as required.

It should be noted that the virtual memory pool specified by the `virtpool` attribute and physical memory pool specified by the `physpool` attribute take precedence over the `virtpool` and `physpool` attributes of the `rootprogram`. If one or both of the `virtpool` and `physpool` attributes of the `program` element were

left unspecified, then the `virtpool` and/or `physpool` attributes of the `rootprogram` element would be used to determine the memory pool used for all memory allocation associated with the program. The `direct` attribute may be set to either `true` or `false` and is used to specify whether the segments belonging to the program are to be placed at the same physical address as the virtual address specified in the ELF program header. Where the `direct` attribute is left unspecified, Elfweaver will set this attribute to the default value of `false`. It should be noted that this attribute takes priority of the `physpool` attribute. The `priority` attribute is used to specify the priority level of the specified thread. The priority level must be a value between 0 and 255, inclusive. If left unspecified, the priority of the thread will be set to the default value of 100.

### A-3.2.4 The Thread Element

Thread Element Attributes		
Attribute:	Type:	Description:
<code>name</code>	<i>required</i>	The name of the thread.
<code>start</code>	<i>required</i>	The start address of the thread.
<code>priority</code>	<i>optional</i>	The priority of the thread.
<code>virtpool</code>	<i>optional</i>	Virtual memory pool used for the thread.
<code>physpool</code>	<i>optional</i>	Physical memory pool used for the thread.

**Example: Thread Element**

```

<thread name="second_thread" start="start_second_thread" >
  <commandline>
    <arg value="argv0"/>
    <arg value="hello"/>
    <arg value="world"/>
  </commandline>
</thread>

```

The `thread` element may be used to specify a thread that is to be started at boot time. The `thread` element can be declared within the `pd` and `program` elements. Where either of these elements specify multiple `thread` elements, they will be started in priority order, though no guarantee is provided.

The `priority` attribute is used to specify the priority level of the thread. The priority level must be a value between 0 to 255 inclusive, where 255 is the highest priority level. If left unspecified, Elfweaver will assign the thread a priority level of 100. The `virtpool` and `physpool` attributes are used to specify the virtual and physical memory pools to be used for the allocation of all memory associated with the thread. Where either or both of these attributes are left unspecified, Elfweaver will use the corresponding attribute specified in the corresponding `program` element to determine the memory pool used for the allocation of all memory associated with the thread. Where the missing attribute is also unspecified in the corresponding `program` element, the corresponding attribute specified in the `rootprogram` element is used.

### A-3.2.5 The Protection Domain Element

Protection Domain Element Attributes		
Attribute:	Type:	Description:
<code>name</code>	<i>required</i>	The name of the element.
<code>file</code>	<i>optional</i>	The location or path of the ELF file.
<code>virtpool</code>	<i>optional</i>	Virtual memory pool used for protection domain.
<code>physpool</code>	<i>optional</i>	Physical memory pool used for protection domain.
<code>pager</code>	<i>optional</i>	Default pager for all segments in <code>file</code> .
<code>direct</code>	<i>optional</i>	A Boolean value.

**Example:** *Protection Domain Element*

```
<pd name="isolated">
  <memsection name="make_dynamically" size="16K" attach="rwx" />
</pd>
```

The `pd` element allows the user finer control over the creation of a protection domain. All contents of the protection domain must be stated explicitly using the `pd` element.

The `virtpool` and `physpool` attributes are used to specify the virtual and physical memory pools used for the allocation of all virtual and physical memory associated with the protection domain. Where the `physpool` attribute of the `pd` element is specified, it takes priority over the `physpool` attribute specified by the `program` and `rootprogram` elements. The `direct` attribute, which may be set to either `true` or `false`, is used to specify whether all segments with the exception of memory sections, belonging to the protection domain should be located at the same physical address as the virtual address specified in the ELF file. The `direct` attribute takes priority over the `physpool` attribute of the `pd` element. If left unspecified, Elfweaver sets the `direct` attribute to its default value, `false`. Where the `file` attribute is specified, the `pager` attribute is used to specify the default pager to be used for all segments in the file.

### A-3.2.6 The Memory Section Element

#### Memory Section Element Attributes

Attribute:	Type:	Description:
<code>name</code>	<i>required</i>	The name of the memory section.
<code>file</code>	<i>optional</i>	The path or location of the memory section.
<code>size</code>	<i>optional</i>	The size of the memory section in bytes, ignored if <code>file</code> is specified.
<code>virt_addr</code>	<i>optional</i>	Virtual address of the base of the memory section.
<code>phys_addr</code>	<i>optional</i>	Physical address of the base of the memory section.
<code>virtpool</code>	<i>optional</i>	Virtual memory pool used to allocate the memory section.
<code>physpool</code>	<i>optional</i>	Physical memory pool used to allocate the memory section.
<code>align</code>	<i>optional</i>	Memory alignment of the memory section.
<code>attach</code>	<i>optional</i>	The access permissions of the memory section.
<code>direct</code>	<i>optional</i>	A Boolean value.
<code>pager</code>	<i>optional</i>	The pager associated with the memory section.

#### Example: Memory Section Element

```
<memsection name="data" file="/path/to/data" virt_addr="0x10000000"
  physpool="somemore" pager="custom">
  <cap name="data_rx">
    <right value="read" />
    <right value="execute" />
  </cap>
</memsection>

<memsection name="make_dynamically" size="0x4000" attach="rwx" />
```

The `memsection` element is used to describe the creation of a new memory section. Elfweaver allows the user to set the newly created memory section to contain data located in a specific file, as required. The `memsection` element allows the user the flexibility of relating a memory section to sections in the output file as well as the layout of the memory section in physical and/or virtual memory.

The `virtpool` and `physpool` attributes of the `memsection` element are used to describe the virtual and physical memory pools from which the memory section is to be allocated. Where either or both of these attributes are left unspecified, Elfweaver will use the corresponding attribute specified by the corresponding `pd` element. If either or both attributes are also not specified by the `pd` element, Elfweaver will use the corresponding attribute specified by the corresponding `program` or `rootprogram` element, in that order.

The `align` attribute is used to specify the memory alignment of the memory section. Where the `align` attribute is specified, the memory section will be placed in the selected memory pool and aligned to the supplied boundary. The alignment must be a power of two greater than or equal to the smallest page size. The `attach` attribute is used to specify the permissions with which the memory section is mapped into the protection domain's address space. The `attach` attribute may contain any combination of the characters, `r`, `w` and `e`, which represent the permissions, *read*, *write* and *execute*, respectively.

The `direct` attribute is a Boolean value. Where the `direct` attribute is set to `true`, the memory section will be placed at the same physical address as the virtual address specified by the `virt_addr` attribute. The `pager` attribute serves a dual purpose. Firstly, where present, it indicates that the memory section's virtual address should not be mapped to its physical address when the memory section is created. Secondly, the value specified using `pager` attribute is used to indicate the pager used to perform the mapping. Where the `pager` attribute is used to specify a value other than `okl4`, this refers to a custom pager.

The `memsection` element may contain one or more `cap` elements.

### A-3.2.7 The Capability Element

Capability Element Attributes		
Attribute:	Type:	Description:
<code>name</code>	<i>required</i>	Name of the capability.

**Example: Capability Element**

```
<memsection name="data" file="/path/to/data">
  <cap name="rx">
    <right value="read" />
    <right value="execute" />
  </cap>
</memsection>
```

The `cap` element allows the user to create a capability associated with specific rights with regards to a specific object.

Each object is implicitly created with a master capability known as `<name>/master`, where `<name>` denotes the name of the object. For example, the example capability tag below is specified within a `memsection` element named `data`. The master capability of this `memsection` element is denoted as `data/master`.

Each `right` element consists of a single attribute, `value` which is used to denote the name of the right. The example capability tag below specifies a capability known as `data/rx` which gives the holder both read and execute permissions on the memory section `data`.

Currently the values that may be specified by the `value` attribute in the `right` element are:

<code>read:</code>	Read permission.
<code>write:</code>	Write permission.
<code>execute:</code>	Execute permission.
<code>master:</code>	Master rights to the object.

### A-3.2.8 Memory Pools

Iguana has introduced *memory pools* to allow physical and virtual memory to be grouped together as described in Section A-1.1. Virtual memory pools and physical memory pools are described in Sections A-3.2.8.1 and A-3.2.8.2, below.

#### A-3.2.8.1 Virtual Memory Pools

Currently all virtual memory is grouped together in a single default memory pool with the name `"virtual"`. The user may however, specify the region of memory belonging to the default virtual memory pool by specifying the `base` and `end` addresses of the memory region. The `iguana_memory_pool` tag below provides an example of the specification of the default `"virtual"` memory pool.

**Example:** *Iguana Memory Pool Tag - Virtual*

```
<iguana_memory_pool name="virtual">
  <iguana_memory base="0x1000" end="0xd0000000"/>
</iguana_memory_pool>
```

**A-3.2.8.2 Physical Memory Pool**

Iguana allows physical memory to be grouped into separate memory pools as required by the user. The user may specify their own physical memory pool or use an existing physical memory pool. As with the virtual memory pool, Iguana provides a default physical memory pool with the name "physical". As with the virtual memory pool, the region of memory belonging to a physical memory pool may be specified by indicating the start and end of the memory region by setting the `base` and `end` attributes appropriately.

The first example `iguana_memory_pool` tag below, provides an example of the specification of the default "physical" memory pool. The second example tag provides an example of the specification of a tightly coupled memory or "tcm" memory pool.

**Example 1:** *Iguana Memory Pool Tag - Physical*

```
<iguana_memory_pool name="physical">
  <iguana_memory base="0xa0000000" end="0xa3800000"/>
</iguana_memory_pool>
```

**Example 2:** *Iguana Memory Pool Tag - TCM*

```
<iguana_memory_pool name="tcm">
</iguana_memory_pool>
```

# **PART B**

**The Iguana Library**



---

## B-1 The Iguana Library

---

Part B of this manual provides a description of the Iguana library. Each header file in the Iguana library is described in a separate chapter. Each chapter lists the functions contained within the header file and provides a brief description of the function.

Some header files contain functions that begin with one or more underscore characters (`_`), for example the `memsection.h` header file contains the function, `_cap_memsection_create()`. These functions are internal to Iguana and its use by user programs is strongly discouraged as these functions may be modified without warning. Internal functions are not listed or described in this manual.

A brief overview of the Iguana IDL description is also provided in Appendix C-1.

---

## B-2 The <cap.h> Header

---

### B-2.1 `iguana_get_cap()`

```
cap_t iguana_get_cap(uintptr_t ref, unsigned interface);
```

The `iguana_get_cap()` function is used to retrieve a particular capability. This function requires the arguments `ref` and `interface`. The `ref` argument is used to specify a reference to the object and the `interface` argument is used to specify an interface identifier which uniquely specifies a particular interface. Capabilities, object references and interface identifiers are further described in Section A-1.3.

Iguana conducts a search of the capability lists belonging to the caller until a capability matching both the specified object reference and interface identifier is located. The `iguana_get_cap()` function returns the matching capability on successful completion and `INVALID_CAP` on failure.

### B-2.2 `cap_matches()`

```
int cap_matches(cap_t cap, objref_t object, int interface);
```

The `cap_matches()` function is used to verify whether a particular capability corresponds to a specified object reference and interface identifier. This function requires the arguments `cap`, `object` and `interface`. Iguana checks whether the capability specified by the `cap` argument corresponds to the object reference and interface identifier specified by the arguments `object` and `interface`, respectively. Where a match is found, the `cap_matches()` function returns 1, otherwise 0 is returned. Capabilities, object references and interface identifiers are further described in Section A-1.3.

### B-2.3 `clist_create()`

```
clist_ref_t clist_create(void);
```

The `clist_create()` function is used to create a new empty capability list owned by the current protection domain. The `clist_create()` function returns a reference to the newly created capability list on successful completion.

### B-2.4 `pd_create_clist()`

```
clist_ref_t pd_create_clist(pd_ref_t pd);
```

The `pd_create_clist()` function is used to create a new empty capability list owned by the protection domain specified by the argument, `pd`. The `pd_create_clist()` function returns a reference to the newly created capability list on successful completion.

## B-2.5 `clist_delete()`

```
void clist_delete(clist_ref_t clist);
```

The `clist_delete()` function is used to delete the capability list specified by the `clist` argument.

## B-2.6 `clist_insert()`

```
int clist_insert(clist_ref_t clist, cap_t cap);
```

The `clist_insert()` function is used to add the capability specified by the `cap` argument to the capability list specified by the `clist` argument. The `clist_insert()` function returns 0 on successful completion and a non-zero value on failure. This function may fail for a number of reasons including the capability list being over full and not being capable of expansion or due to invalid input parameters.

## B-2.7 `clist_lookup`

```
int clist_lookup(clist_ref_t clist, uintptr_t ref,  
                unsigned interface, cap_t *cap);
```

The `clist_lookup` function is used to find a capability matching a specified object reference and interface identifier. This function requires the arguments `clist`, `ref`, `interface` and `cap`. Iguana conducts a linear search of the capability list specified by the `clist` argument for a capability matching the object reference and interface identifier specified by the arguments `ref` and `interface`, respectively. Where matching capability is found, it is returned using the `cap` argument. The `clist_lookup` function returns 0 on successful completion and a non-zero value on failure.

## B-2.8 `clist_remove()`

```
int clist_remove(clist_ref_t clist, cap_t cap);
```

The `clist_remove()` function is used to remove a single copy of the capability specified by the `cap` argument from the capability list specified by the `clist` argument. Where multiple copies of a particular capability are present within the capability list, multiple calls are required to remove all copies of the capability. This function returns 0 on success and a non-zero value on failure.

---

## B-3 The <eas.h> Header

---

### B-3.1 eas\_create()

```
eas_ref_t eas_create(L4_Fpage_t kip, L4_Fpage_t utcb,  
                    L4_SpaceId_t *l4_id);
```

The `eas_create(L4_Fpage_t kip, L4_Fpage_t utcb, L4_SpaceId_t *l4_id)` function may be used by an existing protection domain to create a new *External Address Space (EAS)*. This method requires the arguments `kip` and `utcb`, used to specify the location of the `kip` and `utcb` of the new EAS. The `eas_create()` method returns a reference to the newly created EAS on successful completion.

It should be noted that a separate `eas_create()` method exists for the ARM architecture. This method is described below.

### B-3.2 eas\_create()

```
eas_ref_t eas_create(L4_Fpage_t kip, L4_Fpage_t utcb,  
                    int pid, L4_SpaceId_t *l4_id);
```

The `eas_create(L4_Fpage_t kip, L4_Fpage_t utcb, int pid, L4_SpaceId_t *l4_id)` function may be used by an existing protection domain to create a new *External Address Space (EAS)*. It should be noted that this method is specific to the ARM architecture. The `eas_create()` function described above in Section B-3.1 should be used for all other architectures.

This function requires the arguments `kip`, `utcb` and `pid`. The `kip` and `utcb` arguments are used to specify the location of the `kip` and `utcb` of the new EAS. The `pid` argument is used to specify the process-id, which is a special feature of the ARM fast context switching extensions. For further information regarding process-ids, refer to the *ARM Architecture Reference Manual*.

The `eas_create()` method returns a reference to the newly created EAS on successful completion.

### B-3.3 eas\_delete()

```
void eas_delete(eas_ref_t eas);
```

The `eas_delete()` function may be used by the owning protection domain to delete an existing external address space. This function requires a single argument `eas`, which is used to specify a reference to the external address space to be deleted.

### B-3.4 eas\_create\_thread()

```
L4_ThreadId_t eas_create_thread(eas_ref_t eas, L4_ThreadId_t pager,  
                               L4_ThreadId_t scheduler, void *utcb,  
                               L4_ThreadId_t *handle_rv);
```

The `eas_create_thread()` function is used to create a new thread in a particular external address space. This function requires the arguments `eas`, `pager`, `scheduler` and a pointer to the `utcb`. The `eas` argument is used to specify the external address space in which the thread with the `pager` and `scheduler`, specified by the arguments `pager` and `scheduler`, is to be created. The `eas_create_thread()` function returns the `L4_ThreadId` of the newly created thread upon successful completion.

Upon successful completion, the `eas_create_thread()` function returns the global thread id of the newly created thread in the function return value, and a thread handle to the newly created thread in the `handle_rv` argument.

### B-3.5 eas\_delete\_thread()

```
void eas_delete_thread(eas_ref_t eas, L4_ThreadId_t thread);
```

The `eas_delete_thread()` function is used to delete a particular thread residing in a particular external address space. This function requires the arguments `eas` and `thread`. The `eas` argument is used to specify a reference to the external address space in which the thread specified by the argument, `thread`, resides.

### B-3.6 eas\_map()

```
int eas_map(eas_ref_t eas, L4_Fpage_t src_fpage, uintptr_t dst_addr,  
            uintptr_t attributes);
```

The `eas_map()` function is used to set up a mapping in a particular external address space. This function requires the arguments `eas`, `src_fpage`, `dst_addr` and `attributes`. The `eas` argument is used to specify a reference to the external address space in which the mapping is to be set up. Iguana maps the page specified by the `src_fpage` argument into the external address space specified by the `eas` argument, at the address specified by the argument `dst_addr`. The argument, `attributes`, is used to specify the OKL4 memory attributes of the newly created mapping. Memory attributes are further described in Section A-1.1.2.

The `eas_map()` function returns 0 on successful completion, and a non-zero value on failure.

### B-3.7 eas\_unmap()

```
void eas_unmap(eas_ref_t eas, L4_Fpage_t dst_fpage);
```

The `eas_unmap()` function is used to unmap existing mappings in a particular external address space. This function requires the arguments `eas` and `dst_fpage`. Iguana unmaps all existing mappings in the region specified by `dst_fpage` argument, of the external address space specified by the `eas` argument.

### B-3.8 `eas_share_domain()`

```
int eas_share_domain(eas_ref_t eas, L4_Fpage_t src_fpage);
```

The `eas_share_domain()` function is used to allow the calling protection domain access to the memory range specified by the `src_fpage` within the external address space specified by the `eas` argument. It should be noted that the `src_fpage` must be 1MB in size and aligned on a 1MB boundary. This function returns 0 on success and 1 on failure.

### B-3.9 `eas_unshare_domain()`

```
void eas_unshare_domain(eas_ref_t eas, L4_Fpage_t src_fpage);
```

The `eas_unshare_domain()` function is used to revoke the access of the calling protection domain to the memory range specified by the `src_fpage` argument in the external address space specified by the `eas` argument.

### B-3.10 `eas_modify()`

```
uintptr_t eas_modify(eas_ref_t eas, int pid);
```

The `eas_modify()` is used to change the PID register on ARM9 systems. The `eas_modify()` function returns 0 on success and 1 on failure.

## B-4 The <env.h> Header

---

### B-4.1 The Environment Item Type

```
ENV_CONSTANT
ENV_MEMSECTION
ENV_THREAD
ENV_THREAD_ID
ENV_VIRTPOOL
ENV_PHYSPOOL
ENV_ELF_SEGMENT
ENV_CLIST
ENV_ELF_FILE
```

Iguana allows for eight types of environment items to be specified. The entry item determines the functions that may be used to access the information contained within the environment entry.

### B-4.2 `iguana_getenv()`

```
const envitem_t *iguana_getenv(const char *name);
```

The `iguana_getenv()` function is used to obtain the environment item corresponding to the string, `name`. The `iguana_getenv()` function returns a pointer to the opaque data type, `envitem_t` on success and `NULL` on failure.

### B-4.3 `env_get_next()`

```
const envitem_t *env_get_next(const envitem_t* last);
```

The `env_get_next()` function is used to iterate through the environment. This function will return the next item in the environment or `NULL` if the end of the list is reached. The first item of the list may be obtained by calling `env_get_next()` with the argument `last` specified as `NULL`. It should be noted that the object environment is unordered.

### B-4.4 `env_name()`

```
const char *env_name(const envitem_t *item);
```

The `env_name()` function is used to obtain the name corresponding to a particular environment item. The `env_name()` function returns the name corresponding to the environment item specified in the argument `item`.

### B-4.5 `env_type()`

```
envtype_t env_type(const envitem_t *item);
```

The `env_type()` function is used to determine the type of a particular environment item. A list of the environment types is provided in Section B-4.1.

### B-4.6 `env_const()`

```
uintptr_t env_const(const envitem_t *item);
```

The `env_const()` function is used to obtain the value of the constant described by an environment item of type `ENV_CONSTANT`. A list of the environment types is provided in Section B-4.1.

### B-4.7 `env_memsection()`

```
memsection_ref_t env_memsection(const envitem_t *item);
```

The `env_memsection()` function is used to obtain the memory section reference of the memory section described by an environment item of type `ENV_MEMSECTION`. A list of the environment types is provided in Section B-4.1.

### B-4.8 `env_memsection_base()`

```
void *env_memsection_base(const envitem_t *item);
```

The `env_memsection_base()` function is used to obtain the base address of the memory section described by an environment item of type `ENV_MEMSECTION`. A list of the environment types is provided in Section B-4.1.

### B-4.9 `env_memsection_size()`

```
size_t env_memsection_size(const envitem_t *item);
```

The `env_memsection_size()` function is used to obtain the size of the memory section described by an environment item of type `ENV_MEMSECTION`, in bytes. A list of the environment types is provided in Section B-4.1.

### B-4.10 env\_thread()

```
L4_ThreadId_t env_thread_id(const envitem_t *item);
```

The `env_thread_id()` function is used to obtain the Iguana thread reference of the thread described by an environment item of type `ENV_THREAD`. A list of the environment types is provided in Section B-4.1.

### B-4.11 env\_thread\_id()

```
thread_ref_t env_thread(const envitem_t *item);
```

The `env_thread_id()` function is used to obtain an `L4_ThreadId_t` that attributes to the thread described by an environment item type of `ENV_THREAD_ID`. A list of the environment types is provided in Section B-4.1.

### B-4.12 env\_virtpool()

```
virtpool_ref_t env_virtpool(const envitem_t *item);
```

The `env_virtpool()` function is used to obtain a reference of the virtual memory pool described by an environment item of type `ENV_VIRTPOOL`. A list of the environment types is provided in Section B-4.1.

### B-4.13 env\_physpool()

```
physpool_ref_t env_physpool(const envitem_t *item);
```

The `env_physpool()` function is used to obtain a reference to the physical memory pool described by an environment item of type `ENV_PHYSPOOL`. A list of the environment types is provided in Section B-4.1.

### B-4.14 env\_clist()

```
clist_ref_t env_clist(const envitem_t *item);
```

The `env_clist()` function is used to obtain a reference to the capability list described by an environment item of type `ENV_CLIST`. A list of the environment types is provided in Section B-4.1.

### B-4.15 env\_elf\_segment\_paddr()

```
uintptr_t env_elf_segment_paddr(const envitem_t *item);
```

The `env_elf_segment_paddr()` function is used to obtain the physical address within the ELF file.

### B-4.16 `env_elf_segment_offset()`

```
uintptr_t env_elf_segmenet_offset(const envitem_t *item);
```

The `env_elf_segment_offset()` function is used to obtain the offset within the ELF file.

### B-4.17 `env_elf_segment_filesz()`

```
uintptr_t env_elf_segment_filesz(const envitem_t *item);
```

The `env_elf_segment_filesz()` function is used to obtain the file size of the segment.

### B-4.18 `env_elf_segment_memsz()`

```
uintptr_t env_elf_segment_memsz(const envitem_t *item);
```

The `env_elf_segment_memsz()` function is used to obtain the memory size of the segment.

### B-4.19 `env_elf_segment_vaddr()`

```
uintptr_t env_elf_segment_vaddr(const envitem_t *item);
```

The `env_elf_segment_vaddr()` function is used to obtain the base virtual address of the ELF segment described by an environment item of type `ENV_ELF_SEGMENT`. A list of the environment types is provided in Section B-4.1.

### B-4.20 `env_elf_segment_flags()`

```
uintptr_t env_elf_segment_flags(const envitem_t *item);
```

The `env_elf_segment_flags()` function is used to obtain the flags of the ELF segment described by an environment item of type `ENV_ELF_SEGMENT`. A list of the environment types is provided in Section B-4.1.

### B-4.21 `env_elf_file_type()`

```
uintptr_t env_elf_file_type(const envitem_t *item);
```

The `env_elf_file_type()` function is used to obtain the type of an ELF file described by an environment item of type `ENV_ELF_FILE`. This function will return a value of either `ELF_TYPE_EXTENSION` or `ELF_TYPE_PROGRAM`. The `ELF_TYPE_EXTENSION` to indicate that the ELF file described by the environment item is an Iguana server extension. The `ELF_TYPE_PROGRAM` is used to indicate that the ELF file described by the environment item is an ordinary program. A list of the environment types is provided in Section B-4.1.

### B-4.22 `env_elf_file_entry()`

```
uintptr_t env_elf_file_entry(const envitem_t *item);
```

The `env_elf_file_entry()` function is used to obtain the entry point to the ELF file described by an environment item of type `ENV_ELF_FILE`. A list of the environment types is provided in Section B-4.1.

### B-4.23 `env_zone()`

```
zone_ref_t env_zone(const envitem_t *item);
```

The `env_zone()` function is used to obtain a zone reference from an object environment item.

---

## B-5 The <hardware.h> Header

---

### B-5.1 hardware\_register\_interrupt()

```
int hardware_register_interrupt(L4_ThreadId_t thread, int interrupt);
```

The `hardware_register_interrupt()` function is used to register a particular thread as the interrupt handler thread for a particular interrupt. This function requires the arguments `thread` and `interrupt`. The `thread` argument is used to specify the `L4_ThreadId` of the thread to be registered as the interrupt handler. The `interrupt` argument is used to specify the number corresponding to the interrupt for which the thread is being registered. The `hardware_register_interrupt()` function returns 0 on successful completion and a non-zero value on failure.

### B-5.2 hardware\_back\_memsection()

```
int hardware_back_memsection(memsection_ref_t memsection,  
                             uintptr_t paddr, uintptr_t attributes);
```

The `hardware_back_memsection()` function is used to establish a mapping between the specified memory section and physical memory. This function requires the arguments `memsection`, `paddr` and `attributes`. The `paddr` argument is used to specify the physical address at which the mapping of the memory section, specified by the argument `memsection`, commences. This function changes the attributes of the memory section specified by the `memsection` argument to the OKL4 memory attributes specified by the `attributes` argument. Memory attributes are further described in Section A-1.1.2.

## B-6 The <memsection.h> Header

### B-6.1 Memory Types

MEM_NORMAL	0x1
MEM_FIXED	0x2
MEM_DIRECT	0x4
MEM_UTCB	0x8
MEM_USER	0x10

Iguana allows five different memory types. `MEM_NORMAL` is the default or standard memory type of memory sections in Iguana. Memory sections of type `MEM_FIXED` are fixed at a particular virtual address. Memory sections of the type `MEM_DIRECT` are located at the exact same virtual and physical address, that is they have one to one virtual to physical mapping. The memory type `MEM_UTCB`, indicates that the memory section is used as the utcb of a thread. Lastly, memory sections of the type `MEM_USER` are used to describe memory sections that use their own pager to map virtual to physical memory rather than using the Iguana pager.

### B-6.2 memsection\_create()

```
memsection_ref_t memsection_create(uintptr_t size, uintptr_t *base);
```

The `memsection_create()` function is used to create a new memory section. This function requires the arguments `size` and `*base`. The `size` argument is used to specify the size of the new memory section in bytes. The `*base` argument is used by Iguana to return the base virtual address of the newly created memory section. The `memsection_create()` function returns a reference to the newly created memory section of memory type `MEM_NORMAL`, with the memory attributes `L4_DefaultMemory`, on successful completion. Memory attributes are further described in Section A-1.1.2.

### B-6.3 memsection\_create\_with\_pools()

```
memsection_ref_t memsection_create_with_pools(uintptr_t size,
                                              uintptr_t *base,
                                              physpool_ref_t physpool,
                                              virtpool_ref_t virtpool);
```

The `memsection_create_with_pools()` function is used to create a new memory section. This function requires the arguments `size`, `base`, `physpool` and `virtpool`. The `size` argument is used to specify the size of the memory section in bytes. The `base` argument is used to return the base virtual address of the newly created memory section. The `physpool` and `virtpool` arguments are used to specify the physical and virtual memory pools to be used for the allocation of the new memory section. The `memsection_create_with_pools()` function returns a reference to the newly created memory section on successful completion.

## B-6.4 memsection\_create\_user()

```
memsection_ref_t memsection_create_user(uintptr_t size,  
                                       uintptr_t *base);
```

The `memsection_create_user()` function is used to create a new memory section which uses its own custom pager thread to map virtual to physical memory instead of using the Iguana pager. This function requires the arguments `size` and `*base`. The `size` argument is used to specify the size of the new memory section in bytes and the `*base` argument is used by Iguana to return the base virtual address of the newly created memory section. The `memsection_create_user()` function returns a reference to the newly created memory section of type `MEM_USER`, with the memory attributes `L4_DefaultMemory`, on successful completion.

## B-6.5 memsection\_create\_dma()

```
memsection_ref_t memsection_create_dma(uintptr_t size,  
                                       uintptr_t *base,  
                                       physmem_ref_t *pm,  
                                       uintptr_t attributes);
```

The `memsection_create_dma()` function is used to create a new memory section. This function also creates a `Physmem` object and uses it to contiguously back the newly created memory section. This function requires the arguments `size`, `base`, `pm` and `attributes`. The `size` argument is used to specify the size of the memory section in bytes. The `base` argument is used to return the base virtual address of the memory section and the `pm` argument is used to return a reference to the `Physmem` object. The `attributes` argument is used to specify the attributes to be associated with the newly created memory section.

The `memsection_create_dma()` function returns a reference to the newly created memory section on successful completion.

## B-6.6 memsection\_create\_direct()

```
memsection_ref_t memsection_create_direct(uintptr_t size,  
                                       uintptr_t *base);
```

The `memsection_create_direct()` function is used to create a new memory section that has the same virtual and physical address, that is, the memory section has a one to one virtual to physical mapping. This function requires the arguments `size` and `*base`. The `size` argument is used to specify the size of the new memory section in bytes and the `*base` argument is used by Iguana to return the base virtual address of the newly created memory section. The `memsection_create_direct()` function returns a reference to the newly created memory section of type `MEM_DIRECT`, with the memory attributes `L4_DefaultMemory`, on successful completion.

## B-6.7 memsection\_create\_fixed()

```
memsection_ref_t memsection_create_fixed(uintptr_t size,  
                                         uintptr_t base);
```

The `memsection_create_fixed()` function is used to create a new memory section that is fixed at a particular virtual memory location. This function requires the arguments `size` and `base`. The `size` argument is used to specify the size of the new memory section in bytes. The `base` argument may be used to specify the base virtual address of the new memory section. Setting the `base` argument to 0 allows Iguana to determine the virtual memory location of the memory section. The `memsection_create_fixed()` function returns a reference to the newly created memory section of type `MEM_FIXED`, with the memory attributes `L4_DefaultMemory`, on successful completion.

## B-6.8 memsection\_create\_fixed\_user()

```
memsection_ref_t memsection_create_fixed_user(uintptr_t size,  
                                              uintptr_t base);
```

The `memsection_create_fixed_user()` function is used to create a new memory section that is fixed at a particular virtual address and uses its own custom pager thread to map virtual to physical memory instead of using the Iguana pager. This function requires the arguments `size` and `base`. The `size` argument is used to specify the size of the new memory section in bytes. The `base` argument is used to specify the base virtual address of the newly created memory section. Setting the `base` argument to 0 allows Iguana to determine the virtual memory location of the new memory section. The `memsection_create_fixed_user()` function returns a reference to the newly created memory section of type `MEM_FIXED + MEM_USER`, with the memory attributes `L4_DefaultMemory` on successful completion.

## B-6.9 memsection\_create\_in\_zone()

```
memsection_ref_t memsection_create_in_zone(uintptr_t size,  
                                           uintptr_t *base,  
                                           zone_ref_t zone);
```

The `memsection_create_in_zone()` function is used to create a new memory section in the specified zone. This function requires the arguments `size`, `base` and `zone`. The `size` and `zone` arguments are used to specify the size and zone of the memory section to be created. Iguana uses the argument, `base`, to return the base virtual address of the newly created memory section.

## B-6.10 memsection\_register\_server()

```
int memsection_register_server(memsection_ref_t memsection,  
                              thread_ref_t server);
```

The `memsection_register_server()` function is used to register a particular thread as the server thread for a particular memory section. This function requires the arguments `memsection` and `server`.

The `memsection` argument is used to specify a reference to the memory section for which the thread specified by the `server` argument is registered as the server thread. The `memsection_register_server()` function returns 0 on successful completion and a non-zero value on failure.

### B-6.11 `memsection_virt_to_phys()`

```
uintptr_t memsection_virt_to_phys(uintptr_t vaddr, size_t *size);
```

The `memsection_virt_to_phys()` function is used to obtain the base physical address of the Fpage backing a particular memory section at the virtual address specified by the `vaddr` argument. This function also returns the size of the Fpage in bytes using the argument `size`.

### B-6.12 `memsection_lookup()`

```
memsection_ref_t memsection_lookup(objref_t object,  
                                  thread_ref_t *server);
```

The `memsection_lookup()` function is used to obtain a reference to the memory section in which a particular object resides as well as the Iguana thread reference of the server thread associated with the particular memory section. This function requires the arguments `object` and `*server`. The `object` argument is used to specify a reference to the object. The `*server` argument is used by Iguana to return the Iguana thread reference of the server thread associated with the memory section containing the object. The `memsection_lookup()` function returns a reference to the memory section containing the specified object on successful completion.

### B-6.13 `memsection_delete()`

```
void memsection_delete(memsection_ref_t memsection);
```

The `memsection_delete()` function is used to delete a particular memory section. This function requires a single argument, `memsection`, which is used to specify a reference to the memory section to be deleted.

### B-6.14 `memsection_set_attributes()`

```
void memsection_set_attributes(memsection_ref_t memsection,  
                              uintptr_t attr);
```

The `memsection_set_attributes()` function is used to associate a particular set of attributes with a particular memory section. This function requires the arguments `attr` and `memsection`. The `attr` argument is used to specify the set of memory attributes and the `memsection` argument is used to specify a reference to the memory section. Memory attributes are further described in Section A-1.1.2.

### B-6.15 memsection\_base()

```
void *memsection_base(memsection_ref_t memsection);
```

The `memsection_base()` function is used to obtain the base virtual address of a particular memory section. This function requires a single argument, `memsection`, which is used to specify a reference to the memory section. The `memsection_base()` function returns the base virtual address of the specified memory section on successful completion.

### B-6.16 memsection\_size()

```
uintptr_t memsection_size(memsection_ref_t memsection);
```

The `memsection_size()` function is used to obtain the size of a particular memory section in bytes. This function requires a single argument, `memsection`, which is used to specify a reference to the memory section. The `memsection_size()` function returns the size of the specified memory section on successful completion.

### B-6.17 memsection\_map()

```
int memsection_map(memsection_ref_t memsection,  
                  uintptr_t offset,  
                  physmem_ref_t phys);
```

The `memsection_map()` function is used to map a particular `Physmem` object at a specified offset to a particular memory section. This function requires the arguments `memsection`, `offset` and `phys`. Iguana maps the `Physmem` object specified by the argument `phys`, at the offset specified by the `offset` argument, to the memory section specified by the `memsection` argument. The `memsection_map()` function returns 0 on successful completion and a non-zero value on failure.

### B-6.18 memsection\_unmap()

```
int memsection_unmap(memsection_ref_t memsection,  
                    uintptr_t offset,  
                    uintptr_t size);
```

The `memsection_unmap()` function is used to unmap a specified range of a particular memory section. This function requires the arguments `memsection`, `offset` and `size`. The `memsection` argument is used to specify a reference to the memory section. The `offset` argument is used to specify the offset within the memory section at which the unmapping is to commence. The `size` argument is used to specify the size of the memory region to be unmapped. The `memsection_unmap()` function returns 0 on successful completion and a non-zero value on failure.

## B-6.19 memsection\_page\_map()

```
int memsection_page_map(memsection_ref_t memsection,
                        L4_Fpage_t from_page,
                        L4_Fpage_t to_page);
```

The `memsection_page_map()` function is used to map an `L4_Fpage` to a particular memory section. This function requires the arguments `memsection`, `from_page` and `to_page`. This function maps the `L4_Fpage` specified by the `from_page` argument to the `L4_Fpage` specified by the `to_page` argument. The `memsection` argument is used specify a reference to the memory section containing the `L4_Fpage` specified by the `to_page` argument. The `memsection_page_map()` function returns 0 on successful completion and a non-zero value on failure.

## B-6.20 memsection\_page\_unmap()

```
int memsection_page_unmap(memsection_ref_t memsection,
                          L4_Fpage_t page);
```

The `memsection_page_unmap()` function is used to unmap an `L4_Fpage` from a particular memory section. This function requires the arguments `memsection` and `page`. The `page` argument is used to specify the `L4_Fpage` to be unmapped from the memory section specified by the `memsection` argument. The `memsection_page_unmap()` function returns 0 success and a non-zero value on failure. Once this function has returned successfully, the `L4_Fpage` specified by the `page` argument will no longer be backed and will result in a page fault if it is accessed.

## B-6.21 memsection\_back\_range()

```
int memsection_back_range(memsection_ref_t memsection,
                          uintptr_t base, uintptr_t end);
```

The `memsection_back_range()` function is used to back a selected range of memory. This function requires the arguments `memsection`, `base` and `end`. The arguments `base` and `end` are used to specify the memory range in the memory section, specified by the `memsection` argument, to be backed. Iguana will back the specified memory range rounded up to the nearest multiple of the base page size. The `memsection_back_range()` function returns 0 on successful completion and a non-zero value on failure.

---

## B-7 The mutex.h Header

---

### B-7.1 okl4\_mutex\_allocate()

```
int okl4_mutex_allocate(L4_MutexId_t *id);
```

The `okl4_mutex_allocate()` function is used to allocate an OKL4 kernel mutex object. This function requires a single argument, `id`, which is used by Iguana to return a reference to the newly created OKL4 kernel mutex object. The `okl4_mutex_allocate()` function returns 0 on successful completion and an error code on failure.

### B-7.2 okl4\_mutex\_deallocate()

```
int okl4_mutex_deallocate(L4_MutexId_t *id);
```

The `okl4_mutex_deallocate()` function is used to free a previously allocated OKL4 kernel mutex object. This function requires a single argument, `id`, which is used to specify the mutex identifier corresponding to the OKL4 kernel mutex object to be deleted. The `okl4_mutex_deallocate()` function returns 0 on successful completion and error code on failure.

---

## B-8 The <pd.h> Header

---

### B-8.1 `pd_myself()`

```
pd_ref_t pd_myself(void);
```

The `pd_myself()` function may be used to obtain a reference to the current protection domain. This function does not require any arguments and returns a reference to the current protection domain on successful completion.

### B-8.2 `pd_create()`

```
pd_ref_t pd_create(void);
```

The `pd_create()` function is used to create a new protection domain that is owned by the current protection domain. This function does not require any arguments and returns a reference to the newly created protection domain on successful completion.

### B-8.3 `pd_create_pd()`

```
pd_ref_t pd_create_pd(pd_ref_t pd);
```

The `pd_create_pd()` function is used to create a new protection domain that is owned by a particular protection domain. This function requires a single argument, `pd` which is used to specify a reference to the owning protection domain. The `pd_create_pd()` function returns a reference to the newly created protection domain on successful completion.

### B-8.4 `pd_create_restricted()`

```
pd_ref_t pd_create_restricted(void);
```

The `pd_create_restricted()` function is used to create a new *restricted* protection domain that is owned by the current protection domain. This function does not require any arguments and returns a reference to the newly created restricted protection domain on successful completion.

## B-8.5 `pd_create_restricted_in_pd()`

```
pd_ref_t pd_create_restricted_in_pd(pd_ref_t pd);
```

The `pd_create_restricted_in_pd()` function may be used to create a new *restricted* protection domain that is owned by a specific protection domain. This function requires a single argument, `pd`, which is used to specify a reference to the owning protection domain. The `pd_create_restricted_in_pd()` function returns a reference to the newly created restricted protection domain on successful completion.

## B-8.6 `pd_delete()`

```
void pd_delete(pd_ref_t pd);
```

The `pd_delete()` function is used to delete a specific protection domain. This function requires a single argument, `pd`, which is used to specify a reference to the protection domain to be deleted.

## B-8.7 `pd_create_thread()`

```
thread_ref_t pd_create_thread(pd_ref_t pd, L4_ThreadId_t *thread);
```

The `pd_create_thread()` function is used to create a new thread in a specific protection domain. This function requires the arguments `pd` and `*thread`. The `pd` argument is used to specify the protection in which the thread is to be created. Iguana uses the `*thread` argument to return the `L4_ThreadId` of the newly created thread. This function returns the Iguana thread reference of the newly created thread on successful completion.

## B-8.8 `pd_create_thread_with_priority()`

```
thread_ref_t pd_create_thread_with_priority(pd_ref_t pd,  
                                           int priority,  
                                           L4_ThreadId_t *thread);
```

The `pd_create_thread_with_priority()` function is used to create a new thread of a specified priority in a specific protection domain. This function requires the arguments `pd`, `priority` and `*thread`. The argument `pd` is used to specify the protection domain in which the new thread is to be created. The `priority` argument is used to specify the priority level of the new thread, which must be an integer between 0 and 255. Iguana uses the `*thread` argument to return the `L4_ThreadId` of the newly created thread. The `pd_create_thread_with_priority()` function returns the Iguana thread reference of the newly created thread on successful completion.

## B-8.9 `pd_create_memsection()`

```
memsection_ref_t pd_create_memsection(pd_ref_t pd,
                                      uintptr_t size,
                                      uintptr_t *base);
```

The `pd_create_memsection()` function is used to create a new memory section of type `MEM_NORMAL`, in a particular protection domain. This function requires the arguments `pd`, `size` and `*base`. The `pd` argument is used to specify the protection domain in which the new memory section is to be created. The `size` argument is used to specify the size of the new memory section in bytes. The `*base` argument is used by Iguana to return the base virtual address of the newly created memory section. The `pd_create_memsection()` function returns a reference to the newly created memory section on successful completion. Memory types are further described in Section A-1.1.1.

## B-8.10 `pd_create_memsection_user()`

```
memsection_ref_t pd_create_memsection_user(pd_ref_t pd,
                                           uintptr_t size,
                                           uintptr_t *base);
```

The `pd_create_memsection_user()` function is used to create a new memory section of the memory type `MEM_USER`, in a particular protection domain. This function requires the arguments `pd`, `size` and `*base`. The `pd` argument is used to specify the protection domain in which the new memory section is to be created. The `size` argument is used to specify the size of the new memory section in bytes. Iguana uses the `*base` argument to return the base virtual address of the newly created memory section. The `pd_create_memsection_user()` function returns a reference to the newly created memory section on successful completion. Memory types are further described in Section A-1.1.1.

## B-8.11 `pd_create_memsection_fixed_user()`

```
memsection_ref_t pd_create_memsection_fixed_user(pd_ref_t pd,
                                                 uintptr_t size,
                                                 uintptr_t base);
```

The `pd_create_memsection_fixed_user()` function is used to create a new memory section of type `MEM_FIXED` and `MEM_USER`, which is fixed at a particular virtual address in the specified protection domain. This function requires the arguments `pd`, `size` and `base`. The `pd` argument is used to specify the protection domain in which the new memory section is to be created. The `size` argument is used to specify the size of the new memory section in bytes and the `base` argument is used to specify the base virtual address of the new memory section. This function returns a reference to the newly created memory section on successful completion. Memory types are further described in Section A-1.1.1.

## B-8.12 `pd_add_clist()`

```
uintptr_t pd_add_clist(pd_ref_t pd, memsection_ref_t clist);
```

The `pd_add_clist()` function is used to add a particular capability list to a specified protection domain. This function requires the arguments, `pd` and `clist`. The `pd` argument is used to specify the protection domain to which the capability list specified by the `clist` argument is to be added. This function returns a non-zero value on successful completion, and zero on failure.

## B-8.13 `pd_release_clist()`

```
void pd_release_clist(pd_ref_t pd, uintptr_t clist, int slot);
```

The `pd_release_clist()` function is used to delete a particular capability list from a specified protection domain. This function requires the arguments, `pd`, `clist` and `slot`. The `pd` argument is used to specify the protection domain from which the capability list specified by the argument, `clist` is to be removed. Please note that the `slot` argument is ignored by Iguana and it is expected that it will be omitted from this function in future versions of Iguana.

## B-8.14 `pd_attach()`

```
int pd_attach(pd_ref_t pd, memsection_ref_t ms, int rwx);
```

The `pd_attach()` function is used to make a particular memory section available to a specific protection domain. This function requires the arguments `pd`, `ms` and `rwx`. The `pd` argument is used to specify a reference to the protection domain to which the memory section specified by the `ms` argument will be made available. The `rwx` argument is used to specify the access rights with which the memory section is attached. Note that the permissions have been encoded using the standard OKL4 method, that is, bit 3 contains the write permission, bit 2 contains the read permission and bit 1 contains the execute permission. The `pd_attach()` function returns 0 on successful completion and an error code on failure.

## B-8.15 `pd_detach()`

```
void pd_detach(pd_ref_t pd, memsection_ref_t ms);
```

The `pd_detach()` function is used to remove a particular memory section from a specific protection domain. This function requires the arguments, `pd` and `ms`. The `pd` argument is used to specify a reference to the protection domain from which the memory section referred to by the `ms` argument is removed.

## B-8.16 `pd_attach_zone()`

```
int pd_attach_zone(pd_ref_t pd, zone_ref_t zone, int rwx);
```

The `pd_attach_zone()` function is used to attach a zone to a protection domain with specified access permissions. This function requires the arguments `pd`, `zone` and `rwx`. The `pd` argument is used to specify a reference to the protection domain to which the zone specified by the argument `zone` is to be attached. The zone is attached to the protection domain with the access permissions specified by the argument, `rwx`. Once attached, all memory sections belonging to the specified zone will also be attached to the protection domain with the specified rights. The `pd_attach_zone()` function returns 0 on success and an error code on failure.

## B-8.17 `pd_detach_zone()`

```
void pd_detach_zone(pd_ref_t pd, zone_ref_t zone);
```

The `pd_detach_zone()` function is used to detach a particular zone from a protection domain. This function requires the arguments `pd` and `zone`. The `pd` argument is used to specify the protection domain from which the zone specified by the argument `zone` is to be detached. All memory sections belonging to the specified zone will also be detached from the protection domain.

## B-8.18 `pd_extension_activate()`

```
int pd_extension_activate(void);
```

The `pd_extension_activate()` function is used to enable a thread to access the memory section containing the *protected* extension. If successful, this function makes the extension visible to the thread. It should be noted that each system may only contain a single extension. In addition, the extension may only be visible to a single thread at any one time. Once activated the extension continues to be visible to the thread until it is deactivated using the `pd_extension_deactivate()` function described below. The `pd_extension_activate()` function returns 0 on successful completion and a non-zero value on failure.

## B-8.19 `pd_extension_deactivate()`

```
int pd_extension_deactivate(void);
```

The `pd_extension_deactivate()` function is used to cease the visibility of the extension to the calling thread. Once the extension is deactivated it is available to be activated by a different thread as the extension is only visible to a single thread at any one time. The `pd_extension_deactivate()` function returns 0 on successful completion and a non-zero value on failure.

## B-8.20 `pd_l4id()`

```
L4_SpaceId_t pd_l4id(pd_ref_t pd);
```

The `pd_l4id()` function is used to obtain the `L4_SpaceId_t` corresponding to the protection domain specified by the argument `pd`.

---

## B-9 The <phymem.h> Header

---

### B-9.1 `phymem_delete()`

```
void phymem_delete(phymem_ref_t phymem);
```

The `phymem_delete()` function is used to delete an existing Phymem object. This function requires a single argument, `phymem`, which is used to specify a reference to the Phymem object to be deleted.

### B-9.2 `phymem_info()`

```
void phymem_info(phymem_ref_t phymem, uintptr_t *paddr,  
                uintptr_t *psize);
```

The `phymem_info()` function is used to obtain information about a specific Phymem object. This function requires the arguments `phymem`, `*paddr` and `*psize`. The `phymem` argument is used to specify a reference to the Phymem object. The `phymem_info()` function returns the base physical address of the Phymem object in the `*paddr` argument and its size in bytes in the `*psize` argument.

---

## B-10 The <physpool.h> Header

---

### B-10.1 physpool\_alloc()

```
phymem_ref_t physpool_alloc(physpool_ref_t physpool, uintptr_t size);
```

The `physpool_alloc()` function is used to create a new `Phymem` object in the current protection domain by allocating memory from a specified pool. This function requires the arguments `physpool` and `size`. The `physpool` argument is used to specify the memory pool from which the `Phymem` object is to be created. The `size` argument is used to specify the size of the `Phymem` object in bytes. The `physpool_alloc()` function returns a reference to the newly created `Phymem` object on successful completion and `INVALID_ADDR` on failure.

### B-10.2 physpool\_alloc\_fixed()

```
phymem_ref_t physpool_alloc_fixed(physpool_ref_t physpool,  
                                  uintptr_t size, uintptr_t addr);
```

The `physpool_alloc_fixed()` function is used to create a new `Phymem` object by allocating memory from the specified pool at the specified base address in the current protection domain. The `physpool` argument is used to specify the physical memory pool from which the new `Phymem` object is allocated. The `size` argument is used to specify the size of the `Phymem` object in bytes and the `base` argument is used to specify the base physical address of the `Phymem` object. The `physpool_alloc_fixed()` function returns a reference to the newly created `Phymem` object on successful completion.

---

## B-11 The <segment\_info.h> Header

---

### B-11.1 get\_elf\_info()

```
int get_elf_info(const char* name, uintptr_t* type, uintptr_t* entry);
```

The `get_elf_info()` function is used to obtain the type and/or entry point of a particular ELF file. This function requires the arguments `name`, which is used to specify the name of the ELF file, and `type` and `entry`, both of which are used to return information about the ELF file. The Iguana library uses the `type` argument to return either `PROGRAM` or `EXTENSION`, which may be used to determine whether the ELF file specified by the `name` argument is a regular program or an Iguana server extension. The `entry` argument is used to return entry point of the ELF file specified by the `name` argument.

The user may select the information they wish to receive by specifying either the `type` or the `entry` argument as `NULL`. This argument will then be ignored by Iguana. The `get_elf_info()` function returns 0 on success and -1 on failure.

### B-11.2 get\_segment\_info()

```
int get_segment_info(const char *name, uintptr_t *r_vaddr,  
                    uintptr_t *r_flags, uintptr_t *r_paddr,  
                    uintptr_t *r_offset, uintptr_t *r_filesz,  
                    uintptr_t *r_memsz);
```

The `get_segment_info()` function is used to obtain the base virtual address, flags, physical address, offset into the ELF file, file size and the memory size of the segment specified by the argument `name`. All other arguments are used to return information about ELF segment.

The user may select the information they wish to receive by setting the argument of the value they do not wish to receive to `NULL`. These arguments will then be ignored by Iguana. The `get_segment_info()` function returns 0 on success and -1 on failure.

---

## B-12 The <thread.h> Header

---

### B-12.1 `thread_create()`

```
thread_ref_t thread_create(L4_ThreadId_t *thrd);
```

The `thread_create()` function is used to create a new inactive thread in the current protection domain. This function requires a single argument, `*thrd`, which is used by Iguana to return `L4_ThreadId` of the newly created thread. This thread may be subsequently activated using the `OKL4_ExchangeRegisters()` system call or alternatively, by using the `thread_start()` function described in Section B-12.4. The `OKL4_ExchangeRegisters()` system call is further described in the *OKL4 Programming Manual*. The `thread_create()` function returns the Iguana thread reference of the newly created thread on successful completion.

### B-12.2 `thread_create_priority()`

```
thread_ref_t thread_create_priority(int priority,  
                                   L4_ThreadId_t *thrd);
```

The `thread_create_priority()` function is used to create a new thread with a specific priority level. This function requires the arguments `priority` and `*thrd`. The `priority` argument is used to specify the priority of the new thread. The valid range of thread priority levels in the Iguana object environment range between the values 0 and 255, where 255 is the highest priority level. Iguana uses the `*thrd` argument to return the `L4_ThreadId` of the newly created thread. The `thread_create_priority()` function returns the Iguana thread reference of the newly created thread on successful completion.

### B-12.3 `thread_create_simple()`

```
thread_ref_t thread_create_simple(void (*fn) (void *),  
                                  void *arg, int priority);
```

The `thread_create_simple()` function is used to create a thread to run a specified function with a specified set of arguments. This function requires the function, the arguments and the priority at which the thread is run. The `thread_create_simple()` function returns the Iguana thread reference of the newly created thread on successful completion.

## B-12.4 `thread_start()`

```
void thread_start(thread_ref_t thread, uintptr_t ip,  
                 uintptr_t sp);
```

The `thread_start()` function is used to activate a particular thread. This function requires the arguments `thread`, `ip` and `sp`. The `thread` argument is used to specify the thread that is to be activated. The `ip` and `sp` arguments are used to specify the instruction pointer and stack pointer of the thread, respectively. Once the specified thread is activated, it is placed in the scheduling queue.

## B-12.5 `thread_myself()`

```
thread_ref_t thread_myself(void);
```

The `thread_myself()` function is used to obtain the Iguana thread reference of the current thread.

## B-12.6 `thread_l4tid()`

```
L4_ThreadId_t thread_l4tid(thread_ref_t thread);
```

The `thread_l4tid()` function is used to obtain the `L4_ThreadId` of a particular thread. This function requires a single argument, `thread`, which is used to specify the Iguana thread reference of the thread. The `thread_l4tid()` function returns the `L4_ThreadId_t` of the specified thread on successful completion.

## B-12.7 `thread_id()`

```
thread_ref_t thread_id(L4_ThreadId_t thread);
```

The `thread_id()` function is used to obtain the Iguana thread reference of a particular thread. This function requires a single argument, `thread`, which is used to specify the `L4_ThreadId` of the thread. The `thread_id()` function returns the Iguana thread reference of the specified thread on successful completion.

## B-12.8 `thread_delete()`

```
void thread_delete(L4_ThreadId_t thread);
```

The `thread_delete()` function is used to delete a particular thread. This function requires a single argument, `thread`, which is used to specify the `L4_ThreadId` of the thread to be deleted.

## B-12.9 `thread_delete_self()`

```
void thread_delete_self(void);
```

The `thread_delete_self()` function is used to delete the current thread.

## B-13 The `<tls.h>` Header

The `tls.h` header describes the thread local storage in Iguana. Some functions require Iguana threads to be set up with thread local storage. This includes `ERRNO` in `libc`.

As illustrated in Figure B-13.1 below, threads can be set up with thread local storage by storing a pointer to the first function called by the thread, followed by the arguments required by that function in the threads stack, and setting the instruction pointer of the thread to call the function `__thread_stub`.

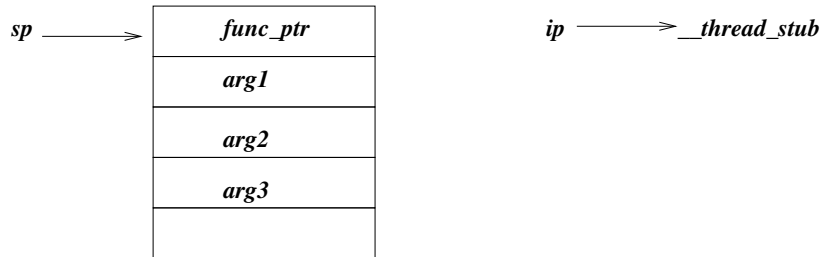


Figure B-13.1: *Initializing the TLS*

As illustrated in Figure B-13.2 below, a thread that has been set up with thread local storage will contain a pointer to the TLS structure. The TLS structure is a range of memory that has been allocated from the heap and contains data, which are usually pointers. For example, in Figure B-13.2, the second entry which contains the key `TLS_TIMER_KEY`, points to the Timer TLS.

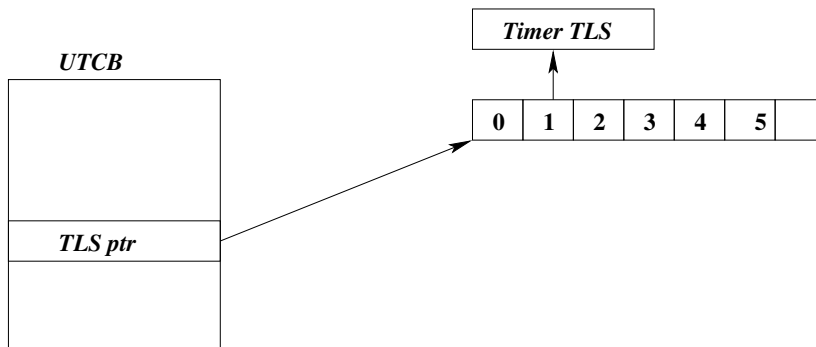


Figure B-13.2:

The TLS keys available in Iguana are listed below.

***TLS Keys***

TLS_ERRNO_KEY	0
TLS_TIMER_KEY	1
TLS_NAMING_KEY	2
TLS_SYNCH_BITS_KEY	3
TLS_THREAD_ID	6

---

## B-14 The <zone.h> Header

---

### B-14.1 zone\_create()

```
zone_ref_t zone_create(void);
```

The `zone_create()` function is used to create a new zone. This function does not require any arguments and returns a reference to the newly created zone on successful completion. Zones are further described in Section A-1.5.

### B-14.2 zone\_delete()

```
void zone_delete(zone_ref_t zone);
```

The `zone_delete()` function is used to delete a particular zone. This function requires a single argument, `zone`, which is used to specify a reference to the zone to be deleted. Zones are further described in Section A-1.5.

# **PART C**

## **Appendix**



## C-1 Iguana IDL Description

---

Iguana provides the interface identifiers listed below. Each interface is described in the following Sections. The name of each function available in each interface are listed in a table at the beginning of the Section along with the parameters both in and out required or supplied by the function, the return value if any and a reference to the description of the function within the section.

The Interface Identifiers described in the following sections are listed below.

```
INTERFACE_IGUANA_PD_UUID  
INTERFACE_IGUANA_EAS_UUID  
INTERFACE_IGUANA_THREAD_UUID  
INTERFACE_IGUANA_HARDWARE_UUID  
INTERFACE_IGUANA_MEMSECTION_UUID  
INTERFACE_IGUANA_QUOTA_UUID  
INTERFACE_IGUANA_PHYSPOOL_UUID  
INTERFACE_IGUANA_PHYSMEM_UUID  
INTERFACE_IGUANA_CLIST_UUID
```

## C-1.1 INTERFACE\_IGUANA\_PD\_UUID

INTERFACE_IGUANA_PD_UUID				
Method:	Parameters(in):	Parameters(out):	Return:	Section:
<code>mypd()</code>	<code>()</code>	<code>()</code>	<code>objref_t</code>	C-1.1.1
<code>create_memsection()</code>	<code>(pd, sz, bse, flgs, pp, vp, clst)</code>	<code>(base_out)</code>	<code>cap_t</code>	C-1.1.2
<code>create_pd()</code>	<code>(pd, clist)</code>	<code>()</code>	<code>cap_t</code>	C-1.1.3
<code>create_thread()</code>	<code>(pd, priority, clist)</code>	<code>(l4_tid)</code>	<code>cap_t</code>	C-1.1.4
<code>create_eas()</code>	<code>(pd, kip, utcb, clist)</code>	<code>(l4_id)</code>	<code>cap_t</code>	C-1.1.5
<code>create_eas()</code>	<code>(pd, kip, utcb, clist, pid)</code>	<code>(l4_id)</code>	<code>cap_t</code>	C-1.1.6
<code>set_callback()</code>	<code>(pd, callback_buffer)</code>	<code>()</code>	<code>void</code>	C-1.1.7
<code>add_clist()</code>	<code>(pd, clist)</code>	<code>()</code>	<code>uintptr_t</code>	C-1.1.8
<code>release_clist()</code>	<code>(pd, clist, slot)</code>	<code>()</code>	<code>void</code>	C-1.1.9
<code>delete()</code>	<code>(pd)</code>	<code>()</code>	<code>void</code>	C-1.1.10
<code>create_restricted_pd()</code>	<code>(pd, clist)</code>	<code>()</code>	<code>cap_t</code>	C-1.1.11
<code>attach()</code>	<code>(pd, ms, rwx)</code>	<code>()</code>	<code>int</code>	C-1.1.12
<code>detach()</code>	<code>(pd, ms)</code>	<code>()</code>	<code>void</code>	C-1.1.13

### C-1.1.1 mypd()

The `mypd()` method returns a reference to the current protection domain, that is, the owning protection domain of the calling thread. This method does not require any parameters.

### C-1.1.2 create\_memsection()

The `create_memsection()` method creates a new memory section of the size specified by the `size` parameter, aligned to the nearest page size boundary. The newly created memory section will belong to the protection domain specified by the parameter `pd`.

### C-1.1.3 create\_pd()

The `create_pd()` method creates a new protection domain owned by the current protection domain, that is, the protection domain owning the calling thread.

### C-1.1.4 create\_thread()

The `create_thread()` method creates a new thread of the specified priority in the current protection domain, that is, the protection domain owning the calling thread. This method returns the `L4_ThreadId` of the newly created thread in an output parameter. The Iguana thread reference of the newly created thread is returned to the caller as the return value.

### C-1.1.5 create\_eas()

The `create_eas()` method creates a new external address space owned by the specified protection domain.

### C-1.1.6 create\_eas()

The `create_eas()` method creates a new external address space owned by the specified protection domain. Note that this method requires an extra argument `pid` and is used only for ARM processors.

### C-1.1.7 set\_callback()

The `set_callback()` method sets the call back buffer of the of the target protection domain specified by the `pd` parameter to the memory section specified by the parameter, `callback_buffer`.

**C-1.1.8 add\_clist()**

The `add_clist()` method adds the capability list specified by the `clist` parameter to the protection domain specified by the parameter, `pd`.

**C-1.1.9 release\_clist()**

The `release_clist()` method removes the capability list specified by the `clist` parameter from the protection domain specified by the parameter, `pd`. The `slot` parameter is currently ignored by Iguana.

**C-1.1.10 delete()**

The `delete()` method deletes the protection domain specified by the `pd` parameter and all it encompasses as well as any other protection domains and external address spaces owned by it.

**C-1.1.11 create\_restricted\_pd()**

The `create_restricted_pd()` method creates a new restricted protection domain owned by the protection domain specified by the parameter, `pd`. A restricted protection domain may only communicate with its owning protection domain and Iguana.

**C-1.1.12 attach()**

The `attach()` method is used to attach a memory section specified by the parameter `ms`, to the protection domain specified by the parameter, `pd`.

**C-1.1.13 detach()**

The `detach()` method is used to remove the memory section specified by the `ms` parameter from the protection domain specified by the parameter, `pd`.

## C-1.2 INTERFACE\_IGUANA\_EAS\_UUID

INTERFACE_IGUANA_EAS_UUID				
Method:	Parameters(in):	Parameters(out):	Return:	Section:
create_thread()	( <i>eas</i> , <i>pgr</i> , <i>scheduler</i> , <i>utcb</i> , <i>clst</i> )	( <i>l4_tid</i> , <i>l4_handle</i> )	<i>cap_t</i>	C-1.2.1
delete()	( <i>eas</i> )	()	void	C-1.2.2
map()	( <i>eas</i> , <i>src_fpage</i> , <i>dst_addr</i> , <i>attr</i> )	()	int	C-1.2.3
unmap()	( <i>eas</i> , <i>dst_fpage</i> )	()	void	C-1.2.4

### C-1.2.1 create\_thread()

The `create_thread()` method is used to create a new thread in the external address space specified by the input parameter, *eas*.

### C-1.2.2 delete()

The `delete()` method is used to delete the external address space specified by the input parameter, *eas*.

### C-1.2.3 map()

The `map()` method is used to set up a mapping of the FPage specified by the *src\_fpage* input parameter at the address specified by the *dst\_addr* input parameter with the attributes of the mapping determined by the input parameter *attr*, in the external address space specified by the *eas* input parameter.

### C-1.2.4 unmap()

The `unmap()` method is used to unmap a region specified by the *dst\_fpage* input parameter in the external address space specified by the input parameter, *eas*.

**C-1.3 INTERFACE\_IGUANA\_THREAD\_UUID**

<b>INTERFACE_IGUANA_THREAD_UUID</b>				
<b>Method:</b>	<b>Parameters(in):</b>	<b>Parameters(out):</b>	<b>Return:</b>	<b>Section:</b>
id()	(tid)	()	thread_ref_t	C-1.3.1
l4id()	(thread_ref)	()	L4_ThreadId_t	C-1.3.2
start()	(thread, ip, sp)	()	void	C-1.3.3
delete()	(thread_ref)	()	void	C-1.3.4
set_exception()	(thread, exc, ip, sp)	()	void	C-1.3.5

**C-1.3.1 id()**

The `id()` method returns the Iguana thread reference of the thread specified by the input parameter, `tid`. The `tid` input parameter requires the caller to specify the `L4_ThreadId` of the thread.

**C-1.3.2 l4id()**

The `l4id()` method returns the `L4_ThreadId` of the thread specified by the input parameter, `thread_ref`. The `thread_ref` input parameter requires the caller to specify the Iguana thread reference of the thread.

**C-1.3.3 start()**

The `start()` method is used to activate the thread specified by the parameter `thread`. This function also requires the parameters `sp` and `ip`, which are used to specify the stack pointer and the instruction pointer of the thread to be activated.

**C-1.3.4 delete()**

The `delete()` method is used to delete the thread specified by the `thread_ref` input parameter. The `thread_ref` input parameter requires the user to specify the Iguana thread reference of the thread to be deleted.

**C-1.3.5 set\_exception()**

The `set_exception()` method is a new method which is yet to be implemented.

**C-1.4 INTERFACE\_IGUANA\_HARDWARE\_UUID**

<b>INTERFACE_IGUANA_HARDWARE_UUID</b>				
<b>Method:</b>	<b>Parameters(in):</b>	<b>Parameters(out):</b>	<b>Return:</b>	<b>Section:</b>
register_interrupt()	(hw, handler, interrupt)	()	int	C-1.4.1
back_memsection()	(hw, ms, paddr, attr)	()	int	C-1.4.2

**C-1.4.1 register\_interrupt()**

The `register_interrupt()` method registers the thread specified using the `handler` input parameter as the interrupt handler for the interrupt specified using the `interrupt` input parameter. This method returns 0 on successful completion and an error code on failure.

**C-1.4.2 back\_memsection()**

The `back_memsection()` method is used to establish a mapping between a specified memory section and physical memory. This method returns 0 on successful completion and an error code on failure.

## C-1.5 INTERFACE\_IGUANA\_MEMSECTION\_UUID

INTERFACE_IGUANA_MEMSECTION_UUID				
Method:	Parameters(in):	Parameters(out):	Return:	Section:
register_server()	(memsec, thread)	()	int	C-1.5.1
lookup()	(addr)	(server)	memsection_ref	C-1.5.2
info()	(memsec)	(size)	uintptr_t	C-1.5.3
delete()	(memsec)	()	void	C-1.5.4
set_attributes()	(memsec, attr)	()	void	C-1.5.5
map()	(memsec, offset, phys)	()	int	C-1.5.6
unmap()	(memsec, offset, size)	()	int	C-1.5.7
page_map()	(memsec, from_page, to_page)	()	int	C-1.5.8
page_unmap()	(memsec, to_page)	()	int	C-1.5.9
virt_to_phys()	(vaddr)	(size)	uintptr_t	C-1.5.10

### C-1.5.1 register\_server()

The `register_server()` method is used to register a particular thread as the server thread for a particular memory section. The server thread and the memory section are specified using the `server` and `memsec` input parameters, respectively. The `register_server()` method returns 0 on successful completion.

### C-1.5.2 lookup()

The `lookup()` method is used to acquire a memory reference to the memory section in which a particular object resides as well as the server thread associated with that memory section. The `lookup()` method returns a reference to the memory section containing the object on successful completion and returns the server associated with the memory section in the `server` output parameter.

### C-1.5.3 info()

The `info()` method returns the base address and size of the memory section specified by the input parameter, `memsec`. The size of the memory section is returned in the output parameter, `size`, and the base address of the memory section is returned as the return value.

### C-1.5.4 delete()

The `delete()` method is used to delete the memory section specified by the input parameter `memsec`.

### C-1.5.5 set\_attributes()

The `set_attributes()` method is used to set the memory attributes of the memory section specified by the input parameter, `memory`. Memory sections are further described in Part A of this manual in Section A-1.1. A list of the memory attributes available in Iguana is provided in Section A-1.1.2.

### C-1.5.6 map()

The `map()` method is used to establish a mapping between a `Phymem` object specified by the input parameter `phys`, at the offset specified by the input parameter `offset`, in the memory section specified by the input parameter `memsec`.

### C-1.5.7 unmap()

The `unmap()` method is used to unmap any mapping in the range specified by the input parameters, `offset` and `size`.

**C-1.5.8 page\_map()**

The `page_map()` method is used to map the `L4_Fpage` specified by the input parameter `from_page`, to the `L4_Fpage` specified by the input parameter `to_page`, in the memory section specified by the input parameter `memsec`.

**C-1.5.9 page\_unmap()**

The `page_unmap()` method is used to unmap the `L4_Fpage` specified by the input parameter `to_page` in the memory section specified by the input parameter `memsec`.

**C-1.5.10 virt\_to\_phys()**

The `virt_to_phys()` method is used to obtain the base physical address and the size of the `Fpage` backing a particular memory section at the specified virtual address.

## C-1.6 INTERFACE\_IGUANA\_PHYSPOOL\_UUID

INTERFACE_IGUANA_PHYSPOOL_UUID				
Method:	Parameters(in):	Parameters(out):	Return:	Section:
alloc()	(pool, size, owner)	()	cap_t	C-1.6.1
alloc_fixed()	(pool, size, base, clist, owner)	()	cap_t	C-1.6.2

### C-1.6.1 alloc()

The `alloc()` method creates a new `Phymem` object in the target protection domain, specified in the input parameter, `owner`. The `pool` input parameter is used to specify the memory pool from which the new `Phymem` object is created. The size of the new `Phymem` object is specified using the `size` input parameter.

### C-1.6.2 alloc\_fixed()

The `alloc_fixed()` method creates a new `Phymem` object in the protection domain specified by the input parameter, `owner`. The `pool` input parameter is used to specify the memory pool from which the new `Phymem` object is created. The size of the new `Phymem` object is specified using the input parameter, `size`. The base address of new `Phymem` object is specified in the `base` input parameter.

**C-1.7 INTERFACE\_IGUANA\_PHYSMEM\_UUID**

<b>INTERFACE_IGUANA_PHYSMEM_UUID</b>				
<b>Method:</b>	<b>Parameters(in):</b>	<b>Parameters(out):</b>	<b>Return:</b>	<b>Section:</b>
delete()	(pm)	()	void	C-1.7.1
info()	(pm)	(paddr, psize)	void	C-1.7.2

**C-1.7.1 delete()**

The `delete()` method deletes the Phymem object specified by the parameter `pm`.

**C-1.7.2 info()**

The `info()` method provides the base physical address of the Phymem object specified using the parameter `pm`, in the `paddr` output parameter. The `info()` method also returns the size of the Phymem object in bytes in the output parameter, `psize`.

## C-1.8 INTERFACE\_IGUANA\_CLIST\_UUID

INTERFACE_IGUANA_CLIST_UUID				
Method:	Parameters(in):	Parameters(out):	Return:	Section:
delete()	(clist)	()	void	C-1.8.1
insert()	(clist, cap)	()	int	C-1.8.2
lookup()	(clist, obj, interface)	(cap)	int	C-1.8.3
remove()	(clist, cap)	()	int	C-1.8.4

### C-1.8.1 delete()

The `delete()` method is used to delete a specified capability list.

### C-1.8.2 insert()

The `insert()` method is used to insert a particular capability into a specified capability list.

### C-1.8.3 lookup()

The `lookup()` method is used to locate a capability matching a specified object reference and interface identifier in a particular capability list.

### C-1.8.4 remove()

The `remove()` method is used to remove a single copy of specified capability from a particular capability list.

## C-2 Example Configuration File

```
<?xml version="1.0"?>
<!DOCTYPE image SYSTEM "weaver-1.0.dtd">

<image>
  <machine>
    <word_size size="4" />
    <page_size size="4K" />

    <virtual_memory name="virtual_addr">
      <region base="0x0" size="0xc0000000"/>
    </virtual_memory>

    <!-- Banks of normal memory. -->
    <physical_memory name="main_memory">
      <region base="0x800000" size="0x1000000"/>
      <region base="0x1000000" size="0x100000"/>
    </physical_memory>

    <!-- Uncached memory. -->
    <physical_memory name="more_ram">
      <region base="0x2000000" size="0x100000"/>
      <region base="0x4000000" size="0x100000"/>
    </physical_memory>

    <physical_memory name="tcm_core">
      <region base="0x80000000" size="0x100000"/>
    </physical_memory>
  </machine>

  <virtual_pool name="main_virt">
    <memory src="virtual_addr" />
  </virtual_pool>

  <physical_pool name="main_phys">
    <memory src="main_memory" />
  </physical_pool>

  <physical_pool name="somemore" cached="false">
    <memory src="more_ram" size="0x80000" />
  </physical_pool>

  <physical_pool name="evenmore">
    <memory base="0x4000000" size="0x80000" />
  </physical_pool>

```

## Example Configuration File - Continued

```
<physical_pool name="tcm">
  <memory src="tcm_core" />
</physical_pool>

<kernel file="/path/to/kernel" xip="true">
  <segment name="kip" physpool="somemore" />

  <heap size="4M" />

  <!-- KIP config. By default edits the kernel.kip section. -->
  <config>
    <option key="spaces" value="255" />
  </config>
</kernel>

<rootprogram file="/path/to/iguana_server"
  virtpool="main_virt" physpool="main_phys">
  <segment name="data" physpool="somemore" />

  <!-- Load an extension into iguana_server. -->
  <extension name="library" file="/path/to/extension" >
  </extension>
</rootprogram>

<pd name="isolated">
  <memsection name="make_dynamically" size="16K" attach="rwx" />
</pd>

<program name="demo" file="/path/to/file" physpool="default"
  priority="110" >

  <!-- Extra_data segment is placed in TCM -->
  <segment name="extra_data" physpool="tcm" attach="rwx"/>

  <patch address="__phys_addr_ram" value="0xa0000000" bytes="4"/>
  <stack size="0x4000" />
</program>

<program name="demo2" direct="true" file="/path/to/file">
  <thread name="second" start="__start_second" priority="150" >
    <stack physpool="tcm" attach="rwx"/>
  </thread>

  <memsection name="data" file="/path/to/data"
    physpool="somemore" pager="custom">
  </memsection>
</program>
```

## Example Configuration File - Continued

```
<program name="demo3" file="/path/to/file">
  <commandline>
    <arg value="demo3" />
    <arg value="Hello" />
    <arg value="World" />
  </commandline>

  <environment>
    <entry key="SOMEMEMORY" cap="data/rw" attach="rw" />
    <entry key="TALKTOME" cap="extrathread_master" />
    <entry key="TCM_POOL" cap="tcm_master" />
  </environment>
</program>
</image>
```



