

Elfweaver Reference Manual

Document Number: OK 40000:2007 (revision 10)
Software Version: 3.0
Date: October 13, 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.

Authors:

Open Kernel Labs

Contact Details:

Open Kernel Labs Pty Ltd
Attention: Open Kernel Labs

Suite 3, 540 Botany Road
Alexandria, NSW 2015
Australia

email: enquiries@ok-labs.com

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

Contents

| | |
|---------------------------------------|----------|
| Overview | 5 |
| 1 ELF File Layout | 6 |
| 1.1 ELF Header | 6 |
| 1.2 Program Header Table | 7 |
| 1.3 Section Header Table | 7 |
| 2 Introduction to Elfweaver | 8 |
| 2.1 Starting Elfweaver | 8 |
| 2.2 Merge | 8 |
| 2.2.1 <i>Options for Merging</i> | 9 |
| 2.2.2 <i>Help</i> | 9 |
| 2.2.3 <i>Output</i> | 9 |
| 2.2.4 <i>Custom Path</i> | 9 |
| 2.2.5 <i>Last Physical Address</i> | 10 |
| 2.2.6 <i>Ignore</i> | 10 |
| 2.2.7 <i>Program Header Offset</i> | 10 |
| 2.2.8 <i>Kernel Heap Size</i> | 10 |
| 2.2.9 <i>Memory Map</i> | 10 |
| 2.3 Modify | 11 |
| 2.3.1 <i>Options for Modifying</i> | 11 |
| 2.3.2 <i>Help</i> | 11 |
| 2.3.3 <i>Output</i> | 11 |
| 2.3.4 <i>Adjust</i> | 12 |
| 2.3.5 <i>Physical</i> | 12 |
| 2.3.6 <i>Physical Entry</i> | 12 |
| 2.3.7 <i>Change</i> | 13 |
| 2.3.8 <i>Merge Sections</i> | 13 |
| 2.3.9 <i>Remove NOBITS</i> | 13 |
| 2.3.10 <i>Binary</i> | 13 |
| 2.4 Print | 14 |
| 2.4.1 <i>Options for Printing</i> | 14 |
| 2.5 Memstats | 14 |
| 2.5.1 <i>Options for Memstats</i> | 15 |
| 2.5.2 <i>Help</i> | 15 |
| 2.5.3 <i>XML</i> | 15 |
| 2.5.4 <i>Verify</i> | 15 |
| 2.5.5 <i>Report</i> | 16 |
| 2.5.6 <i>Verbose</i> | 20 |
| 2.5.7 <i>Largest Number</i> | 20 |
| 2.5.8 <i>Diff</i> | 20 |
| 2.5.9 <i>Repository and Changeset</i> | 20 |
| 2.6 ElfAdorn | 21 |
| 2.6.1 <i>Options for ElfAdorn</i> | 21 |
| 2.6.2 <i>Output</i> | 21 |
| 2.6.3 <i>File Segment List</i> | 21 |
| 2.6.4 <i>Command Segment List</i> | 22 |
| 2.6.5 <i>Create Segments</i> | 22 |

| | | |
|----------|-----------------------------------|-----------|
| 3 | Configuration File | 23 |
| 3.1 | Image | 23 |
| 3.2 | Include | 24 |
| 3.3 | Machine | 27 |
| 3.3.1 | Word Size | 27 |
| 3.3.2 | Page Size | 28 |
| 3.3.3 | CPU | 28 |
| 3.3.4 | Virtual Memory | 28 |
| 3.3.5 | Physical Memory | 29 |
| 3.3.6 | Kernel Heap Attributes | 30 |
| 3.3.7 | Cache Policy | 31 |
| 3.3.8 | Physical Device | 31 |
| 3.4 | Kernel | 33 |
| 3.4.1 | Config | 34 |
| 3.4.2 | Kernel Heap | 35 |
| 3.4.3 | Patch | 36 |
| 3.4.4 | Segment | 36 |
| 3.4.5 | Use Device | 36 |
| 3.5 | OKL4 Cell | 36 |
| 3.5.1 | IRQ | 39 |
| 3.5.2 | Thread | 39 |
| 3.5.3 | Mutex | 39 |
| 3.5.4 | Use Device | 40 |
| 3.5.5 | Space | 40 |
| 3.5.6 | Environment | 41 |
| 3.5.7 | Memory Sections | 42 |
| 3.6 | Iguana Cell | 44 |
| 3.6.1 | Segment | 44 |
| 3.6.2 | Patch | 44 |
| 3.6.3 | Extension | 45 |
| 3.6.4 | Stack | 45 |
| 3.6.5 | Heap | 45 |
| 3.6.6 | Programs | 46 |
| 3.6.7 | Protection Domains | 50 |
| 3.6.8 | Memory Sections | 52 |
| 3.6.9 | Capabilities | 53 |
| 3.6.10 | Object Environment | 54 |
| 3.7 | Heap | 56 |
| 3.8 | Stack | 57 |
| 3.9 | Segments | 58 |
| 3.10 | Patch | 60 |
| 3.11 | Memory pools | 61 |
| 3.11.1 | Virtual Memory Pools | 61 |
| 3.11.2 | Physical Memory Pools | 61 |
| 4 | Customized Elements | 63 |
| 4.1 | Elfweaver Custom Path | 63 |
| 4.2 | XML Namespaces | 64 |
| 4.3 | Creating an XML Extension | 65 |
| 4.3.1 | Creating a new element | 65 |
| 4.3.2 | XML Collector | 65 |
| 4.3.3 | Registering a customisable object | 67 |
| A | Example Configuration File | 69 |

Overview

Elfweaver is a tool that enables the user to manipulate ELF files. The main functionality provided by Elfweaver is that it allows the user to *merge* multiple ELF files into a single ELF file which may then be used to create an image. Elfweaver achieves this by providing the user with a set of commands and requiring the user to supply a configuration file outlining the intended layout of the resulting ELF file. Elfweaver also allows the user to *modify* simple attributes of ELF files as well as to *print* the contents of ELF files in similar format to that used by *readelf*.

This manual begins by providing a brief overview of the contents of an ELF file, followed by an introduction to Elfweaver and an overview of its functionality. This is followed by a description of the commands and options to those commands provided by Elfweaver. This manual then provides a detailed examination of the configuration file and its constituent elements. A complete example configuration file is provided as an appendix to this manual.

1 ELF File Layout

Executable and Linkable Format(ELF) is a file format for executable files, relocatable object files, core files and shared libraries. Elfweaver is only concerned with executable files.

An ELF file contains a single ELF header which may be followed by either, neither or both, a program header table and a section header table. The following sections provide a brief overview of the main components of an ELF file.

1.1 ELF Header

Each ELF file contains a header which is always located at offset zero of the file. The ELF header describes the type of the object file, its target architecture and the version of ELF used by the ELF file.

The ELF Header also contains the location of the *program header table* and *section header table* within the ELF file as well as the number of program and section headers contained in each table. It also contains the size of each entry in both the program header table and section header table. The program header table and section header table are described in Sections 1.2 and 1.3, respectively.

Example: *ELF Header of an ELF file in a readable format*

```

ELF Header:
  Magic:   7f 45 4c 46 01 01 01 61 00 00 00 00 00 00 00 00
  Class:                               ELF32
  Data:                                   2's complement, little endian
  Version:                               1 (current)
  OS/ABI:                                 ARM
  ABI Version:                            0
  Type:                                    EXEC (Executable file)
  Machine:                                 ARM
  Version:                                 0x1
  Entry point address:                    0xf0000000
  Start of program headers:               52 (bytes into file)
  Start of section headers:               488940 (bytes into file)
  Flags:                                   0x202, has entry point, GNU EABI,
                                           software FP

  Size of this header:                    52 (bytes)
  Size of program headers:                 32 (bytes)
  Number of program headers:               13
  Size of section headers:                 40 (bytes)
  Number of section headers:               41
  Section header string table index:      40

```

1.2 Program Header Table

The program header table contains an array of entries where each entry contains a structure that describes a segment in the object file. The size of an entry and the number of entries are specified in the ELF header.

An ELF segment consists of a group of one or more sections. It is usually used as a means of grouping related, consecutive sections. For example, in the example program header table the first segment contains the sections `kernel.text`, `kernel.rodata` and `kernel.init`. This information is provided in the section to segment mapping. It should be noted that these sections appear consecutively in the ELF file as seen in the example section header table described in Section 1.3.

This example program header table shows the first two entries in the list of program headers as well as the section to segment mappings of those two entries. The program header table provides the type of the segment, in this case both segments shown are of the type `LOAD`, the offset within the ELF file as well as the virtual and physical address of the segment. The program header table also provides the file size of the segment within the ELF file as well as the size of the segment once it is loaded into memory. Lastly the program header table lists the flags of the segment, including its permissions and its alignment within the file.

Example: Program Header Table of an ELF file

Program Headers:

| Type | Offset | VirtAddr | PhysAddr | FileSiz | MemSiz | Flg | Align |
|------|----------|------------|------------|---------|---------|-----|--------|
| LOAD | 0x008000 | 0xf0000000 | 0xa0000000 | 0x1abc8 | 0x1d1b0 | RWE | 0x8000 |
| LOAD | 0x028000 | 0xf0020000 | 0xa0020000 | 0x06000 | 0x06000 | RW | 0x8000 |
| ... | | | | | | | |

Section to Segment mapping:

| Segment | Sections |
|---------|---------------------------------------|
| 00 | kernel.text kernel.rodata kernel.init |
| 01 | kernel.kspace kernel.kip kernel.traps |
| ... | |

1.3 Section Header Table

A section header table is similar to the program header table described in Section 1.2. It contains an array of structures, where each structure corresponds to a section in the ELF file. An ELF section can hold different types of information including but not limited to executable code, data and dynamic linking information.

The following example shows the first three entries of an example section header table. It lists the name of each section. The second and third section headers, `kernel.text` and `kernel.rodata` are two of the sections belonging to the first segment in the example described in Section 1.2.

Example: Section Header Table of an ELF file

Section Headers:

| [Nr] | Name | Type | Addr | Off | Size | ES | Flg | Lk | Inf | Al |
|------|---------------|----------|----------|--------|--------|----|-----|----|-----|------|
| [0] | | NULL | 00000000 | 000000 | 000000 | 00 | | 0 | 0 | 0 |
| [1] | kernel.text | PROGBITS | f0000000 | 008000 | 00eee0 | 00 | AX | 0 | 0 | 4096 |
| [3] | kernel.rodata | PROGBITS | f000eee0 | 016ee0 | 003888 | 00 | A | 0 | 0 | 8 |
| ... | | | | | | | | | | |

2 Introduction to Elfweaver

Elfweaver is a tool that may be used to manipulate ELF files. Elfweaver is written as a Python 2.3 application that only uses cross-platform libraries. Therefore Elfweaver should run on Linux, Mac-OS X and Windows.

The main functionality of Elfweaver is that it allows the user to *merge* multiple ELF files into a single ELF file using the `merge` command described in Section 2.2. The resulting ELF file may then be used to create an image. Elfweaver also provides the ancillary functions *modify*, *print*, and *elfadorn*. The `modify` command may be used to alter certain attributes of an ELF file and the `print` command allows the user to display the contents of an ELF file in a similar format to that used by *readelf*. The `elfadorn` command acts as a wrapper around the standard linker that generates the segment names used by the segment elements. The following sections provide a brief outline of Elfweaver followed by a description of the `merge`, `modify`, `print`, `elfadorn` and `memstats` commands.

2.1 Starting Elfweaver

Elfweaver provides three commands that may be used to manipulate ELF files, these being, `merge`, `modify` and `elfadorn`. Elfweaver also provides the commands `print` and `memstats`. The `print` command allows the user to display the contents of an ELF file in a similar format to *readelf* and the `memstats` command allows the user to print the memory statistics of particular ELF file. Running Elfweaver without specifying a command or any arguments results in the following usage message:

```
Elfweaver -- a tool for manipulating ELF files.

Basic commands:

elfweaver print           Print display an ELF.
elfweaver merge          Merge a set of files into one ELF.
elfweaver modify         Modify attributes of an ELF.
elfweaver memstats       Print memory statistics of an ELF.

elfweaver <cmd> -H      Obtain help on a specific command.
```

2.2 Merge

The `merge` command is used to merge multiple ELF files to produce a single ELF file, which may then be used to create an image. The `merge` command has the following usage:

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

The majority of the functionality of the `merge` command may be utilized using the `specfile` argument as opposed to the command options described in Section 2.2.1. The `specfile` argument is used specify a *configuration file* describing the intended layout of the image using well formed XML. Configuration files are further described in Chapter 3.

2.2.1 Options for Merging

The following options may be used with the `merge` command. Each option is further described in the following sections.

- `-H` or `--help`
- `-o` or `--output`
- `-c` or `--custom-path`
- `--lastphys`
- `--ignore = <regex>`
- `--program-header-offset = <offset>`
- `--kernel-heap-size = <size >`
- `--map`

2.2.2 Help

The `-H` or `--help` option is used to list the usage and options available for the `merge` command.

```
Example Usage: elfweaver merge --help
```

2.2.3 Output

The `-o` or `--output` option is used to specify the name of the file to which the merged ELF file is written. Elfweaver requires an output file to be specified. Where the user fails to specify an output file, an error will be raised by Elfweaver. The `--output` option has the following usage:

```
Example Usage: elfweaver merge config.xml --output=image.elf
```

For example, the above command can be used to merge the ELF files as specified in the configuration file, `config.xml`, and store the resulting ELF file in `image.elf`.

2.2.4 Custom Path

The `-c` or `--custom-path` option is used to specify the directory containing customized extensions. The specified directory must contain the same format as the `$ELFWEAVER_CUSTOM_PATH` environment variable. This will override the value of `$ELFWEAVER_CUSTOM_PATH`. If neither this command line option nor the environment variable is set, Elfweaver will not search for custom modules. This option has the following usage:

```
Example Usage: elfweaver merge -c path/to/custom/ext config.xml
```

Custom extensions are further described in Chapter 4.

2.2.5 Last Physical Address

The `--lastphys` option is used to print the address of the largest region of free physical memory available after Elfweaver has finished building the image. As Elfweaver tends to allocate memory from the bottom of physical memory, this option may be used to determine the amount of physical memory consumed by the image. This option has the following usage:

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

2.2.6 Ignore

The `--ignore` option is used to notify Elfweaver to ignore any element whose name matches the specified regular expression, which in the example below is `example_regex`. This option may be used in conjunction with the `--lastphys` option to gradually assemble an image.

```
Example Usage: elfweaver merge --ignore=example_regex config.xml
```

2.2.7 Program Header Offset

The `--program-header-offset` option is used to locate the ELF program headers, segment data and section headers that are, in the case of the example usage outlined below, *offset* bytes from the start of the file. The data is placed immediately after the ELF header by default. Elfweaver reports an error in the event the specified *offset* is less than the size of the ELF header for the target architecture. This is 52 bytes on 32-bit systems.

```
Example Usage: elfweaver merge --program-header-offset=8096 config.xml
```

2.2.8 Kernel Heap Size

The `--kernel-heap-size` option is used to set the size of the kernel heap to a specified value. This option may be used to override the size of the kernel heap specified in the configuration file. The `--kernel-heap-size` option has the following usage:

```
Example Usage: elfweaver merge --kernel-heap-size=size config.xml
```

2.2.9 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
```

2.3 Modify

The `modify` command is used to modify the contents of an ELF file.

```
Usage: elfweaver modify file [options]
```

2.3.1 Options for Modifying

The following options may be used with the `modify` command. Each option is further described in the following sections.

- `-H` or `--help`
- `-o` or `--output`
- `--adjust`
- `--physical`
- `--physical_entry`
- `--change`
- `--merge_sections`
- `--remove_nobits`
- `--binary`

2.3.2 Help

The `-H` or `--help` option is used to list the usage and options available for the `modify` command.

```
Example Usage: elfweaver modify --help
```

2.3.3 Output

The `-o` or `--output` option may be used to specify the name of the file to which the modified ELF file is written. If an output file is not specified, Elfweaver will over-write the input ELF file.

For example the following command can be used to modify the ELF file `test.elf` such that the virtual address of each segment is set to its physical address using the `--physical` option. By specifying the `result.elf` file to be the output file using the `--output` option, `test.elf` remains unmodified, with the modified ELF file being written to `result.elf`.

```
Example Usage: elfweaver modify test.elf --physical --output=result.elf
```

2.3.4 Adjust

The `--adjust` option is used to modify certain attributes of a specified ELF file. This option requires two additional arguments, `attribute` and `value`.

The `attribute` argument is used to identify the attribute of the ELF file to be modified. Currently the only attribute supported is `segment.paddr`. Specifying `segment.paddr` allows the physical address of all segments in the specified ELF file to be modified.

The `value` argument is used to specify either an absolute or relative offset and may be specified in hexadecimal or as an integer. For example, the following command can be used to increment the physical address of all segments in the ELF file `test.elf`, by an offset of `0x0100000`.

```
Example Usage: elfweaver modify test.elf --adjust segment.paddr +0x0100000
```

A prefix of `+` or `-` may be used to specify a relative offset where the physical address of all segments will be incremented or decremented by the specified value. Specifying a value that is not prefixed with `+` or `-` will result in the physical value of all the segments in the specified ELF file being set to the specified value. Absolute offsets are of little use in this context, and is only supported for consistency and flexibility reasons.

2.3.5 Physical

The `--physical` option is used to set the virtual addresses of the segments and sections in the ELF file to their corresponding physical addresses. This option can be used to ensure that sections contain the appropriate physical address which is necessary when using an ELF loader that loads into virtual addresses instead of physical addresses. This option does not require any additional arguments other than the ELF file being modified.

The following command may be used to change the physical addresses of the sections in each segment, for all segments in the ELF file, `test.elf`.

```
Example Usage: elfweaver modify test.elf --physical
```

Elfweaver sets the virtual address of each segment in the ELF file to its physical address. As sections only have virtual addresses, Elfweaver then sets the address of each section to the address of the section minus an offset for each section in the segment. The offset is calculated as follows:

$$\text{offset} = \text{vaddr} - \text{paddr}$$

where `vaddr` and `paddr` are the virtual and physical addresses of the segment to which the section belongs.

2.3.6 Physical Entry

The `--physical_entry` option is used to change the entry point of the ELF file from a virtual address to a physical address. This option does not require any arguments in addition to the ELF file being modified.

The following command may be used to change the entry point in the ELF file `test.elf`, to the corresponding physical address:

```
Example Usage: elfweaver modify test.elf --physical_entry
```

Elfweaver converts the entry point of `test.elf` from a virtual address to a physical address by setting the entry point to be the entry point minus an offset, where the offset is calculated as follows:

$$\text{offset} = \text{vaddr} - \text{paddr}$$

where `vaddr` and `paddr` are the virtual and physical addresses of the segment in the ELF file containing the entry point, which in this example is `test.elf`. Note that this option cannot be used on an ELF file that has already had its virtual addresses changed to physical addresses using the `--physical` command.

2.3.7 Change

The `--change` option may be used to modify a particular attribute in the specified ELF file. This option requires two additional arguments, `attribute` and `old_value=new_value`.

The `attribute` is used to identify the attribute of the ELF file to be modified. Currently, the only attribute supported is `segment.paddr`. Specifying `segment.paddr` allows the physical address of a particular segment in the specified ELF file to be modified.

The `old_value=new_value` argument is used to specify the `old_value` of the physical address of the segment to be changed to the `new_value`. All attributes which have the `old_value` will be set to the `new_value`, all other attributes will be ignored. For example, the following command may be used to change the physical address of the segment with the old physical address `0xaaa00000` in the ELF file `test.elf`, to `0xbbbb00000`.

```
elfweaver modify test.elf --change segment.paddr 0xaaa00000=0xbbbb00000
```

2.3.8 Merge Sections

The `--merge_sections` option may be used to merge the first group of consecutive sections of an ELF file that begin with a particular name. The `--merge_sections` option requires one additional argument, `name`. All sections in the specified ELF file that begin with `name` will be merged into a single new section called `name`, containing the data from each of the sections in order.

For example, the following command can be used to merge the first group of consecutive sections in the ELF file `test.elf` that start with the name `kernel`.

```
Example Usage: elfweaver modify test.elf --merge_sections kernel
```

2.3.9 Remove NOBITS

The `--remove_nobits` option may be used to replace the NOBITS `.bss` sections with equivalent zero filled PROGBITS sections. For example, the following command can be used to replace the NOBITS section in the ELF file, `test.elf` with an equivalent PROGBITS section.

```
Example Usage: elfweaver modify test.elf --remove_nobits
```

2.3.10 Binary

The `--binary` option may be used to generate a memory dump which may be used by systems without an ELF Bootloader. Note that this option may only be used in conjunction with the `-o` or `--output` option. The `--binary` option provides the same result `objcopy -O binary`.

For example, the following command can be used to generate a binary of the ELF file `test.elf`. The generated binary is written in to the `output_file`.

```
Example Usage: elfweaver modify test.elf --binary -o output_file
```

2.4 Print

The `print` command is used to display the contents of an ELF file or part thereof, in a similar format to that used by `readelf`. The `print` command has the following usage:

```
Usage: elfweaver print [options] file
```

2.4.1 Options for Printing

The following options can be used with the `print` command. The `print` command requires that at least one option is specified. Failing to specify an option results in nothing being printed.

- `-H` or `--help` is used to list the usage and options available for the `print` command.
- `-a` or `--all` is used to print all information.
- `-h` or `--header` is used to print the header of the ELF file.
- `-l` or `--phaders` is used to print the program header table of the ELF file.
- `-S` or `--shaders` is used to print the section header table of the ELF file.
- `-k` or `--kconfig` is used to print the kernel configuration and initialization information.
- `-B` or `--bootinfo` is used to print the contents and size of the OKL4 bootinfo data structure of the ELF file.
- `-m` or `--segnames` is used to print the segment names of the segments in the ELF file.
- `-s` or `--syms` is used to print the symbol table of the ELF file.
- `-r` or `--relocs` is used to print all relocation entries in the ELF file.
- `-W` or `--wide` is used to allow output to exceed the width of 80 characters.
- `-e` or `--elfweaverinfo` is used to print the Elfweaver information section of the kernel ELF file.

2.5 Memstats

The `memstats` command is used to obtain memory usage statistics of an `image.elf` file generated using the `merge` command described in Section 2.2. This command is not supported for files generated using the `modify` command described in Section 2.3 and all other ELF files.

The `memstats` command has the following usage:

```
Usage: elfweaver memstats [options] file [file]
```

2.5.1 Options for Memstats

- **-H** or **--help**
- **-x** or **--xml**
- **-y** or **--verify**
- **-r NAME** or **--report=NAME**
- **-v** or **--verbose**
- **-l N_OBJS** or **--largest-num=N_OBJS**
- **-d** or **--diff**
- **-R REPOSITORY** or **--repository=REPOSITORY**
- **-C CHANGESET** or **--changeset=CHANGESET**

2.5.2 Help

The **-H** or **--help** option is used to list the usage and options available for the `memstats` command.

```
Example Usage: elfweaver memstats --help
```

2.5.3 XML

The **-x** or **--xml** option produces memory usage statistics in XML format. It should be noted that this option requires the file specified by the `file` argument to be an ELF image. The output XML file called `memstats.xml`, will be located in the same directory as the specified ELF image.

```
Example Usage: elfweaver memstats --xml image.elf
```

2.5.4 Verify

The **-y** or **--verify** option is used to verify whether a particular XML file is legal. This command may be used in conjunction with the **--xml** command. Note that this option requires the program `xmllint` to be installed.

```
Example Usage: elfweaver memstats -y --xml image.elf
```

2.5.5 Report

The `-r NAME` or `--report=NAME` option produces a memory usage statistics report of the report type specified by `NAME` in a plain text format. This option requires a memory usage statistics file in an XML format as generated by the `-x` or `--xml` command described in Section 2.5.3.

If used in conjunction with the `-x` or `--xml` command, the report is generated from the statistics generated by that command. Otherwise the file argument must specify a memory usage statistics XML file.

Elfweaver accepts the following values for `NAME`:

- `initial`
- `fixed`
- `ids`
- `heap`
- `env-summary`

Each report type is further described below.

2.5.5.1 Initial Memory Usage Report

The initial memory usage report is used to determine the amount of physical memory used by the image by the time the first user code is called.

Example Initial Memory Usage Report

Memory in Use at end of initialization

=====

Memory claimed by ELF image:

```
ELF Segment 1: 0xa0000000 - 0xa0026bdf ( 154KB)
ELF Segment 2: 0xa0028000 - 0xa002dfff ( 24KB)
ELF Segment 3: 0xa002e000 - 0xa00310d6 ( 12KB)
ELF Segment 4: 0xa00270d8 - 0xa00271e3 ( 0KB)
ELF Segment 5: 0xa0040000 - 0xa0082afb ( 266KB)
ELF Segment 6: 0xa008c000 - 0xa00adfab ( 135KB)
ELF Segment 7: 0xa0032000 - 0xa00324d0 ( 1KB)
ELF Segment 8: 0xa0033000 - 0xa0033653 ( 1KB)
ELF Segment 9: 0xa0034000 - 0xa0034fff ( 4KB)
```

9 Segments giving 600KB

SAS1:

```
thread_handles          0xa0500000 - 0xa0503fef ( 15KB)
                        0xa0027000 - 0xa0027fff ( 4KB)
heap_exhaustion.main/stack 0xa0700000 - 0xa0700fff ( 4KB)
heap_exhaustion.heap      0xa0600000 - 0xa06ffffff (1024KB)
ktest.main/stack         0xa0900000 - 0xa0900fff ( 4KB)
ktest.serial_mem0        0x40100000 - 0x40100fff ( 4KB)
ktest.timer_mem0         0x40a00000 - 0x40a00fff ( 4KB)
ktest.heap                0xa0800000 - 0xa08ffffff (1024KB)
```

8 additional sections giving 2083KB

Cell 1:

Heap 0xa0100000 - 0xa02ffffff (2048KB)

Cell 2:

Heap 0xa0300000 - 0xa04ffffff (2048KB)

Total Memory claimed for use: 6779KB

Physical memory remaining: 50565KB of 57344KB

2.5.5.2 Fixed Memory Usage Report

The fixed memory usage report is used to determine the amount of physical memory used by ELF image.

Example Fixed Memory Usage Report

Statistics for program kernel:

Sections (with debug and empty sections dropped):

| | |
|-----------------------|-------------|
| kernel.text | 82336 Bytes |
| kernel.rodata | 33714 Bytes |
| kernel.init | 6316 Bytes |
| kernel.roinit | 4 Bytes |
| kernel.elfweaver_info | 52 Bytes |
| kernel.data | 5396 Bytes |
| kernel.got | 4 Bytes |
| kernel.got.plt | 12 Bytes |
| kernel.bss | 12120 Bytes |
| kernel.kspace | 16384 Bytes |
| kernel.traps | 4096 Bytes |
| kernel.utcb_page | 4096 Bytes |
| initscript | 4096 Bytes |

Total static size by sections (unrounded) is 164.67 KB

Size by rounded segments (3 total) is 172 KB (42 pages)

Statistics for program heap_exhaustion:

Sections (with debug and empty sections dropped):

| | |
|----------------------------------|-------------|
| heap_exhaustion.text | 11012 Bytes |
| heap_exhaustion.cell_environment | 1233 Bytes |

Total static size by sections (unrounded) is 11.96 KB

Size by rounded segments (2 total) is 12 KB (3 pages)

Statistics for program ktest:

Sections (with debug and empty sections dropped):

| | |
|------------------------|--------------|
| ktest.text | 236836 Bytes |
| ktest.data | 53772 Bytes |
| ktest.cell_environment | 1620 Bytes |

Total static size by sections (unrounded) is 285.38 KB

Size by rounded segments (3 total) is 320 KB (80 pages)

2.5.5.3 Kernel Identifier Usage Report

The kernel identifier usage report provides a list of the types of kernel identifier together with the total number available, total number allocated and total number used.

Example Kernel Identifier Usage Report

```
ID usage
=====

Type spaceid          245 available, 245 allocated to spaces, 2 used
Type clistid          257 available, 257 allocated to spaces, 2 used
Type mutexid          256 available, 256 allocated to spaces, 0 used
Type thread_handles  4092 available, 4092 allocated to spaces, 2 used
Type capabilities0    1024 available, 1024 allocated to spaces, 1 used
Type capabilities1    1024 available, 1024 allocated to spaces, 1 used
```

2.5.5.4 Heap Usage Report

This report provides a description of the memory allocation from the heap.

Example Heap Usage Report

```
Heap Usage
=====

Cell 1:
Heap is Heap0 - size 2048KB with 38KB used

Cell 2:
Heap is Heap1 - size 2048KB with 42KB used
```

2.5.5.5 Environment Summary Usage Report

The environment summary usage report is not a standard report and is only provided for debugging purposes.

Example Environment Summary Usage Report

Quick summary of environment:

```
In kernel:
  Space called kspace
  Id Ranges:
  Pools:
    vm0 on kspc
  Programs:
    kernel with 15 sections

In Cell ctest:
  Space called ctest
  Id Ranges:
    4092 for threads
    256 for spaces
    256 for clists
    256 for mutexes
  Pools:
    vm1 on vm_pool0
  Programs:
    ctest with 6 sections
```

2.5.6 Verbose

The `-v` or `--verbose` option allows the user to generate the verbose version of the generated report.

Example Usage: `elfweaver memstats -v --report=ids --xml image.elf`

2.5.7 Largest Number

The `-l N_OBJS` or `--largest-num=N_OBJS` option allows the user to limit the output from the `memstats` command to the largest `N_OBJS` text and data symbols. This option may be used when generating reports in Section 2.5.5. The generated report will only print the `N_OBJS` largest symbols of each type. Note that this option can only be used in conjunction with generating a fixed report.

2.5.8 Diff

The `-d` or `--diff` option is used to generate a plain text report of a comparison of memory usage statistics of particular type of report on two files. Both files must be memory usage statistics files. The differences of the second file compared against the first file are calculated with the results printed in a plain text report.

2.5.9 Repository and Changeset

The `-r` or `--repository` option and the `-c` or `--changeset` option are used allow the user to track the version control repository and changeset information of the source tree, respectively. Using these command results in the supplied repository and/or changeset information being included in the generated XML file.

2.6 ElfAdorn

The `elfadorn` command acts as a wrapper around the standard linker which generates the segment names used by the segment elements. Elf files with segment names are self-contained, that is, the names are stored within the ELF file in a common section called `.segment_names`. It should be noted that segment names only apply to loadable segments.

The `elfadorn` command invokes the supplied linker and determines the appropriate segment names depending on the supplied options. These options are further described in Section 2.6.1. Where the user fails to specify segment names using either the `--file-segment-list` or `--cmd-segment-list` option, Elfweaver will then search for a linker script within the linker options. Failing this, the segments will be named according to their ELF flags. That is, *read/write* segments will be called `rw` and *read/execute* segments will be called `rx`.

The `elfadorn` command can also create segments for each *orphan* section in the input files. An orphan section is a section that is not mentioned in the linker script. This functionality is used to allow variables in source code that are placed in separate sections to be addressable by `elfweaver` in the merge command. For example, this feature may be used to reposition the variables from normal memory into tightly coupled memory. It should be noted that this functionality is only supported for the *GNU ld* linker. The `-s` or `--create-segments` flag must be supplied to `elfadorn` to invoke this feature.

Linkers currently supported by the `elfadorn` command are *GNU ld* (with the `-T` option) and the *ARM RVCT* and *ADS* linkers, both with the `--scatter` option. The `elfadorn` command has the following usage:

```
Usage: elfadorn -o image [options] -- <linker> [linker_options]
```

2.6.1 Options for ElfAdorn

The following options may be used with the `elfadorn` command. Each option is further described in the following sections.

- `-o` or `--output`
- `-f` or `--file-segment-list`
- `-c` or `--cmd-segment-list`
- `-s` or `--create-segments`

2.6.2 Output

The *output* option, `-o` or `--output`, is used to specify the name of the ELF file to be created by the linker. Currently, this option must be specified when using the `elfadorn` command, unless a `-o` option is given in the linker options. Failure to specify the *output* option will result in an error raised by Elfadorn.

```
Usage: elfadorn -o image --linker
```

2.6.3 File Segment List

The *file segment list* option, `-f` or `--file-segment-list`, is used to specify a path to a file containing the names of the segments. Each name is stored on a single line. The names are assigned to the segments in order of appearance.

```
Usage: elfadorn -o image --file-segment-list=path/to/file --linker
```

2.6.4 Command Segment List

The *command segment list* option, **-c** or **--cmd-segment-list**, is used to specify a comma separated list of names for segments. These names will be assigned to the segments in order of appearance.

```
Usage: elfadorn -o image --cmd-segment-list=seg_a,seg_b --linker
```

2.6.5 Create Segments

The *create segments* option, **-s** or **--create-segments**, is used to place orphan sections in segments with the same name as the segment.

```
Usage: elfadorn -o image -s --linker
```

3 Configuration File

The *configuration file* is supplied as an argument to the `merge` command and is used to specify the intended configuration or layout of the resulting ELF file. The `merge` command is used to merge multiple ELF files to create a single ELF file and is further described in Section 2.2.

The configuration file is specified in well formed XML and uses XML elements to specify the various aspects of the intended layout of the image. All configuration files must begin with the following lines:

Example: Beginning of the Configuration File

```
<?xml version="1.0"?>
<!DOCTYPE image SYSTEM "weaver-2.0.dtd">
```

The user is free to use XML comments, `<!-- -->`, within the configuration file at any location permitted by XML grammar. These comments will be ignored by Elfweaver.

Each of the XML elements used by Elfweaver, together with their sub-elements and attributes are further described in the following sections of this chapter. Though the elements supplied by Elfweaver cover most of layout of the configuration file, they may not adequately cover the description of architecture specific behavior. Thus, Elfweaver allows the user to add customized elements to the configuration file to better describe architecture specific aspects of the resulting image. Customized elements are further described in Chapter 4. A complete example configuration file is provided in Appendix A of the manual.

3.1 Image

The `image` element is the root element of the configuration file and contains all sub-elements used by Elfweaver. Each configuration file must specify exactly one `image` element. The `image` element does not contain any attributes.

Image Element Attributes

| Attribute: | Type: | Description: |
|------------|-------|--------------|
|------------|-------|--------------|

This element does not contain any attributes.

Example: Image Element

```
<?xml version="1.0"?>
<!DOCTYPE image SYSTEM "weaver-2.0.dtd">

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

3.2 Include

| Include Element Attributes | | |
|---|-----------------|---|
| Attribute: | Type: | Description: |
| <code>file</code> | <i>required</i> | The path name of the file to be included. |
| Example: <i>Include Element</i> | | |
| <pre><include file="/path/to/file" /></pre> | | |

The `include` element is used to include the contents of a separate file within the image and contains a single *required* attribute, `file`.

When an `include` element is encountered by Elfweaver, it is replaced with the contents of the file specified by the `file` attribute. It should be noted that the specified file must contain whole elements and may contain one or more elements at the top level. An included file may contain other `include` elements.

An example of using the `include` element consisting of a configuration file containing an `include` element, the included file and the resulting file is provided below.

| |
|---|
| <p>Example: <i>Specification file including the include element.</i></p> <pre><?xml version="1.0"?> <!DOCTYPE image SYSTEM "weaver-2.0.dtd"> <image> <include file="machine_and_pools.xml" /> <kernel file="l4kernel"> </kernel> <rootprogram file="ig_server" virtpool="virtual" physpool="physical"> </rootprogram> <program name="ig_naming" file="ig_naming"> </program> </image></pre> |
|---|

Example: *The file specified by the `include` element.*

```
<machine>
  <word_size size="0x20" />
  <page_size size="0x1000" />

  <virtual_memory name="virtual">
    <region base="0x1000" size="0xcffff000" />
  </virtual_memory>

  <physical_memory name="physical">
    <region base="0xa0000000" size="0x3800000" />
  </physical_memory>
</machine>

<virtual_pool name="virtual">
  <memory base="0x1000" size="0xcffff000"/>
</virtual_pool>

<physical_pool name="physical">
  <memory base="0xa0000000" size="0x3800000"/>
</physical_pool>
```

Example: *The resulting file.*

```
<?xml version="1.0"?>
<!DOCTYPE image SYSTEM "weaver-2.0.dtd">
<image>
<machine>
  <word_size size="0x20" />
  <page_size size="0x1000" />

  <virtual_memory name="virtual">
    <region base="0x1000" size="0xcffff000" />
  </virtual_memory>

  <physical_memory name="physical">
    <region base="0xa0000000" size="0x3800000" />
  </physical_memory>
</machine>

<virtual_pool name="virtual">
  <memory base="0x1000" size="0xcffff000"/>
</virtual_pool>

<physical_pool name="physical">
  <memory base="0xa0000000" size="0x3800000"/>
</physical_pool>

  <kernel file="l4kernel">
  </kernel>

  <rootprogram file="ig_server" virtpool="virtual"
    physpool="physical">
  </rootprogram>

  <program name="ig_naming" file="ig_naming">
  </program>
</image>
```

3.3 Machine

Machine Element Attributes

| Attribute: | Type: | Description: |
|------------|-----------------|---|
| file | <i>optional</i> | The path or location of the machine element |

Example: Machine Element

```
<machine>
  <word_size size="32" />
  <page_size size="4K" />
  <cpu name="arm1136" />

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

  <physical_memory name="main_memory">
    <region base="0x800000" size="0x1000000"/>
  </physical_memory>

  <phys_device name="timer_dev">
    <physical_memory name="timer_mem0">
      <region base="0x51000000" size="0x1000"/>
    </physical_memory>
    <interrupt name="int_timer1" number="11"/>
    <interrupt name="int_timer2" number="12"/>
  </phys_device>

  <kernel_heap_attrs distance="16M" />
  <cache_policy name="user1" value ="50">
</machine>
```

The `machine` element is used to describe the target platform and does not contain any required attributes. The `machine` element may contain one or more of instances of the sub-elements `word_size`, `page_size`, `cpu`, `virtual_memory`, `physical_memory`, `kernel_heap_attrs`, `cache_policy`, and `phys_device`, each of which is further described below.

3.3.1 Word Size

Word Size Element Attributes

| Attribute: | Type: | Description: |
|------------|-----------------|-------------------------------|
| size | <i>required</i> | Word size of machine in bits. |

Example: Word Size Element

```
<word_size size="32" />
```

The `word_size` element is used to specify the word size of the machine. It contains a single attribute `size`

which is used to specify the word size of the machine in bits. The above example machine tag specifies a machine with a word size of 32 bits.

3.3.2 Page Size

| Page Size Element Attributes | | |
|------------------------------|-----------------|-------------------------------|
| Attribute: | Type: | Description: |
| <code>size</code> | <i>required</i> | Supported page size in bytes. |

Example: Page Size Element

```
<page_size size="4K" />
```

The `page_size` element is used to specify the page sizes supported by the architecture. The `page_size` element contains a single attribute, `size`, which is used to specify a supported page size in bytes. Where multiple page sizes are supported by the architecture, multiple `page_size` elements should be declared within the `machine` tag, where each denotes a supported page size. Page sizes may be declared in decimal, as illustrated in the above example `machine` tag, or in hexadecimal using the prefix `0x`.

3.3.3 CPU

| CPU Element Attributes | | |
|------------------------|-----------------|-----------------------------|
| Attribute: | Type: | Description: |
| <code>name</code> | <i>required</i> | CPU of the target hardware. |

Example: CPU Element

```
<cpu name="arm1136" />
```

The `cpu` element is used to specify the CPU of the target hardware. The `cpu` element contains a single attribute, `name`, which is used to specify the CPU of the target hardware. An error will be raised by Elfweaver if it fails to recognize the specified CPU. The `cpu` element does not contain any sub-elements.

3.3.4 Virtual Memory

Virtual Memory Element Attributes

| Attribute: | Type: | Description: |
|------------|-----------------|-------------------------------|
| name | <i>required</i> | Name of the region of memory. |

Example: *Virtual Memory Element*

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

The `virtual_memory` element is used to describe the region of virtual memory available in the system. The `virtual_memory` element contains the single attribute `name`, which is used to specify a name for the specified region of memory as string, allowing easy reference to the same region of memory throughout the XML file.

The `virtual_memory` element may contain one or more `region` elements, each describing a contiguous range of virtual memory. The `region` element contains the attributes `base` and `size`, used to specify the starting address of the virtual memory range and its size in bytes. Both the `base` and `size` attributes may be specified as a decimal value or in hexadecimal. A suffix of **K**, **M** or **G** may be used specify the value of the `size` attribute in Kilo bytes, Mega bytes or Giga bytes, respectively. These attributes should be set to the default values specified in the configuration file supplied by the vendor.

The above example virtual memory tag describes a region of virtual memory which will be referred to throughout the remainder of the XML file as `virtual_addrs` and describes a single region of virtual memory of size `0c0000000`, commencing at virtual address `0x0`.

It should be noted that the same region of virtual memory cannot be described by multiple `region` elements. This will result in Elfweaver reporting an error.

3.3.5 Physical Memory

Physical Memory Element Attributes

| Attribute: | Type: | Description: |
|------------|-----------------|-------------------------------|
| name | <i>required</i> | Name of the region of memory. |

Example: *Physical Memory Element*

```
<physical_memory name="main_memory">
  <region base="0x800000" size="0x1000000" rights="all" />
</physical_memory>
```

The `physical_memory` element is used to describe the regions of physical memory available in the system. The `physical_memory` element contains a single attribute, `name`, which is used to specify a name for the specified region of memory as a string, allowing easy reference to the same grouping of memory throughout the remainder of the XML file.

The `physical_memory` element may contain one or more `region` elements, each describing a contiguous range of physical memory. The `region` element contains the attributes `base`, `size`, `rights` and `cache_policy`. The `base` and `size` attributes are used to specify the starting address of the physical memory range and its size in bytes. Both the `base` and `size` attributes may be specified as a decimal value or in hexadecimal. A suffix of **K**, **M** or **G** may be used specify the value of the `size` attribute in Kilo bytes, Mega bytes or Giga bytes, respectively. These attributes should be set to the default values specified in the configuration file supplied by the vendor. The `cache_policy` element is used to specify the preferred cache policy for any memory segment in the region. Memory segments that do not work with the default cache policy should set this attribute.

The `rights` attribute is used to specify a description of the cache policies permitted in the region.

The following values of the `rights` attributed are provided by Elfweaver:

all All cache policies are permitted.

The above example physical memory tag describes a region of physical memory which will be referred to throughout the remainder of the XML file as `main_memory` and describes a single region of physical memory of size `0x1000000`, commencing at the physical address `0x800000`.

It should be noted that the same region of physical memory cannot be described by multiple `region` elements. This will result in Elfweaver reporting an error.

3.3.6 Kernel Heap Attributes

| Kernel Heap Attributes Element Attributes | | |
|---|-----------------|--|
| Attribute: | Type: | Description: |
| <code>align</code> | <i>optional</i> | The alignment of the kernel heap. |
| <code>distance</code> | <i>required</i> | Maximum distance between the heap and the kernel data segment. |
| Example: <i>Kernel Heap Attributes Element</i> | | |
| <pre><kernel_heap_attrs distance="16M" /></pre> | | |

The `kernel_heap_attrs` element is used to specify the alignment and the distance the kernel heap is from the kernel data segment. The `kernel_heap_attrs` element contains a single *required* attribute `distance`, and a single *optional* attribute, `align`.

The `align` attribute is used to specify the alignment of the kernel heap. As most architectures impose alignment restrictions, Elfweaver aligns the heap to the largest page size that is smaller than the size of the heap. The `align` attribute gives the user the option of specifying the alignment of the kernel heap of less restrictive architectures to reduce memory wastage. Note that, if set, the value of the `align` attribute of the `heap` element takes priority over the `align` attribute of the `kernel_heap_attrs` element. The `heap` element is further described in Section 3.4.2.

The `distance` attribute is used to specify the maximum distance at which the heap may be located from the kernel data segment in bytes. This is useful for architectures that require the kernel heap to be located within a specified distance from the kernel data segment in physical memory. For example, the ARM architecture requires that the heap is placed within 16MB from the kernel data segment. In the event that the `distance` attribute is specified and the kernel heap is placed outside the required distance, an error will be raised by Elfweaver.

Note that a suffix of **K**, **M** or **G** may be used specify the value of the `align` and `distance` attributes in Kilo bytes, Mega bytes or Giga bytes, respectively.

3.3.7 Cache Policy

| Cache Policy Element Attributes | | |
|--|-----------------|----------------------------------|
| Attribute: | Type: | Description: |
| name | <i>required</i> | The name of the policy. |
| value | <i>required</i> | The numeric value of the policy. |
| Example: <i>Cache Policy Element</i> | | |
| <pre><cache_policy name="user1" value ="50"></pre> | | |

The `cache_policy` element is used to specify a customized caching policy. This element has the *required* attributes, `name` and `value`. The `name` attribute is used to specify the name of the caching policy. This name is then used to specify this caching policy in the `memsection` and `segment` elements as required. The `memsection` and `segment` elements are further described in Sections 3.6.8 and 3.9, respectively. Note that `memsection` elements are a subelement of the `iguana` element.

The `value` attribute is used to specify the numeric value of the caching policy. It should be noted that in the event the value specified is not supported by OKL4, its behaviour is undefined.

The standard cache policies provided by OKL4 are shown in Table 3.1. Note that the actual policy values are architecture specific.

| Policy | Description |
|----------------|--|
| default | Architecture-specific default policy |
| cached | Architecture-specific |
| uncached | Caching disabled |
| writeback | Write-back caching policy |
| writethrough | Write-through caching policy |
| coherent | A coherent caching policy |
| device | Architecture-specific policy for device memory |
| writecombining | Uncached write-combining policy |

Table 3.1: *Supported Caching Policies*

3.3.8 Physical Device

Physical Device Element Attributes

| Attribute: | Type: | Description: |
|------------|-----------------|------------------------------|
| name | <i>required</i> | Name of the physical device. |

Example: *Physical Device Element*

```
<phys_device name="timer_dev">
  <physical_memory name="timer_mem0">
    <region base="0x51000000" size="0x1000"/>
  </physical_memory>
  <interrupt name="int_timer1" number="11"/>
  <interrupt name="int_timer2" number="12"/>
</phys_device>
```

The `phys_device` element is used to describe the resources associated with a particular physical device.

A capability that references the physical devices will be automatically generated for the corresponding device server. The `phys_device` element has a single required attribute, `name`, which is used to specify the name of the physical device as a string. This name should be directly derived from the device class to which the physical device belongs. For example, the above example describes a physical device that belongs to the timer device class.

The `phys_device` element may contain one or more instances of the elements `physical_memory` and `interrupt` used to describe the resources physical memory and interrupts, associated with the physical device described by the `phys_device` element. The `physical_memory` element is further described above in Section 3.3.5.

The `interrupt` element is used to describe the interrupts raised by the physical device described by the `phys_dev` element. Elfweaver will request the Iguana server to register the specified interrupt for the main thread of the owning OKL4 program. The Iguana server will register these interrupts during initialization. The `interrupt` element contains the required attributes, `name` and `number`. These attributes are used to specify the name and corresponding interrupt number of the interrupt. The interrupt numbers are granted to the owning OKL4 program and are most important for devices with multiple interrupts as they share the same interrupt handler implemented in the device server.

3.4 Kernel

| Kernel Element Attributes | | |
|----------------------------------|-----------------|---|
| Attribute: | Type: | Description: |
| file | <i>required</i> | Path or location of the kernel ELF file. |
| configuration | <i>optional</i> | Kernel configuration to construct. |
| kernel | <i>optional</i> | Path or location of the kernel.o file. |
| libs | <i>optional</i> | List of library filenames to be linked with kernel.o. |
| linker | <i>optional</i> | Path or location of the linker wrapper script. |
| platform | <i>optional</i> | Name of SoC platform linked to the kernel. |
| sdk | <i>optional</i> | Path or location of the top of the SoC tree. |
| soc | <i>optional</i> | Path or location of the soc.o file. |
| virtpool | <i>required</i> | Default virtual memory pool. |
| physpool | <i>required</i> | Default physical memory pool. |

Example: Kernel Element

```

<kernel file="/path/to/kernel">
  <segment name="RX" pool="ROM" />

  <heap size="0x400000" />

  <config>
    <option key="threads" value="256" />
  </config>

</kernel>

```

The `kernel` element is used to describe the properties of the OKL4 kernel. The `kernel` element has two special properties. The entry point of the kernel file is used as the entry point of the ELF file generated by Elfweaver. Additionally, unless explicitly overridden, the kernel is loaded at the bottom of the memory pool allocated to it. This may be indirectly overridden by specifying the physical address of the location of each segment using the `physaddr` attribute. Segments are further described in Section 3.9.

The `virtpool` attribute is used to specify the default virtual memory pool that will be used for the allocation of virtual memory by the kernel and cells. The `physpool` attribute is used to specify the default physical memory pool that will be used for the allocation of physical memory by the kernel and cells. If either the `virtpool` or `physpool` attributes are left unspecified, Elfweaver will raise an error.

The `sdk`, `configuration` and `platform` attributes are used to link from an SDK directory. As Elfweaver is aware of the SDK directory structure, it can link from the kernel files within the tree. The `kernel` element has the attributes `sdk`, `configuration` and `platform` which are used specify the path to the top of the SoC directory, the kernel configuration to be constructed and the SoC module to be linked to the kernel, respectively. An example of the `kernel` element used in this format is provided below.

| |
|--|
| <p>Example: Linking from an SDK directory.</p> <pre> <kernel sdk="/path/to/sdk_dir" configuration="production" platform="imx31"> </kernel> </pre> |
|--|

Elfweaver currently supports the following configurations:

- production
- benchmark
- trace
- trace_debug
- kdb

The `kernel`, `soc` and `linker` attributes are used to link from separate files. Where the required attributes are specified in directories different to that of the SDK, the individual parts required to link the kernel may be specified as attributes of the `kernel` element to provide greater control over the link process. The `kernel` element has the attributes `kernel`, `soc` and `linker` which are used to specify the path of the `kernel.o` file, the path of the `soc.o` file and the path of the linker wrapper script, respectively. When used in conjunction with the SDK linking attributes, these override the default paths calculated from the SDK. An example of the `kernel` element used in this format is provided below.

Example: *Linking from separate files.*

```
<kernel kernel="/path/to/kernel.o" soc="/path/to/soc.o"
      linker="/path/to/linker.sh">
</kernel>
```

The `kernel` element may contain one or more of the sub-elements `config`, `heap`, `patch`, `segment` and `use_device`, each of which is further described in Sections 3.4.1 to 3.4.5, below.

3.4.1 Config

Config Element Attributes

| Attribute: | Type: | Description: |
|--|-------|--------------|
| <i>This element does not contain any attributes.</i> | | |
| Example: <i>Config Element</i> | | |
| <pre><config> <option key="threads" value="256" /> </config></pre> | | |

The `config` element is used specify various kernel parameters.

The `config` element contains one or more `option` elements, each of which is used to set an individual option.

3.4.1.1 Option

| Option Element Attributes | | |
|---|-----------------|----------------------|
| Attribute: | Type: | Description: |
| key | <i>required</i> | Name of the option. |
| value | <i>required</i> | Value of the option. |
| Example: <i>Option Element</i> | | |
| <pre><option key="threads" value="255" /></pre> | | |

Each option element contains the attributes `key` and `value`. The `key` attribute is used to specify the name of the option with `value` attribute being used to specify the value of the option. The supported options are:

threads The maximum number of threads that the kernel supports. By default 1024 threads are supported.

3.4.2 Kernel Heap

| Kernel Heap Element Attributes | | |
|--|-----------------|-----------------------------------|
| Attribute: | Type: | Description: |
| align | <i>optional</i> | The alignment of the kernel heap. |
| size | <i>optional</i> | Size of the kernel heap in bytes. |
| phys_addr | <i>optional</i> | Base address of the kernel heap. |
| Example: <i>Kernel Heap Element</i> | | |
| <pre><heap size="0x400000" /></pre> | | |

The `heap` element is used to specify the size and load location of the kernel heap. Where the `kernel` tag fails to specify a `heap` element, Elfweaver will allocate a heap of size equivalent to the sum of the heap sizes of all the cells in the system.

The `heap` element contains the *optional* attributes `align`, `size` and `phys_addr`. The `align` attribute is used to specify the alignment of the kernel heap. As most architectures impose alignment restrictions, Elfweaver aligns the heap to the largest page size that is smaller than the size of the heap. The `align` attribute gives the user the option of specifying the alignment of the kernel heap of less restrictive architectures to reduce memory wastage. Note that, if set, the value of the `align` attribute of the `heap` element takes priority over the `align` attribute of the `kernel_heap_attrs` element. The `kernel_heap_attrs` element is further described in Section 3.3.6.

The `size` attribute is used to specify the size of the kernel heap in bytes. The suffixes `K`, `M` or `G` maybe used when specifying both the `align` and `heap` attributes. If left unspecified, Elfweaver will specify a heap of size equivalent to the sum of the following:

- The value of the `kernel_heap` attribute of each cell.
- The size of the thread handle array. This is an array of *threads* pointers, where *threads* is the maximum number of threads as declared by the `thread` kernel configuration option.

The `phys_addr` attribute is used to specify the base address of the kernel heap. The kernel heap must be aligned on a natural boundary and placed close to the beginning of the kernel. For ARM, this is within 64MB of the kernel. Where this attribute is left unspecified, the address will be chosen by Elfweaver.

3.4.3 Patch

The `patch` element is used to modify the content of the ELF file prior to building the final image. The `patch` element is further described in Section 3.10.

3.4.4 Segment

The `segment` element describes the load location of an ELF segment. Segments are further described in Section 3.9.

3.4.5 Use Device

| Use Device Element Attributes | | |
|---|-----------------|---------------------|
| Attribute: | Type: | Description: |
| <code>name</code> | <i>required</i> | Name of the device. |
| Example: <i>Use Device Element</i> | | |
| <pre><use_device name="serial_dev"/> <use_device name="interrupt_dev"/></pre> | | |

The `use_device` element is used to specify the name of the device. This name should refer to the name attribute of the corresponding `phys_device` sub-element of the `machine` element. After gathering a list of kernel devices and their regions, Elfweaver writes out corresponding mappings to the page table, as required. Elfweaver will also patch the virtual address of the driver's region into the kernel's variable ready for use. The `phys_device` sub-element is further described in Section 3.3.8.

The `kernel` element may contain multiple `use_device` sub-elements. Note that this element, when declared within the `kernel` element is only used for OKL4 Nano and has no effect if the the OKL4 microkernel is used. A similar effect can be achieved for the OKL4 microkernel by specifying the `use_device` element within the `okl4` element.

3.5 OKL4 Cell

Cell Element Attributes

| Attribute: | Type: | Description: |
|------------------|-----------------|---|
| name | <i>required</i> | Name of the cell instance. |
| file | <i>required</i> | Filename of the initial program (must be an ELF file). |
| kernel_heap | <i>required</i> | Size of the kernel heap of the cell in bytes |
| caps | <i>optional</i> | Number of cap slots per clist. |
| clists | <i>optional</i> | Maximum number of clists that can be created by the cell. |
| direct | <i>optional</i> | A boolean value. |
| mutexes | <i>optional</i> | Maximum number of mutexes that can be created/accessed by the cell. |
| priority | <i>optional</i> | Priority of the main thread, default is 255. |
| platform_control | <i>optional</i> | A boolean value |
| max_priority | <i>optional</i> | Maximum priority of threads. |
| spaces | <i>optional</i> | Maximum number of spaces that can be created/accessed by the cell. |
| threads | <i>optional</i> | Maximum number of threads that can be created/accessed by the cell. |
| physpool | <i>optional</i> | Name of the default physical pool. |
| virtpool | <i>optional</i> | Name of the default virtual pool. |

Example: Cell Element

```

<okl4 priority="100" clists="256" file="cell_init" kernel_heap="0x200000"
  mutexes="256" name="ktest" spaces="256" platform_control="True">
  <mutex name="cell_mutex"/>

  <environment>
    <entry key="test_key" value="42"/>
    <entry key="my_mutex" cap="test_space/test_space_mutex1" />
    <entry key="my_memory" cap="normal" />
    ...
  </environment>

  <memsection cache_policy="uncached" name="serial_vaddr"
    phys_addr="0x40100000" size="0x1000" />
  <memsection mem_type="unmapped" name="ktest_segment" size="0x4000" />
  <memsection name="normal" size="0x4000" />

  <space name="test_space">
    <mutex name="test_space_mutex1" />
    <memsection name="normal2" size="0x4000" />
  </space>

  <space name="thread_space">
    <thread name="test_thread" start="_entry" />
  </space>

  <irq value="10" />
  <heap size="0x100000" />
  <thread name="second_main_thread" start="_start" />
  <use_device name="serial_dev" />
</okl4>

```

Cell elements have the *required* attributes, `name`, `file` and `kernel_heap`. In the event one of these attributes is left unspecified, an error will be raised by Elfweaver.

The `caps` attribute is used to specify the number of cap slots that will be created in each clist belonging to the cell. If left unspecified, this value will default to 1024 cap-slots per clist. The `clists` attributes used to specify the maximum number of capability lists that can be created and or accessed by the cell. Unless otherwise specified, a cell will only be able to access its default clist.

The `direct` attribute is used to specify whether all memory regions and ELF segments belonging to the cell will be placed in the same physical address as the virtual address specified in the ELF program header. This attribute takes precedence over the `physpool` attribute.

The `max_priority` attribute is used to specify the maximum priority of threads running within the cell. If left unspecified, this value defaults to 255 for OKL4 and 31 for OKL4 Nano.

The `mutexes` and `spaces` attributes are used to specify the maximum number of mutexes or spaces that can be created and or accessed by the cell, respectively. Note that a cell can only access the mutexes and spaces created explicitly or implicitly from the XML definition of a cell.

The `platform_control` attribute is used to specify whether a cell has access to the *PlatformControl* system call. Cells do not have access to this system call by default. The `threads` attribute is used to specify the maximum number of threads that can be created and or accessed by the cell. A cell can only access threads that are created explicitly or implicitly from the XML definition of a cell.

The `physpool` and `virtpool` attributes are used to specify the default physical and virtual memory pools to be used for the allocation of physical and virtual memory for objects within the cell. If left unspecified, the physical and virtual memory pools specified by the `kernel` element will be used to allocate physical and virtual memory for objects belonging to the cell.

The cell element may contain one or more instances of the following sub-elements:

- `memsection`
- `heap`
- `stack`
- `segment`
- `patch`
- `irq`
- `thread`
- `mutex`
- `use_device`
- `space`
- `environment`

Each of these sub-elements is described in Sections 3.5.1 to 3.5.7

3.5.1 IRQ

| IRQ Element Attributes | | |
|-------------------------------------|-----------------|-----------------|
| Attribute: | Type: | Description: |
| value | <i>required</i> | The IRQ Number. |
| Example: <i>IRQ Element</i> | | |
| <pre><irq value="10" /></pre> | | |

The `irq` element is used to assign an IRQ to the space of the main thread. This element has a single attribute, `value`, used to specify the IRQ number.

3.5.2 Thread

| Thread Element Attributes | | |
|--|-----------------|--|
| Attribute: | Type: | Description: |
| name | <i>required</i> | The name of the thread. |
| start | <i>required</i> | Starting address of the thread. |
| priority | <i>optional</i> | Priority of the thread. |
| virtpool | <i>optional</i> | Virtual memory pool to be used for memory allocation of the thread. |
| physpool | <i>optional</i> | Physical memory pool to be used for memory allocation of the thread. |
| Example: <i>Thread Element</i> | | |
| <pre><thread name="second_main_thread" start="_start" /></pre> | | |

The `thread` element contains the *required* attributes, `name` and `start`. The `start` attribute is used to specify the starting address of the thread. This may be specified as a virtual address or symbol defined in the ELF file of the program.

The `thread` element contains the *optional* attributes `priority`, `virtpool` and `physpool`. The `priority` attribute is used to specify the priority of the thread, if left unspecified the thread will be allocated a priority of 100. The `virtpool` and `physpool` attributes are used to specify the virtual memory pool and physical memory pool to be used for all memory allocation associated with the thread. The `thread` element may contain a `stack` sub-element.

3.5.3 Mutex

| Mutex Element Attributes | | |
|---|-----------------|------------------------|
| Attribute: | Type: | Description: |
| name | <i>required</i> | The name of the mutex. |
| Example: <i>Mutex Element</i> | | |
| <pre><mutex name="test_space_mutex1" /></pre> | | |

The `mutex` element is used to create a mutex object. The `mutex` element contains a single *required* attribute, `name`, which is used to specify the name of the mutex.

3.5.4 Use Device

| Use Device Element Attributes | | |
|---|-----------------|----------------------------------|
| Attribute: | Type: | Description: |
| name | <i>required</i> | The name of the physical device. |
| Example: <i>Use Device Element</i> | | |
| <pre><use_device name="serial_dev" /></pre> | | |

The `use_device` element is used to assign a physical device to a cell. This element contains a single *required* attribute, `name`, which is used to specify the name of the physical device.

Physical devices are described using the `phys_device` element described in Section 3.3.8.

3.5.5 Space

| Thread Element Attributes | | |
|----------------------------------|-----------------|---|
| Attribute: | Type: | Description: |
| name | <i>required</i> | The name of the space. |
| direct | <i>optional</i> | A boolean value. |
| max_priority | <i>optional</i> | Maximum priority of threads belonging to the space. |
| physpool | <i>optional</i> | Physical memory pool to be used for memory allocation of the space. |
| virtpool | <i>optional</i> | Virtual memory pool to be used for memory allocation of the space. |

Example: Thread Element

```
<space name="test_space">
  <mutex name="test_space_mutex1" />
  <mutex name="test_space_mutex2" />
  <memsection name="normal2" size="0x4000" />
</space>
```

The `direct` attribute is used to specify whether all allocated memory should be placed at the same physical address as the virtual address. This attribute takes priority over the `physpool` attribute.

The `max_priority` attribute is used to specify the the maximum priority of threads residing within the address space. If left unspecified, the maximum priority of the threads will be set to the same value specified by the cell to which the address space belongs. Note that the `max_priority` specified by the space may be greater than the value specified by the cell to which it belongs.

The `space` element may contain one or more instances of the `irq`, `thread`, `mutex` and `memsection` sub-elements.

3.5.6 Environment

| Environment Element Attributes | | |
|--|--------------|---------------------|
| Attribute: | Type: | Description: |
| <i>This element does not contain any attributes.</i> | | |

Example: Environment Element

```
<environment>
  <entry key="test_key" value="42"/>
  <entry key="my_mutex" cap="test_space/test_space_mutex1" />
  <entry key="my_memory" cap="normal" />
  <entry key="some_text" cap="rx" />
  <entry key="some_data" cap="rw" attach="rw" />
  <entry key="my_memory2" cap="test_space/normal2" />
</environment>
```

The `environment` element is used to describe the entries in the cell environment. This is similar to the `environment` element specified by iguana cells.

3.5.7 Memory Sections

| 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 of a file containing the contents 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>phys_addr</code> | <i>optional</i> | Physical address of the base of the memory section. |
| <code>virt_addr</code> | <i>optional</i> | Virtual address of the base of the memory section. |
| <code>physpool</code> | <i>optional</i> | Physical memory pool used to allocate the memory section. |
| <code>virtpool</code> | <i>optional</i> | Virtual 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. |
| <code>cache_policy</code> | <i>optional</i> | The cache policy of the memory section. |
| <code>mem_type</code> | <i>optional</i> | The memory type. |

Example: Memory Section Element

```
<memsection cache_policy="uncached" name="serial_vaddr"
             phys_addr="0x40100000" size="0x1000" />
<memsection mem_type="unmapped" name="ktest_segment" size="0x4000" />
<memsection name="normal" size="0x4000" />
```

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 `okl4` element.

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 at 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. By default, memory sections are aligned on a natural boundary. 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 `x`, 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 `mem_type` attribute should be set to `unmapped` if the memory section is not be mapped.

The `cache_policy` attribute can be used to set the caching policy for the memory section. If left unspecified, Elfweaver will set the value of the `cache_policy` attribute to `default`. Table 3.1 contains the list of

standard cache policies.

Custom cache policies may also be specified, but they must first be declared within the machine element. This is further described in Section 3.3

3.6 Iguana Cell

Iguana Cell 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: Iguana Cell Element

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

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

The `iguana` element is used to specify an Iguana cell and its programs.

The `virtpool` and `physpool` attributes are used to specify the default virtual and physical memory pools, respectively. Where the `iguana` element fails to specify valid virtual and physical memory pools using the `virtpool` and `physpool` attributes, the corresponding values specified by the `kernel` element are used as the virtual and physical memory pools, respectively. 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 `iguana` element.

The `iguana` element may contain the sub-elements, `segment`, `patch`, `extension`, `program` and `pd` each of which are described in Sections 3.6.1 to 3.6.7, below.

The example `iguana` tag described above, specifies a `iguana` element which specifies the `iguana_server` as the root task. The `iguana` element specifies the virtual memory pool `main_virt` and the physical memory pool `main_phys` as the default virtual and physical memory pools.

The `iguana` element also specifies four sub-elements, `segment`, `extension`, `stack` and `heap`. These elements are further described below.

3.6.1 Segment

The `segment` element describes the load location of an ELF segment. Segments are further described in Section 3.9.

3.6.2 Patch

The `patch` element is used to modify the content of the ELF file prior to building the final image. The `patch` element is further described in Section 3.10.

3.6.3 Extension

| 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> | The entry point for the extension. |

Example: *Extension Element*

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

The `extension` element is used to describe an Iguana 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 `iguana` element. Where the `physpool` attribute is left unspecified, the physical memory pool specified by the `physpool` attribute of the `iguana` 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.

An extension element may contain on or more `segment` and/or `patch` elements. The elements `segment` and `patch` are further described in Sections 3.9 and 3.10, respectively.

The above example root program tag contains a single `extension` element which specifies code named `library` contained at the location, `/path/to/extension`.

3.6.4 Stack

The `stack` element is used to describe the root program's stack. Stacks are further described in Section 3.6.6.1.

3.6.5 Heap

The `heap` element is used to specify the root program's heap. Heaps are further described in Section 3.6.6.2.

3.6.6 Programs

Program Element Attributes

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

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>

<program name="vtimer_server" file="path/to/vtimer" server="OKL4_VTIMER_SERVER">
  <virt_device name="vserial10">
  <virt_device name="vserial11">

  <environment>
    <entry key="SERIAL_RESOURCES" cap="/dev/serial_dev">
  </environment>
</program>

<program name="example" file="example.elf">
  <zone name="z1">
    <memsection name="zoned_ms" size="0x100000">
  </zone>
</program>

```

The program element is used to specify an Iguana program. 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 `iguana`. 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 `iguana` 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.

The `platform_control` attribute grants the program access to the PlatformControl system call. By default this attribute is `false`.

The `program` element may contain one or more of the elements `stack`, `heap`, `cmdline`, `memsection`, `environment`, `segment`, `patch`, `thread`, `virt_device` and `zone`. Each of these elements are further described in Sections 3.6.6.1 to 3.6.6.10, below.

The example program tag specifies a program specifies three programs, `demo`, `vtimer_server` and `example`. The program `demo` is located at `path/to/file` and will run with a priority level of 110. The `demo` program will be allocated a stack of size `0x4000` and a heap of size `0x10000`. The `demo` program will be started with two command line arguments, `demo` and `second_arg`. The programs `vtimer_server` and `example` are further described in Sections 3.6.6.9 and 3.6.6.10, respectively.

3.6.6.1 Stack

The `stack` element is used to describe the stack belonging to the program thread. The master capability of the stack will be held by the program's owning protection domain. Stack elements are further described in Section 3.8.

3.6.6.2 Heap

The `heap` element is used to specify the program's heap. As with the `stack` element described above, the master capability of the program's heap will be held by the owning protection domain of the program. Heap elements are further described in Section 3.7.

3.6.6.3 Command Line

The `commandline` element is used to specify command line arguments to the specified program. The `commandline` element may be used to specify one or more `arg` elements. Each `arg` element specifies a single command line argument using the attribute, `value`. These arguments are passed to the program in the order they are declared, that is the first `arg` element is treated as `argv[0]`, the second as `argv[1]`, etc. For example, the example program tag specifies a `commandline` element which specifies two arguments, `demo` which is treated as `arg[0]` and `second_arg` which is treated as `arg[1]`. Elfweaver will pass these arguments unmodified to the program, `demo`.

Elfweaver does not perform any processing on the command line arguments, that is spaces and shell special characters are passed unmodified to the program. The `commandline` arguments are stored near the top of the program's stack.

3.6.6.4 Memsection

The `memsection` element describes the creation of a memory section, which may optionally contain data from a file. Memory sections are further described in Section 3.6.8.

3.6.6.5 Object Environment

Object environments are further described in Section 3.6.10.

3.6.6.6 Segment

The `segment` element describes the load location of an ELF segment. Segments are further described in Section 3.9.

3.6.6.7 Patch

The `patch` element is used to modify the content of the ELF file prior to building the final image. The `patch` element is further described in Section 3.10.

3.6.6.8 Thread

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 `iguana` element is used.

The `thread` element may contain the elements `stack` and `commandline`. These elements are further described in Sections 3.6.6.1 and 3.6.6.3, respectively.

3.6.6.9 Virtual Device

Virtual Device Element Attributes

| Attribute: | Type: | Description: |
|------------|-----------------|---------------------------------|
| name | <i>required</i> | The name of the virtual device. |

Example: Virtual Device Element

```
<program name="vserial_server" file="path/to/vserial" server="OKL4_VSERIAL_SERVER">

  <virt_device name="vserial0" />
  <virt_device name="vserial1" />

  <environment>
    <entry key="SERIAL_RESOURCES" cap="/dev/serial_dev" />
    <entry key="VSERIAL" cap ="/dev/vserial0">
  </environment>

</program>

<program name="v tty_server" file="path/to/vtty" server="OKL4_VTTY_SERVER">

  <virt_device name="vtty0" />

  <environment>
    <entry key="TTY_RESOURCES" cap ="/dev/tty_dev">
  </environment>

</program>
```

The `virt_device` element is used specify the virtual devices that should be instantiated by the device server of the corresponding device class. The `virt_device` element contains a single *required* attribute, `name`, which is used to specify the name of the virtual device.

For a the device server to be granted access to the physical device resouces, described by the `phys_device` element, it should include the automatically generated physical device capability. This capability should be named `"/dev/string"`, where `string` corresponds to the name attribute of the `phys_device` element. The `phys_device` element is further described in Section 3.3.8.

3.6.6.10 Zones

| Zone Element Attributes | | |
|-------------------------|-----------------|-----------------------|
| Attribute: | Type: | Description: |
| name | <i>required</i> | The name of the zone. |

Example: Zone Element

```
<zone name="z1">
  <memsection name="zoned_ms" size="0x100000">
</zone>
```

The zone element is used to specify a particular group of memory sections and/or segments which may be accessed by the program specified by the `program` element. The zone element contains a single *required* attribute, `name`, and an *optional* attribute, `virtpool`. The `name` attribute is used to specify the name of the zone. The `virtpool` element is used to specify the virtual memory pool to be used for the allocation of the memory sections belonging to the zone. If unspecified, the default virtual memory pool is used for the allocation of the memory sections belonging to the zone.

All memory sections and/or segments declared within the `zone` element are placed in the zone. The 1MB region of virtual memory that contains the memory section or segment becomes the *zone window*. Any memory sections not declared as being within the zone are not permitted to reside within the window address range.

A zone is attached to a protection domain with a single set of access permissions calculated using the union of the access permissions of the memory sections residing in the zone. All memory sections and segments residing within the zone will then appear to be attached to the protection domain with these permissions.

Once declared, a zone may be attached to other protection domains or programs using an `environment` element. Object environments are further described in Section 3.6.10. The `memsection` and `segment` elements are further described in Sections 3.6.8 and 3.9, respectively. Protection domains are further described in Section 3.6.7.

3.6.7 Protection Domains

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. |
| <code>platform_control</code> | <i>optional</i> | Grant the program access to the PlatformControl system call. |

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. Though this element is similar to the `program` element, by default no segments or memory sections are mapped into the address space and no threads are started. 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 `iguana` 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.

The `platform_control` attribute grants the `pd` access to the PlatformControl system call. By default this attribute is `false`.

Each `pd` element may contain one or more of the elements `segment`, `thread`, `patch` or `memsection`. The `segment` element describes the load location of an ELF file and is further described in Section 3.9. The `patch` element is used to modify the content of the ELF file prior to building the final image and is further described in Section 3.10. The `thread` element allows the user to specify an additional thread to be started at boot time and is further described in Section 3.6.6.8. Lastly, the `memsection` element describes the creation of a memory section, which may optionally contain data from a file. Memory sections are further described in Section 3.6.8.

The example protection domain tag specifies a protection domain named `isolated` which consists of a single memory section named `make_dynamically`. This `make_dynamically` memory section is of size 16KB. The `attach` attribute describes the access permissions with which the memory section is mapped into the protection domain. The example memory section `make_dynamically` has been attached to the protection domain `isolated` with the permissions `read`, `write` and `execute`.

3.6.8 Memory Sections

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 of a file containing the contents 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>phys_addr</code> | <i>optional</i> | Physical address of the base of the memory section. |
| <code>virt_addr</code> | <i>optional</i> | Virtual address of the base of the memory section. |
| <code>physpool</code> | <i>optional</i> | Physical memory pool used to allocate the memory section. |
| <code>virtpool</code> | <i>optional</i> | Virtual 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. |
| <code>zero</code> | <i>optional</i> | A Boolean value. |
| <code>cache_policy</code> | <i>optional</i> | The cache policy of the memory section. |

Example: Memory Section Element

```
<memsection name="data" file="/path/to/data" virt_addr="0x10000000"
  physpool="somemore" pager="custom" cache\_policy="uncached">
  <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 program or protection domain, which is the parent of the memory section will hold the master capability for the memory section. Capabilities are further described in Section 3.6.9.

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 `iguana` 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. By default, memory sections are aligned on a natural boundary. 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 `x`, 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 `zero` attribute is a Boolean value used to specify whether the memory section is to be zero filled on creation. If left unspecified, the `zero` attribute is set to the default value of `true`, indicating that the memory section should be zero filled on creation. This attribute is used to prevent memory sections mapping to device registers from writing zeros to these registers on creation.

The `cache_policy` attribute can be used to set the caching policy for the memory section. If left unspecified, Elfweaver will set the value of the `cache_policy` attribute to `default`. Table 3.1 contains the list of standard cache policies.

Custom cache policies may also be specified, but they must first be declared within the machine element. This is further described in Section 3.3

The `memsection` element may contain one or more `cap` elements. The `cap` element is further described in Section 3.6.9.

The above example `memsection` tags specify two `memsection` elements. The first `memsection` element, named `data`, is to be located at the virtual address `0x10000000`. This memory section will be allocated out of the physical memory pool `somemore` and will contain the data located at `/path/to/data`. This memory section will not be located at the same virtual and physical addresses and uses a custom pager. The memory section `data` is created with an extra capability named `data_rx` which will be passed on to its parent protection domain or program.

The second example `memsection` element, named `make_dynamically` is a memory section of size `0x4000` bytes and is mapped into the protection domain's address space with the permissions, `read`, `write` and `execute`.

3.6.9 Capabilities

| Capability Element Attributes | | |
|--|-----------------|-------------------------|
| Attribute: | Type: | Description: |
| <code>name</code> | <i>required</i> | Name of the capability. |
| Example: Capability Element | | |
| <pre> <memsection name="data" file="/path/to/data"> <cap name="rx"> <right value="read" /> <right value="execute" /> </cap> </memsection> </pre> | | |

The `cap` element allows the user to create a capability associated with specific rights with regards to a specific object. Currently, one or more `cap` elements may be declared within the elements `segment`, `stack`, `heap` and `memsection`. It is expected that Elfweaver will be extended to allow `cap` elements to be specified within the elements `thread`, `pd` and `program`.

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`.

In addition to the implicit creation of the master capability, read only and read write capabilities are also implicitly created with object. The read only and read write capabilities of an element take on the name of the element followed by `ro` and `rw`. For example the read only and read write capabilities for the `memsection` element specified below will be denoted as `data/ro` and `data/rw`, respectively.

Each `cap` element contains a single attribute, `name`, which is used to denote the name of the capability. The `cap` element contains one or more `right` elements, each of which describes a single right granted by the capability. The `right` element is further described below.

3.6.9.1 Right Element

| Right Element Attributes | | |
|--|-----------------|-------------------|
| Attribute: | Type: | Description: |
| <code>value</code> | <i>required</i> | Name of the right |
| Example: Right Element | | |
| <pre><right value="read" /></pre> | | |
| <pre><right value="execute" /></pre> | | |

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:

- `read`: Read permission.
- `write`: Write permission.
- `execute`: Execute permission.
- `master`: Master rights to the object.

3.6.10 Object Environment

| Object Environment Element Attributes | | |
|--|-------|--------------|
| Attribute: | Type: | Description: |
| <i>This element does not contain any attributes.</i> | | |
| Example: Object Environment Element | | |
| <pre><environment> <entry key="SOMEMEMORY" cap="data/rw" attach="rw" /> <entry key="ANSWER" value="42"> </environment></pre> | | |

Each protection domain contains an object environment mechanism, described by the `environment` tag. The object environment mechanism is provided to enable threads to locate objects specified in the configuration file

that they may access. It should be noted that the object environment is a static data structure and no provision is made for finding objects that are not described in the XML file.

The `environment` tag may contain one or more `entry` elements. The `entry` element is further described below.

3.6.10.1 Entry Elements

| Entry Element Attributes | | |
|--------------------------|-----------------|--|
| Attribute: | Type: | Description: |
| <code>key</code> | <i>required</i> | The name of the entry. |
| <code>value</code> | <i>optional</i> | A word size integer. |
| <code>cap</code> | <i>optional</i> | The capability of the object. |
| <code>attach</code> | <i>optional</i> | Access permissions assigned to the protection domain for the memory section. |

Example: Entry Element

```
<environment>
  <entry key="SOMEMEMORY" cap="data/rw" attach="rw" />
  <entry key="TALKTOME" cap="/extrathread/master" />
  <entry key="TCM_POOL" cap="/tcm" />
  <entry key="ANSWER" value="42">
</environment>
```

The `environment` tag may contain one or more `entry` elements, where each entry element is used to specify a single object.

The `key` attribute is used to specify the name of the entry as a string. This name may then be used by an application to locate the object at run time. By convention, the `key` attribute should be specified in uppercase, though keys will be searched for in a case sensitive manner. The `value` attribute is used to specify a word size integer object. If the `value` attribute is specified, the `cap` and `attach` attributes will be ignored. The `cap` attribute is used to specify the capability corresponding to the object. Capabilities are named in a similar manner to POSIX paths. That is, relative paths are used to access capabilities in the current program and absolute paths are used to access capabilities in other programs. Capabilities are further described in Section 3.6.9. Where the `cap` attribute specifies a capability relating to a memory section, the `attach` attribute is used to specify the permissions given to the memory section within the protection domain. It should be noted that the master capability of every object within the protection domain is automatically added to its object environment.

The example `environment` tag above specifies an object environment containing four entries, `SOMEMORE`, `TALKTOME`, `TCM_POOL` and `ANSWER`.

3.7 Heap

| Heap Element Attributes | | |
|---------------------------|-----------------|---|
| Attribute: | Type: | Description: |
| <code>size</code> | <i>optional</i> | The size of the heap. |
| <code>virt_addr</code> | <i>optional</i> | The virtual address of the base of the heap. |
| <code>phys_addr</code> | <i>optional</i> | The physical address of the base of the heap. |
| <code>direct</code> | <i>optional</i> | A Boolean value. |
| <code>virtpool</code> | <i>optional</i> | Virtual memory pool for the heap. |
| <code>physpool</code> | <i>optional</i> | Physical memory pool for the heap. |
| <code>align</code> | <i>optional</i> | The memory alignment of the heap. |
| <code>attach</code> | <i>optional</i> | The access permissions for the heap. |
| <code>pager</code> | <i>optional</i> | The name of the pager used to map the heap. |
| <code>zero</code> | <i>optional</i> | A Boolean value. |
| <code>cache_policy</code> | <i>optional</i> | The cache policy of the memory section. |
| <code>user_map</code> | <i>optional</i> | Allow the program to access the heap with the MapControl system call. |

Example: Heap Element

```
<heap size="0x10000" />
```

The `heap` element is used to specify the heap of a program in the case of an Iguana cell or the heap of the cell's initial program for all other cells. Note that the master capability of the program's heap will be held by the owning protection domain of the program for Iguana cells.

If one or both of the attributes `virt_addr` and `phys_addr` are left unspecified those values will be determined by Elfweaver. The `direct` attribute is used to specify whether the base of the heap will be located at the same physical address as that specified by the `virt_addr` attribute. Where this is the case, the `direct` attribute should be set to `true`. This attribute takes priority of the `physpool` attribute.

The `zero` attribute is used to specify whether the memory allocated to the heap should be zero filled on creation. If left unspecified, the `zero` attribute is set to the default value of `true`, indicating that the memory should be zero filled on creation. Note that this attribute can only be used by Iguana cells.

The `user_map` attribute is used to allow the program access to the the physical memory backing the heap in calls to MapControl. This attribute can only be used by Iguana cells. By default `user_map` is `false`. Where a program element fails to specify a heap element, a heap of size 65536 bytes will be allocated from the default memory pool.

3.8 Stack

| Stack Element Attributes | | |
|---------------------------------|-----------------|---|
| Attribute: | Type: | Description: |
| <code>size</code> | <i>optional</i> | Size of the stack. |
| <code>virt_addr</code> | <i>optional</i> | Virtual address of the base of the stack. |
| <code>phys_addr</code> | <i>optional</i> | Physical address of the base of the stack. |
| <code>direct</code> | <i>optional</i> | A Boolean value. |
| <code>virtpool</code> | <i>optional</i> | Virtual memory pool used for the stack. |
| <code>physpool</code> | <i>optional</i> | Physical memory pool used for the stack. |
| <code>align</code> | <i>optional</i> | Memory alignment of the stack. |
| <code>attach</code> | <i>optional</i> | The access permissions for the stack. |
| <code>zero</code> | <i>optional</i> | A Boolean value. |
| <code>pager</code> | <i>optional</i> | The pager associated with the memory section. |
| <code>cache_policy</code> | <i>optional</i> | The cache policy of the stack section. |

Example: *Stack Element*

```
<stack size="0x4000" />
```

The `stack` element is used to describe the stack belonging to a thread, which is a program thread for Iguana cells. Also note that for Iguana cells, the master capability of the stack will be held by the program's owning protection domain.

If one or both of the attributes `virt_addr` and `phys_addr` are not specified those values will be determined by Elfweaver. The attribute `direct` is used to specify whether the base of the stack will be set to the same physical address as the address specified by the `virt_addr` attribute. Where this is the case, the `direct` attribute should be set to `true`. This attribute takes priority over the `physpool` attribute.

The `zero` attribute is used to specify whether the memory allocated to the stack will be zero filled on creation. It should be noted that this attribute can only be used by Iguana cells. If left unspecified by an Iguana cell, the `zero` attribute is set to the default value of `true`, indicating that the memory should be zero filled on creation.

Where a `program` element fails to specify a `stack` element, a stack of size 4096 bytes will be allocated from the default memory pool.

3.9 Segments

| Segment Element Attributes | | |
|----------------------------|-----------------|---------------------------------------|
| Attribute: | Type: | Description: |
| <code>name</code> | <i>required</i> | The name of the segment. |
| <code>phys_addr</code> | <i>optional</i> | The physical address of segment. |
| <code>physpool</code> | <i>optional</i> | Physical memory pool for the segment. |
| <code>align</code> | <i>optional</i> | Alignment of the segment. |
| <code>attach</code> | <i>optional</i> | Access permissions for the segment. |
| <code>pager</code> | <i>optional</i> | Pager to be used to map the stack. |
| <code>direct</code> | <i>optional</i> | A Boolean value. |
| <code>protected</code> | <i>optional</i> | A Boolean value. |
| <code>cache_policy</code> | <i>optional</i> | The cache policy of the segment. |

Example: Segment Element

```
<segment name="data" physpool="example_physpool" />
<segment name="sensitive_segment" protected="true" />
```

The `segment` element is used to specify the location of a particular segment of an ELF file in physical memory. Segments are further described in Section 1.2.

The `segment` element may be a sub-element of the `kernel`, `rootprogram`, `program`, `extension` and `pd` elements. The `segment` element allows the user to override the properties of a particular segment. Where these elements fail to list the relevant `segment` elements, they will be allocated dynamically by Elfweaver.

The `direct` attribute may be set to `true` or `false` and is used to specify whether physical address of the segment is to be set to the same address as the virtual address of the segment specified in ELF program header. The `direct` attribute takes priority over both the `phys_addr` and `physpool` attributes. Where the `direct` attribute is left unspecified, Elfweaver will set this attribute to its default value, `false`.

The `physpool` attribute is used to specify the physical memory pool to be used to allocate the segment. Where this attribute is omitted, Elfweaver will allocate the segment using the physical memory pool specified using the `physpool` attribute of the corresponding `program` element. Where `physpool` attribute is also not specified by the corresponding `program` element, Elfweaver will use the physical memory pool specified using the `physpool` attribute of the `rootprogram` element. The `protected` attribute allows the user to store a protected library as separate segments within a program. Currently, only a single program in a system may contain a protected segment and that program may only contain a single protected segment.

The `cache_policy` attribute can be used to set the caching policy for the segment. If left unspecified, Elfweaver will set the value of the `cache_policy` attribute to `default`. Table 3.1 contains the list of standard cache policies.

Custom cache policies may also be specified, but they must first be declared within the machine element. This is further described in Section 3.3

The segments are mapped into the address space of a protection domain at the virtual addresses specified in the ELF program header. The segment name is mapped to the segment number via a mapping table found in the `.segment_names` section of the ELF file. The mapping table may be generated using the `elfadorn` command in Elfweaver, which is further described in 2.6. The virtual address of a segment cannot be modified using Elfweaver.

The first example segment tag describes a segment element named `data` which is to be allocated from the physical memory pool, `example_physpool`. The second segment element describes a protected segment element named `sensitive_segment`.

3.10 Patch

| Patch Element Attributes | | |
|--------------------------|-----------------|--|
| Attribute: | Type: | Description: |
| address | <i>required</i> | The address at which the data is stored. |
| value | <i>required</i> | The data to be stored |
| bytes | <i>optional</i> | Size of the data in bytes. |

Example: Patch Element

```
<patch address="__phys_addr_ram" value="0xa0000000" bytes="4"/>
```

The `patch` XML element allows the content of a segment to be modified prior to building the image. For example the `patch` element allows RAM and ROM addresses to be written into the kernel as appropriate.

As with the `segment` element described above, the `patch` element may be a sub-element of the elements `kernel`, `rootprogram`, `program`, `extension` and `pd`.

The `patch` element contains the *required* attributes `value` and `address`. The `address` attribute is used to specify a location in virtual memory to be modified and may be specified either as a memory location in hexadecimal (with a `0x` prefix, or as a symbol name. The `value` attribute is used specify the data to be stored at the location specified by the `address` attribute.

The `patch` element contains a single *optional* attribute `bytes`, which is used to specify the size of the data specified by the attribute `value` in bytes. Note that if the `address` is specified as a virtual memory location, the size of the data specified in `value` must be specified in the `bytes` attribute. However if the `address` is specified as a symbol name as in the Example Patch Tag, the size of the data in `value` must only be specified using the `bytes` attribute where the size of the variable name is not internally specified in the ELF file.

For example, the example patch tag specifies that the data at the virtual address, `__phys_addr_ram` is set to `0xa0000000` as a four byte value.

3.11 Memory pools

As OKL4 allocates virtual and physical memory from memory pools, Elfweaver allows the user to group both virtual memory and physical memory into memory pools. This is achieved via the use of the elements `virtual_pool`, which is used to group virtual memory, and `physical_pool`, which is used to group physical memory.

3.11.1 Virtual Memory Pools

Virtual Memory Pool Element Attributes

| Attribute: | Type: | Description: |
|-------------------|-----------------|--------------------------|
| <code>name</code> | <i>required</i> | Name of the memory pool. |

Example: *Virtual Memory Pool Element*

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

The `virtual_pool` element contains a single *required* attribute, `name`, which is used to associate a name with the memory pool which may be subsequently used by other elements to uniquely refer to the specified virtual memory pool.

3.11.2 Physical Memory Pools

Physical Memory Pool Element Attributes

| Attribute: | Type: | Description: |
|---------------------|-----------------|--------------------------|
| <code>name</code> | <i>required</i> | Name of the memory pool. |
| <code>direct</code> | <i>optional</i> | A Boolean value. |

Example: *Physical Memory Pool Element*

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

<physical_pool name="somore">
  <memory src="more_ram" size="0x80000" />
</physical_pool>

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

The `physical_pool` element contains a single *required* attribute, `name`, which is used to associate a name with the memory pool which may be subsequently used by other elements to uniquely refer to the specified physical memory pool. The `physical_pool` attribute also contains a single *optional* attribute, `direct`

which is used to specify a direct physical memory pool. When memory is allocated from a direct physical memory pool, a memory block at corresponding virtual address is also allocated. As with ordinary physical pools, the `name` attribute of direct physical pools may also be subsequently used by other elements to uniquely refer to the specified direct physical memory pool.

4 Customized Elements

Elfweaver provides a set of XML elements to specify the various aspects of the intended layout of the image in the *configuration file*. Though the elements supplied by Elfweaver cover most of layout of the configuration file, they may not adequately cover the description of architecture specific behaviour. Thus, Elfweaver allows the user to add customized tags to the configuration file to better describe architecture specific aspects of the resulting image. Note that customization can only be achieved by creating new elements, that is, new attributes **cannot** be added to existing elements.

This chapter provides a description of the steps that must be followed to create a new XML element. The location of the extensions is covered in Section 4.1, followed by an overview of XML namespaces in Section 4.2. A step by step guide to creating a customized element is provided in Section 4.3. The configuration file and the generic elements provided by Elfweaver are further described in Chapter 3.

4.1 Elfweaver Custom Path

All user extensions and cell specifications must be implemented in a user specified location, obtained from the user environment variable, `$ELFWEAVER_CUSTOM_PATH`. Alternatively, the custom path may be specified using the command line option `-c` or `--custom-path` to the `merge` command described in Section 2.2. The custom directory must have the following structure:

Example: *Custom directory structure.*

```
$ELFWEAVER_CUSTOM_PATH/  
  cells/  
  extensions/  
    __init__.py  
    <ext_name1>_cust.py  
    <ext_name2>_cust.py  
    ....
```

As the *extensions* directory is a Python package, it must contain an `__init__.py` file, though this may be empty. These extensions must reside within Python modules with the suffix, `_cust.py`.

4.2 XML Namespaces

Custom sub-elements may be created by adding a new namespace to Elfweaver. The default namespace containing the generic elements provided by Elfweaver is `okl4:elfweaver`. The user may add one or more additional namespaces as required.

Elfweaver will search the `$ELFWEAVER_CUSTOM_PATH` for any customizations. However, in order to use an element described by an added namespace, that namespace must be specified in the *configuration file* using the `xmlns` attribute of the `image` element. The `image` element is further described in Section 3.1.

The following example specifies a custom sub-element, `myexample`, which belongs to the namespace `ex` or `example`. `myexample` is a sub-element of the Elfweaver provided element, `machine`.

Example: *Image tag specifying an additional namespace.*

```
<image xmlns="okl4:elfweaver" xmlns:ex="okl4:example">
  <machine>
    <word_size size="0x20" />
    <virtual_memory name="virtual">
      <region base="0x1000" size="0xcffff000" />
    </virtual_memory>
    <physical_memory name="physical">
      <region base="0xa0000000" size="0x38000000" />
    </physical_memory>
    <page_size size="0x1000" />
    <ex:myexample value="test_machine" />
  </machine>
  ...
</image>
```

Custom sub-elements may be added to the following elements:

- `machine` element described in Section 3.3.
- `kernel` element described in Section 3.4.
- `okl4` element described in Section 3.5.

4.3 Creating an XML Extension

This section provides a step by step guide to creating a customized element. Creating the element is described in Section 4.3.1 followed by a guide on specifying how the newly created element is to be parsed by Elfweaver. Lastly, registering the newly created element is covered in Section 4.3.3.

4.3.1 Creating a new element

Each `Element` supported by Elfweaver must contain a name, a list of attributes, which may be either *required* or *optional*, and a list of allowed child elements. Thus, the first step in creating a new XML element is to define the name, list of attributes and the list of allowed child elements.

Example: *Element Interface.*

```
class Element(object):
    """
    Defines a type of element, rather than a particular element
    in an XML file. That is, it defines the attributes and
    children a given node, rather than representing a particular node
    with specific attributes and elements.
    """

    def __init__(self, __name, *children, **attributes):
        """
        Construct an Element with a given tagname 'name'. Attributes
        are a dictionary of valid attributes. Children are a list of
        child elements.
        """
```

The above example provides a brief overview of the `weaver.xml_api.Element` class provided by Elfweaver. As described in this example, the `Element` class is a generic class which defines the attributes and children of a given element as opposed to representing the attributes and children of a particular element.

Thus, this class can be extended as specified in the example below to describe a new element, `myexample`, as a sub-element of the `machine` element provided by Elfweaver. The `myexample` element has a single *optional* attribute, `value`, which is specified as a string. New elements should be specified in a `*_cust.py` module as described in Section 4.1.

Example: *Creating a new XML element.*

```
from weaver.xml_api import Machine_el, Element, str_attr, update_element

Myexample_el = Element("myexample",
                       value = (str_attr, "optional"))

MyMachine_el = update_element(Machine_el, Myexample_el)
```

4.3.2 XML Collector

Once the configuration file is specified using the elements provided by Elfweaver, together with any customized elements, it must then be parsed by Elfweaver. The parsing of the configuration file is handled by the `weaver.parse_spec.XmlCollector` class. Each top-level `Element` should have a corresponding

XmlCollector sub-class. Each sub-class should in turn implement the `collect_xml` method, which parses the tags containing matching patterns to that described by the corresponding Element.

Example: *XML Collector Interface.*

```
class XmlCollector(object):
    element = NotImplemented

    def __init__(self, *args, **kwargs):

    def register(cls):
        """
        Register the implementation class with an element
        and its name.
        """

    def collect_xml(self, elem, *args, **kwargs):
        """
        Parse XML elements and create an intermediate representation
        of the element.
        """
        raise NotImplementedError
```

The following top-level elements have an `XmlCollector` associated with them:

- machine element.
- kernel element.
- VirtualPool element.
- PhysicalPool element.

The following example provides an overview of the `ParsedElement` class which is used in implementing the `collect_xml` method.

Example: Parsed Element Interface.

```
class ParsedElement(object):
    """This represents an element once it has been parsed."""

    def dump(self):
        """
        Return an XML representation of the element and its children.
        """

    def find_children(self, tagname):
        """
        Return all children of the element with the given tagname.
        """

    def find_child(self, tagname):
        """
        Return a child of the element with the given tagname.
        """

    def find_all_children(self, tagname):
        """
        Return all children of the element, or its children
        with the given tagname.
        """
```

Where a top-level element is being extended, the easiest method of creating a collector is to create a sub-class of the original collector and extend the `collect_xml` method. The first argument to the `collect_xml` method is an object of type `ParsedElement`, which represents an element that has been parsed from the `weaver.xml` file. Attributes of the original `Element` may be accessed as normal Python attributes of objects. Child elements may be accessed using the API methods provided by the `ParsedElement` class, described above. Note that new collectors should be specified in a `*_cust.py` module as described in Section 4.1.

Example: Extending the XML collector.

```
from weaver.xml_api import Machine

class ExampleMachine(Machine):

    element = MyMachine_el

    def collect_xml(self, machine_el, ignore_name):
        Machine.collect_xml(self, machine_el, ignore_name)
        print "machine MYEXAMPLE =", \
              machine_el.find_child("myexample").value
```

4.3.3 Registering a customisable object

Once new elements and collectors have been specified as described in Section 4.3.1 and 4.3.2, they must be

associated with a particular XML namespace. This is achieved by creating a sub-class of the `Customisable` class described below.

Example: *Extension Interface.*

```
class Customisable(object):
    """
    This abstract class provides a way to to define namespace specific
    customisations of existing classes.
    """

    def __init__(self):

    def register(cls, uri):
        """
        Register this class as being the one associated with the given
        namespace URI.
        """
        register = classmethod(register)

    def customise(self):
        """
        Override the default element registrations
        with customised classes.
        """
        raise NotImplementedError
```

Once a sub-class of the `Customisable` class described above is created, the `customise` method must be implemented. Note that the `customise` method must call the `register` class method of the classes in which it is to be included. This informs Elfweaver of the handler to be used for each element in the configuration file.

The last line of the example described below registers the extension. Each `Customisable` sub-class must register itself when it is imported so that its XML namespace is stored by Elfweaver.

Example: *Creating a customisable object.*

```
from weaver.extensions import Customisable

class ExampleCustomisable(Customisable):
    def customise(self):
        ExampleMachine.register()

ExampleCustomisable.register("okl4:example")
```


Appendix A: Example Configuration File

```
\begin{verbatim}
<?xml version="1.0" ?>
<!DOCTYPE image
  SYSTEM 'weaver-2.0.dtd'>

<image>
  <machine>
    <cpu name="xscale" />

    <word_size size="0x20" />

    <virtual_memory name="virtual">
      <region base="0x80000000" size="0x50000000" />
    </virtual_memory>

    <physical_memory name="sdram">
      <region base="0x5c000000" size="0x40000" rights="all" />
    </physical_memory>

    <physical_memory name="rom">
      <region base="0xa3800000" size="0x800000" />
    </physical_memory>

    <physical_memory name="physical">
      <region base="0xa0000000" size="0x3800000" />
    </physical_memory>

    <phys_device name="timer_dev">
      <physical_memory name="timer_mem0">
        <region base="0x40a00000" size="0x1000" />
      </physical_memory>
      <interrupt name="int_timer0" number="27" />
      <interrupt name="int_timer1" number="28" />
    </phys_device>

    <phys_device name="serial_dev">
      <physical_memory name="serial_mem0">
        <region base="0x40100000" size="0x1000"
          cache_policy="uncached" />
        <region base="0x40101000" size="0x1000" />
        <region base="0x40102000" size="0x1000" />
      </physical_memory>
      <physical_memory name="serial_mem1">
        <region base="0x40103000" size="0x1000" />
      </physical_memory>
      <interrupt name="int_serial0" number="22" />
    </phys_device>
  </machine>
</image>

```

Example Configuration File - Continued

```
<phys_device name="rtc_dev">
  <physical_memory name="rtc_mem0">
    <region base="0x40900000" size="0x1000" />
  </physical_memory>
  <interrupt name="int_rtc0" number="31" />
</phys_device>

<page_size size="0x1000" />
<page_size size="0x10000" />
<page_size size="0x100000" />
</machine>

<physical_pool direct="true" name="sdram">
  <memory src="sdram" />
</physical_pool>

<physical_pool direct="true" name="rom">
  <memory src="rom" />
</physical_pool>

<virtual_pool name="virtual">
  <memory src="virtual" />
</virtual_pool>

<physical_pool direct="true" name="physical">
  <memory src="physical" />
</physical_pool>

<kernel file="l4kernel" physpool="physical" virtpool="virtual">
  <config>
    <option key="threads" value="1024" />
  </config>
</kernel>

<okl4 priority="100" clists="256" file="cell_init"
  kernel_heap="0x200000" mutexes="256" name="ktest"
  spaces="256" platform_control="True">
  <mutex name="cell_mutex" />

  <environment >
    <entry key="test_key" value="42" />
    <entry key="my_mutex" cap="test_space/test_space_mutex1" />
    <entry key="my_memory" cap="normal" />
    <entry key="some_text" cap="rx" />
    <entry key="some_data" cap="rw" attach="rw" />
    <entry key="my_memory2" cap="test_space/normal2" />
  </environment>
```

Example Configuration File - Continued

```
<memsection cache_policy="uncached" name="serial_vaddr"
    phys_addr="0x40100000" size="0x1000" />
<memsection mem_type="unmapped" name="ktest_segment"
    size="0x4000" />
<memsection name="normal" size="0x4000" />

<space name="test_space">
    <mutex name="test_space_mutex1" />
    <mutex name="test_space_mutex2" />
    <memsection name="normal2" size="0x4000" />
</space>

<space name="thread_space">
    <thread name="test_thread" start="_entry" />
</space>

<irq value="825" />

<heap size="0x100000" />

<thread name="second_main_thread" start="_start" />

<use_device name="serial_dev" />
</okl4>
</image>
```


