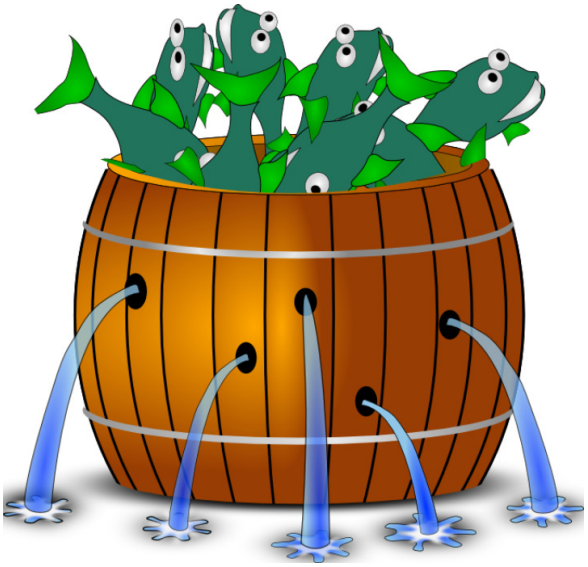
Barrelfish Project
ETH Zurich



Mackerel User Guide

Barrelfish Technical Note 2

Barrelfish project

11.03.2013

Systems Group, ETH Zurich
c/o Department of Computer Science
ETH Zurich CAB F.79, Universitätstrasse 6, Zurich 8092, Switzerland

`http://www.barrelfish.org/`

Revision History

Revision	Date	Author(s)	Description
1.2	31.05.2010	TR	Initial version under new formatting
1.3	15.08.2011	TR	Added support for imports and new command-line options
1.4	09.12.2011	TR	New mapping for constants
1.5	11.03.2013	TR	Added quick reference, corrected errors, and clarified field insertions

Contents

Chapter 1

Introduction and Usage

Mackerel is a domain-specific language used to describe hardware devices and the in-memory formats of registers and data structures like descriptor lists, page-table entries, etc.

The Mackerel language itself is designed to be easy to transcribe into from a hardware data book or similar description - most of the constructs correspond to the kinds of concepts one finds in hardware descriptions.

A feature of the Mackerel language is that, while traditional comments in Mackerel specifications are allowed, explanatory text in English is included as part of a Mackerel specification itself, and used to generate debugging code which explains the meaning of register values.

Typically, it takes as input a specification written in the Mackerel language and outputs a C header file consisting of a large number of C inline function definitions to access and manipulate registers and in-memory data structures, format them for printing in human-readable form, etc.

The Mackerel compiler is written in Haskell using the Parsec parsing library. A quick roadmap of the source code files is provided in Chapter 7.

To manually invoke Mackerel, run:

```
> mackerel --shift-driver -c devFile.dev -o devFile.h
```

This will translate a device file into a single C header file containing all relevant functions for accessing the device, as described in Chapter 4.

If you wish to use the (deprecated) bitfield-based back-end instead, you can use:

```
> mackerel --bitfield-driver -c devFile.dev -o devFile.h
```

This backend should not be used for new code, as it is not portable across compilers. It is described in Chapter 5.

1.1 Command-line options

The complete list of comand-line options supported by the current Mackerel compiler is as follows:

- `-c` *filename* or `--input-file=`*filename*: This option is mandatory and specifies the pathname to be read. This pathname must specify the file: the main input file will not be searched for along the search path.
- `-I` *dir* or `--include-dir` *dir*: this option can be supplied multiple times and specifies the search path for imported files.
- `-v` or `--verbose`: Increases the verbosity level of the compiler.
- `-o` *filename* or `--output-file` *filename*: this option, if supplied, specifies the name of the C header file which will be written by the compiler. If not given, the filename will be derived from the input filename by removing all the directory components (i.e. it will reside in the current working directory) and appending `.h`.
- `-S` or `shift-driver`: Specifies the shift driver backend; this is the default.
- `-B` or `bitfield-driver`: Specifies the bitfield driver backend; this is deprecated. Furthermore, imports are (deliberately) not supported by the bitfield driver. If multiple `-S` and `-B` options are specified, the last one determines the compiler output.

Chapter 2

Lexical Conventions

The following convention has been adopted for Mackerel. It is similar to the convention opted by modern day programming languages like C and Java.

Whitespace: As in C and Java, Mackerel considers sequences of space, newline, tab, and carriage return characters to be whitespace. Whitespace is generally not significant.

Comments: Mackerel supports C-style comments. Single line comments start with `//` and continue until the end of the line. Multiline comments are enclosed between `/*` and `*/`; anything inbetween is ignored and treated as white space.

Identifiers: Valid Mackerel identifiers are sequences of numbers (0-9), letters (a-z, A-Z) and the underscore character "_". They must start with a letter or "_".

$$
\begin{aligned}
identifier &\rightarrow (letter \mid \_)(letter \mid digit \mid \_)^* \\
letter &\rightarrow (\mathsf{A} \ldots \mathsf{Z} \mid \mathsf{a} \ldots \mathsf{z}) \\
digit &\rightarrow (\mathsf{0} \ldots \mathsf{9})
\end{aligned}
$$

Note that a single underscore "_" by itself is a special, "don't care" or anonymous identifier which is treated differently inside the language.

Integer Literals: A Mackerel integer literal is a sequence of digits, optionally preceded by a radix specifier. As in C, decimal (base 10) literals have no specifier and hexadecimal literals start with `0x`. Binary literals start with `0b`.

In addition, as a special case the string `1s` can be used to indicate an integer which is composed entirely of binary 1's.

$$
\begin{aligned}
digit &\rightarrow (\mathsf{0} \ldots \mathsf{9})^1 \\
hexadecimal &\rightarrow (\mathsf{0x})(\mathsf{0} \ldots \mathsf{9} \mid \mathsf{A} \ldots \mathsf{F} \mid \mathsf{a} \ldots \mathsf{f})^1 \\
binary &\rightarrow (\mathsf{0b})(\mathsf{0}, \mathsf{1})^1 \\
allones &\rightarrow \mathsf{1s}
\end{aligned}
$$

Reserved words: The following are reserved words in Mackerel:

```
addr       also       bytewise   constants  datatype   device
io         lsbfirst   many       msbfirst   pci        regarray
register   regtype    space      stepwise   type       valuewise
import
```

Special characters: The following characters are used as operators, separators, terminators or other special purposes in Mackerel:

```
{ } [ ] ( ) + - * / ; , . _ =
```

Chapter 3

Declarations

A Mackerel source file must consist of zero or more *import* declarations, followed by a single *device specification*, which itself consists of further specifications of registers, register types, etc.

3.1 Imports

An import declaration makes the definitions in a different device file available in the current device definition, as described below. The syntax of an import declaration is as follows:

```
import device;
```

device is the name of a device from which to import definitions.

The Mackerel compiler will search for a file with the appropriate name and parse this at the same time as the main file, along with this file's imports, and so on. Cyclic dependencies between device files will not cause errors, but at present are unlikely to result in C header files which will successfully compile.

3.2 Devices

A device declaration in Mackerel specifies a particular type of hardware device (such as an ioAPIC or a particular Ethernet controller). The syntax is as follows:

```
device name [lsbfirst|msbfirst] ( args ) "description"
{
  declaration;
  ...
};
```

name is an identifier for the device type, and will be used to generate identifiers in the target language (typically C). The name of the name of the device *must* correspond to the filename of the file, including case sensitivity: for example, the file `xapic.dev` will define a device type of name `xapic`.

lsbfirst or msbfirst optionally specifies the *bit order* of the declarations inside the device file; if specified, it must be `lsbfirst` (the default) or `msbfirst`.

Note that this does not say *anything* about the endianness of the hardware device, or the host platform, or the execution environment. It simply says which order the bitfields in particular registers are listed in the declarations which follow – least-significant bits first or most-sigificant bits first.

The reason for this feature is that Mackerel specifications are typically transcribed from chip datasheets, and while these are generally unambiguous there is no clear convention on which order to list a hardware register bitfields in a datasheet. This feature means that the Mackerel specification can more closely resemble to text in the datasheet.

When transcribing a datasheet into a Mackerel specification, if the fields in a register or register type are listed starting with the one which includes bit 0, specify `lsbfirst`. If the fields are listed starting with the one which includes the high bit in the register, specify `msbfirst`. In both cases type the fields in the order they are printed in the datasheet.

args are the arguments that specify how to access the device. They are treated as base addresses in one of several *address spaces* supported by Mackerel (see Section 3.3 below). Typical devices have a single base address in either memory or I/O space, though some device types (of which only one instance ever exists, and at a fixed address) have no arguments, and others may have more if they have multiple register sets which can appear at different addresses.

The arguments will become arguments to the function that intializes the struct used to represent the device in Mackerel-generated C. They are also used within the device declaration to specify the locations of individual registers or groups of registers.

Arguments are declared in C-style type notation as either of type `address` (in memory space) or `io` (in I/O space). By convention a single device argument is named `base`.

description is a string literal in double quotes, which describes the device type being specified, for example `"AC97 Baseline Audio"`.

After the description string follows in curly brackets a series of declarations which specify details of the device hardware. Each declaration must be one of the following:

- A `space` declaration, describing a device-specific address space for registers in addition to the built-in address spaces.
- A `constants` list, analogous to a C enumeration type
- A `register` declaration, describing a particular hardware register or group of registers
- A `regtype` definition, giving the type and format of other hardware register values.

3.3 Address spaces

Registers defined in a Mackerel file reside at specified addresses in an *address space* (unless they are specified as `noaddr`, see Section 3.6).

There are currently 3 built-in address spaces that Mackerel understands, and addresses in these spaces require both a base and an offset. Base values are given as an arguments to the device specification, and so each device argument must be declared as being of one of the following types:

addr : Physical memory, for memory-mapped I/O. Register locations in physical address space are given as offsets relative to a base address which is an argument to the device definition.

io : I/O port space, for `ia32` machines. An I/O address is a 16-bit port number, and register locations in I/O port space are given as offsets relative to a base address which is an argument to the device definition.

pci : PCI Configuration space. A PCI configuration space address is a 4-tuple of *(bus, device, function, offset)*, however, Mackerel only deals in integer offsets from an opaque "base" address, allowing for different addressing schemes depending on hardware.

In addition, Mackerel allows the specification of per-device address spaces. This feature can express a variety of hardware features, for example processor model-specific registers, co-processor registers, device registers that must be accessed indirectly through another index registers, etc.

The syntax for defining a new address space is:

```
space name(index) bytewise|valuewise|stepwise(step)
        "description";
```

name is an identifier for the address space, and is used in register declarations instead of a builtin space such as `addr` or `io`.

index is an identifier for the argument giving the address for the registers

bytewise or valuewise defines how registers in the new space are addressed. Bytewise addresses are like conventional memory addresses, for example two adjacent 32-bit registers in a bytewise-addressed space will have offsets that differ by 4. All the built-in address spaces (`addr`, `pci`, and `io`) are bytewise-addressed.

In contrast, registers in a valuewise-addressed space are indexed by a simple integer regardless of size, meaning that two adjacent 32-bit registers will have addresses that differ only by 1. A good example of a valuewise address space is the set of ia32 model-specific registers (MSRs).

If neither of these is sufficient, `stepwise` allows you to explicitly specify how many bytes are occupied by each address in the address space. `bytewise` is therefore a synonym for `stepwise(1)`, but `stepwise(4)` might be useful for devices (such as some IOAPICs) where registers are indexed by 32-bit values but can be accessed as 64-bit quantities.

"description" is a string literal in double quotes, which describes the constant type being specified, for example `"PHY control registers"`.

Register addresses in user-defined address spaces are always given simply as offsets, since the base is implicit in the device definition.

Mackerel files cannot currently say how registers in a particular space are accessed; this functionality must be provided externally by the programmer (typically by short inline functions).

Here is how ia32 MSRs are defined:

```
space msr valuewise "Model-specific Registers";
register platform_id ro msr(0x17) "Platform ID" {
    _ 50;
    id 3 "platform id";
    _ 11;
};
...
```

As a second example, here is a set of space declarations (and associated index and data registers) from the legacy PC real-time clock (RTC) definition:

```
space std(idx) valuewise "Standard register space";
register ndx rw io(base, 0x70) "Standard index"
             type(uint8);
register target rw io(base, 0x71) "Standard target"
             type(uint8);
regarray standard rw std(0x00)[256] type(uint8);

space ext(idx) valuewise "Extended register space";
register endx rw io(base, 0x72) "Extended index"
             type(uint8);
register etarget rw io(base, 0x73) "Extended target''
             type(uint8);
regarray extended rw std(0x00)[256] type(uint8);
```

3.4 Constants

A `constant` declaration defines a series of related values, for example, the possible values of a given register field. Each constant type consists of an identifier for the type, a textual description, and a series of possible values. Each value in turn consists of an identifier for the value, the numerical value itself, and a textual description of its meaning.

The syntax is:

```
constants name [width( width )] "description" {
   name1 = value1 ["description1"] ;
   ...
};
```

name is an identifier for the constant type, and will be used to generate identifiers in the target language (typically C). The scope of this identifier is the enclosing device specification.

width is optional, and specifies the width of the constant value in bits. This is useful, for example, when the type is used as the type of a register.

"description" is a string literal in double quotes, which describes the constant type being specified, for example `"Vector delivery mode"`.

Each constant value is given as follows:

name$_i$ is an identifier for the particular value of the constant type. The scope of this identifier is the enclosing constant declaration, not the device declaration as a whole.

value$_i$ is an expression giving the corresponding numerical value. This will usually be a literal like `0b1101` or `0xf`, though it can potentially be any arithmetic expression supported by Mackerel.

"description$_i$**"** is an optional string literal in double quotes, which describes the meaning of the value being specified, for example `"Lowest priority"`. If not specified, it defaults to the identifier name$_i$.

Here is a complete example of a constants declaration, taken from the xAPIC definition:

```
constants vdm "Vector delivery mode" {
       fixed   = 0b000 "Fixed";
       lowest  = 0b001 "Lowest priority";
       smi     = 0b010 "SMI";
       nmi     = 0b100 "NMI";
       init    = 0b101 "INIT";
       startup = 0b110 "Start Up";
       extint  = 0b111 "ExtINT";
};
```

Note that the collection of constant values does not have to be complete or contiguous.

3.5 Register types

A `regtype` declaration defines a data type corresponding to the format of one or more registers; it can be thought of as a translation of the bitfield descriptions typically found in data sheets. A `regtype` doesn't define any particular hardware registers (see `register` declarations below), merely a data type for representing their contents. The syntax is as follows:

```
regtype name ["description"] {
   fieldname_1 width [type(fieldtype)] [attr] ["description"];
   ...
};
```

name is an identifier for the register type. Its scope is the enclosing device declaration.

description is an optional string in double quotes, giving a description of the register type, for example `"LVT monitor"`. If not present, it defaults to the identifier.

The field descriptions declare the individual bitfields which make up the register type. The order in which these are given is determined by the `lsbfirst` or `msbfirst` declaration in the `device` specification: if `lsbfirst`, the first field given starts at bit 0, etc.

fieldname is an identifier for the bit field. It can be given as an underscore (_) to indicate "don't care" or "reserved" fields, in which case the field is assumed to have an attributed of `rsvd` (see below), and no description is needed.

width is an integer giving the width of the field in bits.

fieldtype is an optional identifier giving the type of this field. This must be either the identifier of a `constants` declaration somewhere in the enclosing `device` declaration, for example `type(vdm)`, or a qualified name of the form *device*`.`*constdef*, where *device* is the name of a device definition previously imported.

attr is an optional field attribute. See section 3.9 for more information.

description is an optional string in double quotes, giving a description of the field, for example `"Logical APIC ID"`. If not present, it defaults to the identifier.

Here is a complete example of a `regtype` declaration, taken from the xAPIC definition:

```
regtype lvt_lint "LVT Int" {
  vector    8 "Vector";
  dlv_mode  4 type(vdm) "Delivery mode";
  _         1;
  status    1 "Delivery status";
  pinpol    1 "Pin polarity";
  rirr      1 "Remote IRR";
  trig_mode 1 "Trigger mode";
  mask      1 "Mask";
  _         15;
};
```

3.6 Registers

A `register` declaration defines a particular hardware register on the device.

```
register name [attr] [also]
            { noaddr | space( address ) } ["description"] type ;
```

name is an identifier for the register. Its scope is the enclosing `device` declaration.

attr is an (optional) attribute. See Section 3.9 for more information.

space gives the address space of this register (e.g. `addr`, `io`, `pci`, or a per-device user-defined address space).

address gives the address of the register in the address space. For builtin address spaces this is given as a comma-separated pair of *(base, offset)* where *base* is a device argument of the correct type and *offset* is an integer-valued Mackerel expression. For user-defined address spaces, the address is a single Mackerel integer expression giving the register index.

By default, it is an error to specify more than one register at the same location, or at locations which overlap. By preceding the address specifier with the `also` keyword, you can allow registers to coexist at the same location.

noaddr is an alternative to the address space definition, and is used for registers which have no address (or, alternatively, an implicit address). A good example of this kind of register is a coprocessor register which requires custom assembler instructions to read and write. Mackerel generates

slightly different access code for this type of register, specified later in this document, and requires the developer to supply the raw register access functions.

description is an optional string in double quotes, giving a description of the register, for example `"Error status"`. If not present, it defaults to the name identifier.

type gives the format of the register. It can consist of `type(`*name*`)`, where *name* is either:

- the identifier of a register type previously declared with a `regtype` declaration;
- the identifier of a constants type previously declared with a `constants` declaration which included an explicit `width()` specifier;
- one of the built in register types `uint8`, `uint16`, `uint32`, or `uint64`; or
- a qualified name of the form *device*`.`*type*, where *device* is the name of a device definition previously imported.

Alternatively, the type can be given as a sequence of fields in braces, exactly as in a `regtype` declaration. In effect, this latter form defines both a register type and a register at the same time, with the same name.

Here are some examples of `register` definitions:

```
register dfr rw at (base, 0x00e0) "Destination Format" {
  _           28 mb1;
  model       4 type(model_type) "Model";
};
```

```
register apr ro at (base, 0x0090) "Arbitration priority" type(priority);
```

```
register isr0 ro at (base, 0x0100) "ISR bits 0-31" type(uint32);
```

```
register cr4 noaddr "Control register 4" {
  _           49 mbz;
  smxe        1 "SMX enable";
  vmxe        1 "VMX enable";
  _           2 mbz;
  osxmmexcpt  1 "OS support for unmasked SIMD FP exceptions";
  osfxsr      1 "OS support for FXSAVE and FXRSTOR instructions";
  pce         1 "Performance-monitoring counter enable";
  pge         1 "Page global enable";
  mce         1 "Machine-check enable";
  pae         1 "Physical address extensions";
  pse         1 "Page size extensions";
  de          1 "Debugging extensions";
  tsd         1 "Time stamp disable";
  pvi         1 "Protected-mode virtual interrupts";
  vme         1 "Virtual 8086 mode extensions";
};
```

3.7 Register Arrays

A `regarray` declaration defines an array of hardware registers on the device, all of which have the same type and attributes.

```
regarray name [attr] [also]
            space( address ) ["description"] type ;
```

All fields are as for register declarations, except:

address gives the address of the registers which form the array in the address space. This is given as for a single `register` declaration, followed by an array specifier in square brackets.

There are three forms of array specifier. The simplest looks like this:

```
regarray ier ro addr(base, 0x0480) [ 8 ]
             "IER" type(uint32);
```

This specifies a contiguous array of 8 32-bit registers starting at `base + 0x0480`. This example therefore occupies 32 bytes of IO space. Alternatively:

```
regarray ier ro addr(base, 0x0480) [ 8; 0x10 ]
             "IER" type(uint32);
```

This second example specifies 8 32-bit registers, which occur every 16 bytes, starting at `base + 0x0480`. Finally, one can explicitly list the individual locations for the array (note: the back-end does not yet generate correct code for this case):

```
regarray ier ro addr(base, 0x0480) [ 0, 0x10, 0x18 ]
             "IER" type(uint32);
```

This third example specifies 3 32-bit registers, at offsets of `0x480`, `0x490`, and `0x498` from `base`.

3.8 Data types

A `datatype` declaration defines a data type corresponding to the format of a structure in memory; it can be used to represent in-memory hardware-defined datastructures like page table entries or DMA descriptors.

The syntax is as follows:

```
datatype name [lsbfirst|msbfirst (wordsize)] ["descr"] {
   fieldname_1 width [type(fieldtype)] [attr] ["description"];
   ...
};
```

name is an identifier for the data type. Its scope is the enclosing device declaration.

lsbfirst or msbfirst together with a (required) word size specify how the fields in the specification are ordered. This is described in more detail below.

description is an optional string in double quotes, giving a description of the register type, for example `"LVT monitor"`. If not present, it defaults to the identifier.

The field descriptions declare the individual bitfields which make up the data type.

The order in which these are given is determined by the `lsbfirst` or `msbfirst` declaration. Data structures in memory are sometimes listed in data books as a sequence of words, each of which is formatted into bit fields. In order for authors of device specifications to easily translate these descriptions into Mackerel, one can specify the "word size" used in the description along with the order in which bit fields within a word are listed. For example, "`msbfirst(32)`" means that the bit fields are to be grouped into words of 32 bits each, and within each word the fields are listed most-significant-bit (i.e. the field containing bit 31) first. On an Intel architecture machine, this will cause the resulting C declaration to be derived by taking the first set of fields whose width adds up to 32, reversing their order in the struct declaration, then the next group, etc.

Specifying a word size of 0 will cause the entire set of fields to be reordered (if necessary) in one go. If no bit order and word size are specified, the default is that of the device itself, with a word size of 0.

fieldname is an identifier for the bit field. It can be given as an underscore (_) to indicate "don't care" or "reserved" fields, in which case the field is assumed to have an attributed of `rsvd` (see below), and no description is needed.

width is an integer giving the width of the field in bits.

fieldtype is an optional identifier giving the type of this field. This must be the identifier of a `constants` declaration somewhere in the enclosing `device` declaration, for example `type(vdm)`.

attr is an optional field attribute. See section 3.9 for more information. The only field attributes allowed for data types are `rsvd`, `mbz`, `mb1`, or `rw`. "Don't care" fields default to `rsvd`; others default to `rw`.

description is an optional string in double quotes, giving a description of the field, for example `"Command header"`. If not present, it defaults to the identifier.

Here is a complete example of a `regtype` declaration, taken from the a SATA AHCI port definition:

```
datatype cls msbfirst(32) "Command list structure" {
    prdtl 16 "Physical region descriptor table length";
    pmp   4  "Port multiplier port";
    _     1;
    c     1 "Clear busy upon R_OK";
    b     1 "BIST";
    r     1 "Reset";
    p     1 "Prefetchable";
    w     1 "Write";
    a     1 "ATAPI";
    cfl   5 "Command FIS length";
    prdbc 32 "Physical region descriptor byte count";
    ctba  32 "Command table descriptor base addr";
    ctbau 32 "Command table descriptor base addr upper";
    _     32;
    _     32;
    _     32;
    _     32;
};
```

3.9 Field and register attributes

This section lists the various attributes that can be applied to register fields or entire registers. Some of these values can only be applied to register fields, and not to entire registers. Some of these attributes are functionally equivalent from the point of view of client code generated by Mackerel, but the aim is to facilitate typing in information from data sheets.

rw Read/write. The value is readable and writable. This is the default.

ro Read-only. The value is readable, but not writeable. No code will be generated to handle writes to this value.

wo Write-only. The value is writable, but not readable. No code will be generated to handle reads from this value, but generated code may keep a "shadow" copy of the last value written to aid in debugging.

rc Read to clear. The value is readable, but not writeable, and a read operation will clear the value (this is sometimes seen with statistical count registers, for example). Driver code generated for registers will behave as if the register was **ro**, so care must be taken if extra reads are inserted. In particular, remember that printing the value of the register in debugging code will clear it.

rwzc Read/write zero to clear. The value is readable. A write of 0 clears (sets to 0) the corresponding bit, and a write of 1 has no effect.

rw1c Read/write one to clear. The value is readable. A write of 1 clears (sets to 0) the corresponding bit, and a write of 0 has no effect.

rwc Read/write clear. A synonmym for **rw1c**.

ros Read-only sticky. Value is readable and not writeable. In Intel nomenclature, the bits can only be reset with a power cycle; in Mackerel this attribute is simply a synonym for `ro`.

rw1cs Read and write one to clear and sticky. In Intel nomenclature, this is equivalent to `rwc` but bits can only be reset on a power cycle.

rwcs Read, write to clear and sticky. A synonym for **rw1cs**.

rwo Read/write once. Can be written to precisely once after reset, and thereafter is read-only.

rws Read/write and sticky. Software can read and write to this bit, but the bit can only be reset after a power cycle.

rsvd Reserved. Mackerel generates code to allow clients to read reserved bits, but when writing a register value, code generated by Mackerel will ensure the bit values are preserved, if necessary by performing a read before the write.

mbz Must be zero. Mackerel will generate no code to read the value, but will ensure that it is always written as 0.

mb1 Must be one. Mackerel will generate no code to read the value, but will ensure that it is always written as 1.

Register attributes can be `rw`, `ro`, or `wo`, and provide a default attribute for each field of the register. If no attributes at all are specified, `rw` is assumed, except in the case of an anonymous register field (denoted by an underscore "_"), which defaults to `rsvd` regardless of the register attribute. The motivation for this design choice is that unused fields of registers are frequently "reserved" rather than "don't care". The default behaviour will therefore result in potential inefficiency (due to reading the value before writing it), but never incorrect behaviour (due to writing the wrong values to a reserved field).

Chapter 4

C mapping for device drivers

For each device specification, Mackerel generates a single C header file giving all the necessary definitions and declarations. While developers are not expected to look at this file in normal use, it is designed to be fairly readable to curious humans, and is worth examining to understand what Mackerel is generating (and to find bugs in Mackerel).

The file for a device called, say, `apic` is conventionally named `apic_dev.h`, and in Barrelfish is usually found in the `dev` subdirectory of the main include directory; thus a C file would include it with:

```
#include <dev/apic_dev.h>
```

This chapter will describe the contents of this header file.

4.1 Preamble

The header file is protected by the customary macro to ensure that it is only included once - for example, for a device called `DEV` this is "`__DEV_H`".

The file includes only one other header file: `mackerel.h`. This latter header file includes declarations for the low-level read and write functions that Mackerel code requires, and some other preprocessor macros to ensure correct C code generation.

In the rest of this chapter, we'll use "`DN_`" as the device prefix that cannot be redefined in this manner, and "`DP_`" as device prefix that can.

4.2 Device-level declarations

For the device, Mackerel generates:

1. a C structure type to represent what it knows about the device, called `DN_t`. To access a device from C via Mackerel, you first need to declare one of these, and then call a function (see below) to initialize it.

   The struct contains fields which correspond to each of the arguments which define the addresses of the device, plus a "shadow" copy of every register on the device which has at least one write-only field - this is used to keep track of the last value written to the register.

2. The initialization function `DN_initialize`. For a device type with an argment of "`io base`", the initialization function looks like:

   ```
   void DN_initialize( DN_t *_dev, mackerel_io_t base );
   ```

   The idiom for accessing a device at IO address `ia` would be:

   ```
   DN_t dev;
   DN_initialize( &dev, ia );
   ...
   code to access registers via dev
   ```

3. An enumeration declaration `enum DN_initials`, with a field `DN_REG_initial` for each register `REG` defined in the device. Currently, all these enumeration fields are set to `0x00`, and are used inside the generated initialization function to initialize any shadow copies of registers in the device struct.

4. An `snprintf`-like function to pretty-print *all* the register contents of the device, with prototype:

   ```
   int DN_pr(char *s, size_t sz, DN_t *v);
   ```

   For a complex device, this is a lot of information, and may require a large buffer to fully capture. This is also potentially a large function to inline. It will result in reads from all readable device registers, including "read to clear" (**rc**) registers and fields. For write-only registers, this function will print the shadow copy of the register maintained in the device structure, which holds the last value written to the register (or 0 if the register has not been written since the initialization function was called). The output identifies which values are "real" and which are shadow copies, includes descriptions of all register fields, and translates symbolic values into their names and descriptions.

4.3 Constants

For a constants declaration of type `CNSTS`, having fields `FIELD1` and `FIELD2`, Mackerel will generate the following:

1. A type definition which defines type `DP_CNSTS_t` to be an unsigned integer type (e.g. `uint8_t` or `uint64_t`). The type is the smallest such type large enough to hold the largest value in the constants declaration, or (if the width is explicitly given) the smallest such type wider than the given width.

2. A set of CPP macro definitions `DP_FIELD1` and `DP_FIELD2`, each of which expands to a C expression consisting of the field value cast to type `DP_CNSTS_t`.

   If the field value is specified as `1s`, the macro will expand to the C value `-1LL` cast to type `DP_CNSTS_t`.

   Note that the field names are not prefixed by the constants name, only the device prefix.

   The motivation for using CPP macros (rather than C enumerations, as in an earlier version of Mackerel) is that `enum` types are only the size of C `int`s, whereas Mackerel can specify values in `constants` declarations which are larger than this (such as 64-bit values).

   The motivation for not using constant unsigned integer declarations is that macros allow us to only generate a header file, without cluttering up the data segment with multiple copies of constants if the generated header file is included more than once.

3. A function which takes an argument of type DP_CNSTS_t and returns a pointer to a string containing the description of the field value, or NULL if the argument does not correspond to a field value:

```
char *DP_CNSTS_describe(DP_CNSTS_t e);
```

   Note that this function can also simply be used as a test of whether the value is valid.

4. An snprintf-like function to pretty-print values of type DP_CNSTS_t, with prototype:

```
int DP_CNSTS_prtval(char *s, size_t sz, DP_CNSTS_t e);
```

4.4 Register types

.

For a regtype declaration REGTYPE, Mackerel will generate the following:

1. A type definition DP_REGTYPE_t, which is an unsigned integer of the same size as the register.
2. A CPP macro DP_REGTYPE_default, which is an integer literal giving the "default" value for the register type. Note that since this is a macro, the prefix of the name cannot be overridden using the trick mentioned above.
3. An snprintf-like function to pretty-print values of the register type, with prototype:

```
int DP_REGTYPE_prtval(char *s, size_t sz, DP_REGTYPE_t v);
```

4. For every field FLD (of type FLDTYPE_t) of the register type, a function to insert a value into that field of a value of type DP_REGTYPE_t:

```
DP_REGTYPE_t DP_REGTYPE_FLD_insert(DP_REGTYPE_t r, FLDTYPE_t val );
```

   Note that this insertion is *not* in place, in contrast to the function generated for datatype declarations (see below). Instead, this it returns a new value of the register type with the specified field modified, and so is intended to be used in a "functional" style.

5. For every field FLD (of type FLDTYPE_t) of the register type, a function to extract a single field from a value of type DP_REGTYPE_t:

```
FLDTYPE_t DP_REGTYPE_FLD_extract(DP_REGTYPE_t r);
```

4.5 Data types

.

For a datatype declaration DATATYPE, Mackerel will generate the following:

1. A type definition DP_DATATYPE_t, which is of type uint8_t *. This is the canonical way to refer to values of the datatype in memory.
2. A static const size_t DP_DATATYPE_size, which is the size in bytes of the data type. This should be used in place of sizeof(DP_DATATYPE_t), which will the wrong result in general.

3. A type definition DP_DATATYPE_array_t, which is of type uint8_t[sz], where sz is the size in bytes of the datatype. This should be used sparingly, for example to allocate a value of the datatype on the stack.

4. An snprintf-like function to pretty-print values of the datatype, with prototype:

```
int DP_DATATYPE_prtval(char *s, size_t sz, DP_DATATYPE_t v);
```

5. For every field FLD (of type FLDTYPE_t) of the datatype, a function to insert a value into that field of a value of type DP_DATATYPE_t:

```
void DP_DATATYPE_FLD_insert(DP_DATATYPE_t r, FLDTYPE_t val );
```

   Note that this insertion is *in place*, in contrast to the "functional" style used for register types.

6. For every field FLD (of type FLDTYPE_t) of the datatype, a function to extract a single field from a value of type DP_DATATYPE_t:

```
FLDTYPE_t DP_DATATYPE_FLD_extract(DP_DATATYPE_t r);
```

4.6 Registers

For a register REG that defines its own type, Mackerel defines a register type with the same name and generates the declarations defined in section 4.4.

For each register REG with type REGTYPE, Mackerel will generate some of the following:

1. If the register has fields which are readable, a function to read the contents of the register:

```
DP_REGTYPE_t DP_REG_rd( DN_t *dev );
```

2. If the register address is not given as noaddr, a function to read the raw contents of the register:

```
DP_REGTYPE_t DP_REG_rawrd( DN_t *dev );
```

   On the other hand, if the register address is given as noaddr, the developer is required to supply this function herself – Mackerel-generated code will use it to access the register.

3. If the register has fields which are writeable, a function to write the contents of the register:

```
void DP_REG_wr( DN_t *dev, DP_REGTYPE_t val );
```

   Note that this is *not* a simple write of the value: any fields which must be 1 or 0 are set accordingly, and reserved fields are read from the register before writing back the value. Therefore, this function will only write the bit fields of val which are meaningful to write according to the register specification. However, this function will not perform a read from the register unless the register contains reserved fields.

4. If the register address is not given as noaddr, a function to write raw values to the register:

```
DP_REGTYPE_t DP_REG_rawwr( DN_t *dev, DP_REGTYPE_t val );
```

   This function is potentially dangerous, since it makes no guarantees that certain fields of the register are set to zero or one, or preserves the contents of reserved fields. Use with caution.

   On the other hand, if the register address is given as noaddr, the developer is required to supply this function herself – Mackerel-generated code will use it to access the register. Again, in

this case, the function should not try to preserve fields, since Mackerel will make sure that the "sanitized" register and field writing functions will do this correctly.

5. For every readable field FLD (of type FLDTYPE) of the register, a function to read just that single field from the register:

   ```
   FLDTYPE DP_REG_FLD_rdf( DN_t *dev );
   ```

6. For every writeable field FLD of the register, a function to write just that single field to the register:

   ```
   void DP_REG_FLD_wrf( DN_t *dev, FLDTYPE val );
   ```

   These functions should be used with care. It is of course impossible to write only one field, hence a complete write to the register will occur. Furthermore, the values of the other read/writeable or reserved register fields will be determined first by doing a hidden read from the register. Any write-only fields will then be initialized from the shadow copy. Any "must be zero" or "must be 1" fields will be set to their respective values.

7. For every write-only field FLD (of type FLDTYPE) of the register, a function to read just that single field from the shadow copy of the register contents:

   ```
   FLDTYPE DP_REG_FLD_rd_shadow( DN_t *dev );
   ```

8. An snprintf-like function to pretty-print the current value of the register, or its shadow copy if the register is not readable, with prototype:

   ```
   int DP_REG_pr( char *s, size_t sz, DN_t *dev );
   ```

4.7 Register Arrays

For a register array RARR that defines its own type, Mackerel defines a register type with the same name and generates the declarations defined in section 4.4.

For each register array RARR with type REGTYPE, Mackerel will generate some of the following:

1. A CPP macro DP_DATATYPE_size, which is an integer literal cast to a size_t giving the length in bytes of the data type.

2. If the register array has fields which are readable, a function to read the contents of a register:

   ```
   DP_REGTYPE_t DP_REG_rd( DN_t *dev, int i );
   ```

3. If the register array has fields which are writeable, a function to write the contents of a register:

   ```
   void DP_REG_wr( DN_t *dev, int i, DP_REGTYPE_t val );
   ```

   Note that this is *not* a simple write of the value: any fields which must be 1 or 0 are set accordingly, and reserved fields are read from the register before writing back the value. Therefore, this function will only write the bit fields of val which are meaningful to write according to the register specification. However, this function will not perform a read from the register unless the register contains reserved fields.

4. For every readable field FLD (of type FLDTYPE) of the register array, a function to read just that single field from a register:

   ```
   FLDTYPE DP_REG_FLD_rdf( DN_t *dev, int i );
   ```

5. For every writeable field FLD of the register array, a function to write just that single field to a register:

```
void DP_REG_FLD_wrf( DN_t *dev, int i, FLDTYPE val );
```

   These functions should be used with care. It is of course impossible to write only one field, hence a complete write to the register will occur. Furthermore, the values of the other read/writeable or reserved register fields will be determined first by doing a hidden read from the register. Any write-only fields will then be initialized from the shadow copy. Any "must be zero" or "must be 1" fields will be set to their respective values.

6. For every write-only field FLD (of type FLDTYPE) of the register array, a function to read just that single field from the shadow copy of a register's contents:

```
FLDTYPE DP_REG_FLD_rd_shadow( DN_t *dev );
```

7. An snprintf-like function to pretty-print the current value of one register in the array, or its shadow copy if the field is write-only, with prototype:

```
int DP_REG_pri( char *s, size_t sz, DN_t *dev, int i );
```

8. An snprintf-like function to pretty-print the current value of all registers in the array, or their shadow copies if the field is write-only, with prototype:

```
int DP_REG_pr( char *s, size_t sz, DN_t *dev );
```

4.8 Address spaces

It is the responsibility of the programmer to supply functions to access registers in user-defined address spaces (though it's common to call Mackerel-generated code in turn to do this). The register access code generated by Mackerel for a 32-bit wide register declared in an address space SPACE will call the following function for a read:

```
uint32_t DP_SPACE_read_32( DN_t *dev, size_t offset );
```

– and the following function for a write:

```
void DP_SPACE_write_32( DN_t *dev, size_t offset, uint32_t value );
```

These functions must be supplied by the programmer in a file named DP_spaces.h, which is included by the generated header file *if and only if* the device specification defines new address spaces. Note that only one header file is required, rather than one for each user-defined address space.

Since the device is passed as an argument, the functions can access any device fields (including device arguments and shadow copies of write-only registers)

They are not declared anywhere in the generated header file, which means that not all the read/write functions for an address space need to be supplied, only the ones actually used. Furthermore, these can be declared to be `static inline`, or `extern`, etc. according to taste.

4.9 Overall header file structure

The generated header file is not too difficult to read, and doing so can be useful for debugging purposes. The declarations emitted by Mackerel occur in the file in the following order (to minimise the need for forward declarations):

1. Preamble, naturally.
2. Type, print, and check declarations for each “constants” clause.
3. Type, union, and print declarations for each register type (including implicit types).
4. The device structure, and its initialization function.
5. Read, write, and print functions for each register
6. The device print function.

In addition, comments in the generated file are intended to help you find the relevant declarations quickly if you need to.

Chapter 5

Bitfield C mapping for device drivers

Important

This appendix contains the deprecated C mapping which uses bitfield structures to represent registers. This back end should not be used, since it is not portable across architectures and does not guarantee that register accesses will be of the same size as the register, which breaks some architectures (such as Rock Creek). It is included here to help understand legacy code. Such code should be converted as soon as possible.

For each device specification, Mackerel generates a single C header file giving all the necessary definitions and declarations. While developers are not expected to look at this file in normal use, it is designed to be fairly readable to curious humans, and is worth examining to understand what Mackerel is generating (and to find bugs in Mackerel).

This chapter will describe the contents of this header file.

5.1 Preamble

The header file is protected by the customary macro to ensure that it is only included once - for example, for a device called `DEV` this is "`__DEV_H`".

The file includes only one other header file: `mackerel.h`. This latter header file includes declarations for the low-level read and write functions that Mackerel code requires, and some other preprocessor macros to ensure correct C code generation.

Mackerel prefixes all declarations in the file with the name of the file plus an underscore ("`DEV_`").

In the rest of this chapter, we'll use "`DN_`" as the device prefix that cannot be redefined in this manner, and "`DP_`" as device prefix that can.

5.2 Device-level declarations

Mackerel provides a C structure type to represent what it knows about the device, called `DN_t`. To access a device from C via Mackerel, you first need to declare one of these, and the call a function to initialize

it.

The initialize function is called `DN_initialize`. For a device type with an argment of "`io base`", the initialization function looks like:

```
void DN_initialize( DN_t *_dev, mackerel_io_t base );
```

The idiom for accessing a device at IO address `ia` would be:

```
DN_t dev;
DN_initialize( &dev, ia );
...
code to access registers via dev
```

In addition, Mackerel generates an `snprintf`-like function to pretty-print *all* the register contents of the device, with prototype:

```
int DN_pr(char *s, size_t sz, DP_CNSTS_t v);
```

For a complex device, this is a lot of information, and may require a large buffer to fully capture. This is also potentially a large function to inline. It will result in reads from all readable device registers, including "read to clear" (**rc**) registers and fields. For write-only registers, this function will print the shadow copy of the register maintained in the device structure, which holds the last value written to the register (or 0 if the register has not been written since the initialization function was called). The output identifies which values are "real" and which are shadow copies, includes descriptions of all register fields, and translates symbolic values into their names and descriptions.

5.3 Constants

For a constants declaration of type `CNSTS`, having fields `FIELD1` and `FIELD2`, Mackerel will generate the following:

1. An enumeration type `DP_CNSTS_t` with fields `DP_FIELD1` and `DP_FIELD2`. Note that the field names are not prefixed by the enumeration name, only the device prefix.
2. An `snprintf`-like function to pretty-print values of the enumeration, with prototype:

   ```
   int DP_CNSTS_prt(char *s, size_t sz, DP_CNSTS_t e);
   ```

3. A function which returns 1 if an enumeration value is valid, 0 if otherwise:

   ```
   int DP_CNSTS_chk(DP_CNSTS_t e);
   ```

5.4 Register types

.

For a `regtype` declaration `REGTYPE`, Mackerel will generate the following:

1. A bitfield structure type `DP_REGTYPE_t`, whose field names are those of the fields specified in the Mackerel file, with no prefix, and whose physical layout corresponds to the hardware register type.

2. A union type DP_REGTYPE_un, whose members are the bitfield struct above (as "val") and the smallest enclosing unsigned integer type (as "raw"),.

3. An snprintf-like function to pretty-print values of the register type, with prototype:

```
int DP_REGTYPE_prtval(char *s, size_t sz, DP_CNSTS_t v);
```

This functionality is being replaced by one based on shifts and masks, rather than using bitfields, which have portability issues. For a regtype declaration REGTYPE, Mackerel will soon generate the following:

1. A type definition DP_REGTYPE_t which is an unsigned integer of the same size as the register.

2. For every field FLD (of type FLDTYPE) of the register type, a function to insert a value into that field of a value of type DP_REGTYPE_t:

```
DP_REGTYPE_t DP_REGTYPE_FLD_ins(DP_REGTYPE_t r, FLD_TYPE val );
```

3. For every field FLD (of type FLDTYPE) of the register type, a function to extract a single field from a value of type DP_REGTYPE_t:

```
FLDTYPE DP_REGTYPE_FLD_ext(DP_REGTYPE_t r);
```

5.5 Data types

.

For a datatype declaration DATATYPE, Mackerel will generate the following:

1. A bitfield structure type DP_DATATYPE_t, whose field names are those of the fields specified in the Mackerel file, with no prefix, and whose physical layout corresponds to the hardware data type.

2. An snprintf-like function to pretty-print values of the data type, with prototype:

```
int DP_DATATYPE_prtval(char *s, size_t sz, DP_CNSTS_t v);
```

5.6 Registers

For a register REG that defines its own type, Mackerel defines a register type with the same name and generates the declarations defined in section 5.4.

For each register REG with type REGTYPE, Mackerel will generate some of the following:

1. If the register has fields which are readable, a function to read a "raw" (integer) value from the register:

```
uintx_t DP_REG_rd_raw( DN_t *dev );
```

   – where uint$x$_t is a standard C unsigned integer type with a fixed size, such as uint32_t.,

2. If the register has fields which are readable, a function to read a bitfield value from the register:

```
DP_REGTYPE_t DP_REG_rd( DN_t *dev );
```

3. If the register has fields which are not readable, a function to read a bitfield value from a saved copy of the last value written to the register using this device struct:

```
DP_REGTYPE_t DP_REG_rd_shadow( DN_t *dev );
```

4. If the register has fields which are writeable, a function to write a "raw" (integer) value to the register:

   ```
   void DP_REG_wr_raw( DN_t *dev, uintx_t val );
   ```

   – where `uintx_t` is smallest suitable standard C unsigned integer type with a fixed size, such as `uint32_t`.

5. If the register has fields which are writeable, a function to write a bitfield value to the register:

   ```
   void DP_REG_wr( DN_t *dev, DP_REGTYPE val );
   ```

   Unlike the raw write function, this will (only if necessary) perform a read first from the register to ensure that "reserved" fields will be written back with their correct values. It will also force any "must be zero" or "must be 1" fields to be their respective values.

6. For every writeable field `FLD` of the register, a function to write just that single field to the register:

   ```
   void DP_REG_FLD_wrf( DN_t *dev, uintx_t val );
   ```

   – where `uintx_t` is smallest suitable standard C unsigned integer type with a fixed size, such as `uint8_t`.

   These functions should be used with care. It is of course impossible to write only one field, hence a complete write to the register will occur. Furthermore, the values of the other read/writeable or reserved register fields will be determined first by doing a hidden read from the register. Any write-only fields will then be initialized from the shadow copy. Any "must be zero" or "must be 1" fields will be set to their respective values.

7. An `snprintf`-like function to pretty-print the current value of the register, or its shadow copy if the register is not readable, with prototype:

   ```
   int DP_REG_pr( char *s, size_t sz, DN_t *dev );
   ```

5.7 Register Arrays

For a register array `RARR` that defines its own type, Mackerel defines a register type with the same name and generates the declarations defined in section 5.4.

For each register array `RARR` with type `REGTYPE`, Mackerel will generate some of the following:

1. A static constant giving the length of the array (42 in this example):

   ```
   static const int DP_RARR_length = 0x26;
   ```

2. If the registers have fields which are readable, a function to read a "raw" (integer) value from a register:

   ```
   uintx_t DP_RARR_rd_raw( DN_t *dev, int i );
   ```

   – where `uintx_t` is a standard C unsigned integer type with a fixed size, such as `uint32_t`, and `i` is an index into the array.

3. If the registers have fields which are readable, a function to read a bitfield value from a register:

   ```
   DP_REGTYPE_t DP_RARR_rd( DN_t *dev, int i );
   ```

4. If the registers have fields which are not readable, a function to read a bitfield value from a saved copy of the last value written to each register using this device struct:

   ```
   DP_REGTYPE_t DP_RARR_rd_shadow( DN_t *dev, int i );
   ```

5. If the registers have fields which are writeable, a function to write a "raw" (integer) value to a register:

   ```
   void DP_RARRG_wr_raw( DN_t *dev, int i, uintx_t val );
   ```

   – where `uint`x`_t` is smallest suitable standard C unsigned integer type with a fixed size, such as `uint32_t`.

6. If the registers have fields which are writeable, a function to write a bitfield value to a register:

   ```
   void DP_RARR_wr( DN_t *dev, int i, DP_REGTYPE val );
   ```

   – where `uint`x`_t` is smallest suitable standard C unsigned integer type with a fixed size, such as `uint32_t`.

   Unlike the raw write function, this will (only if necessary) perform a read first from the register to ensure that "reserved" fields will be written back with their correct values. It will also force any "must be zero" or "must be 1" fields to be their respective values.

7. An `snprintf`-like function to pretty-print the current value of one element of the array, or its shadow copy if the register is not readable, with prototype:

   ```
   int DP_RARR_pri( char *s, size_t sz, DN_t *dev, int i );
   ```

8. An `snprintf`-like function to pretty-print the entire register array, or its shadow copies if the registers in the array are not readable, with prototype:

   ```
   int DP_RARR_pr( char *s, size_t sz, DN_t *dev );
   ```

5.8 Address spaces

It is the responsibility of the programmer to supply functions to access registers in user-defined address spaces (though it's common to call Mackerel-generated code in turn to do this). The register access code generated by Mackerel for a 32-bit wide register declared in an address space `SPACE` will call the following function for a read:

```
uint32_t DP_SPACE_read_32( DN_t *dev, size_t offset );
```

– and the following function for a write:

```
void DP_SPACE_write_32( DN_t *dev, size_t offset, uint32_t value );
```

These functions must be supplied by the programmer in a file named `DP_spaces.h`, which is included by the generated header file *if and only if* the device specification defines new address spaces. Note that only one header file is required, rather than one for each user-defined address space.

Since the device is passed as an argument, the functions can access any device fields (including device arguments and shadow copies of write-only registers)

They are not declared anywhere in the generated header file, which means that not all the read/write functions for an address space need to be supplied, only the ones actually used. Furthermore, these can be declared to be `static inline`, or `extern`, etc. according to taste.

5.9 Overall header file structure

The generated header file is not too difficult to read, and doing so can be useful for debugging purposes. The declarations emitted by Mackerel occur in the file in the following order (to minimise the need for forward declarations):

1. Preamble, naturally.
2. Type, print, and check declarations for each “constants” clause.
3. Type, union, and print declarations for each register type (including implicit types).
4. The device structure, and its initialization function.
5. Read, write, and print functions for each register
6. The device print function.

In addition, comments in the generated file are intended to help you find the relevant declarations quickly if you need to.

Chapter 6

Migrating from the bit-field based version of Mackerel

Previous versions of Mackerel generated a C API by defining packed C bitfield structs for register types. This version of Mackerel avoids this in favor of masks and shifts, hidden behind inline functions. This chapter gives some hints for migrating code written using the old Mackerel to the new version.

- The initialize function for a device structure remains the same.
- `DEV_REG_wr_raw()` and `DEV_REG_rd_raw()` functions should be replaced with `DEV_REG_wr()` and `DEV_REG_rd()` with the same arguments.
- The following code to set individual register fields:

```
//deassert PHY_RESET
{
    e1000_ctrl_t c = e1000_ctrl_rd(d);
    c.phy_rst = 0;
    e1000_ctrl_wr(d, c);
    e1000_status_t s = e1000_status_rd(d);
    s.phyra = 0;
    e1000_status_wr(d, s);
}
```

  – must be replaced with:

```
//deassert PHY_RESET
e1000_ctrl_phy_rst_wrf(d, 0);
e1000_status_phyra_wrf(d, 0);
```

- The following code to read individual register fields:

```
speed = e1000_status_rd(d).speed;
```

  – must be replaced by:

```
speed = e1000_status_speed_rdf(d);
```

- The following code to access a field from a register:

```
// test LAN ID to see if we need to modify the MAC from EEPROM
```

```
{
    e1000_status_t s = e1000_status_rd(d);
    if (s.lan_id == e1000_lan_b) {
        mac_word2 ^= e1000_lan_b_mask;
    }
}
```

– must be replaced by:

```
// test LAN ID to see if we need to modify the MAC from EEPROM
{
    e1000_status_t s = e1000_status_rd(d);
    if (e1000_status_lan_id_extract(s) == e1000_lan_b) {
        mac_word2 ^= e1000_lan_b_mask;
    }
}
```

- The following idiom for setting fields using struct literals:

```
e1000_rxdctl_wr(d, 0, (e1000_rxdctl_t){ .gran=1, .wthresh=1 } );
```

– should be replaced by:

```
e1000_rxdctl_t tmp = e1000_rxdctl_default;
tmp = e1000_rxdctl_gran_insert(tmp,1);
tmp = e1000_rxdctl_wthresh_insert(tmp,1);
e1000_rxdctl_wr(d, 0, tmp);
```

Chapter 7

Source code structure

The Mackerel source directory contains the files listed below.

Attr.hs Dealing with register field attributes.

BitFieldDriver.hs Old code generator for device drivers using bitfields, deprecated.

CAbsSyntax.hs Contains combinators for rendering C code.

CSyntax.hs Old-style C combinators, deprecated.

Checks.hs Compile-time checks.

Dev.hs Overall device representation.

Fields.hs Representing register fields.

MackerelParser.hs ParSec parser for language.

Main.hs Top level file, which is used to run Mackerel.

Poly.hs Polynomial arithmetic for compile-time expression reduction (currently unused)

RegisterTable.hs Representing registers and register arrays.

ShiftDriver.hs Code generator for device drivers using shifts and masks.

Space.hs Representing address spaces.

TypeName.hs Data type for scoping type names when importing.

TypeTable.hs Representing register, data, and constants types.

Chapter 8

Quick reference

In the following quick reference, the following identifiers are used as examples, assuming a Mackerel file named `dev1.dev`:

dev1 A device definition.

const1 A constants definition.

cval1 One value of the constants definition.

rtype2 A register type definition.

field2 A field in the above register type.

ftype2 The type of this field.

dtype3 A data type definition.

field3 A field in the above data type.

ftype3 The type of this field.

reg4 A register definition with its own type.

rtype4 Type of the above register.

field4 A field in the above register.

ftype4 The type of this field.

array5 A register array.

field5 A field in the elements of the above array.

ftype5 The type of this field.

Quick list of Mackerel C declarations:

C declaration	Description
Device declarations	
`struct dev1_t`	Device structure
`typedef dev1_t`	Device type defintion (as above)
`enum dev1_initials`	Initial register values
`void dev1_initialize( dev1_t *dev, ... )`	Initialization
`int dev1_pr( char *s, size_t sz, device_t *v )`	Print
Constants declarations	
`typedef dev1_const1_t`	Integer type
`#define dev1_cval1`	Macro giving integer value
`char *dev1_const1_describe( dev1_const1_t e )`	Description
`int dev1_const1_prtval( char *s, size_t sz, dev1_const1_t e )`	Print
`int dev1_const1_chk( dev1_const1_t e)`	Check value
Register type declarations	
`typedef dev1_rtype2_t`	Type defintion
`#define dev1_rtype2_default`	Default value
`int dev1_rtype2_prtval( char *s, size_t sz, dev1_rtype2_t v )`	Print
`dev1_rtype2_t dev1_rtype2_field2_insert( dev1_rtype2_t r, ftype2_t val )`	Insert a field value
`ftype2_t dev1_rtype2_field2_extract( dev1_rtype2_t r )`	Extract field value
Datatype declarations	
`typedef dev1_dtype3_t`	Pointer to datatype
`const size_t dev1_dtype3_size`	Size
`typedef dev1_dtype3_array_t`	Byte array of the same size
`int dev1_dtype3_prtval( char *s, size t sz, dev1_dtype3_t v )`	Print
`void dev1_dtype3_field3_insert( dev1_dtype3_t r, ftype3_t val )`	Insert field
`ftype3_t dev1_dtype3_field3_extract( dev1_dtype3_t r )`	Extract field
Register declarations	
`dev1_rtype4_t dev1_reg4_rd( dev1_t *dev )`	Read register
`dev1_rtype4_t dev1_reg4_rawrd( dev1_t *dev )`	Raw read
`void dev1_reg4_wr( dev1_t *dev, dev1_rtype4_t val )`	Write register
`void dev1_reg4_rawwr( dev1_t *dev, dev1_rtype4_t val )`	Raw write
`ftype4_t dev1_reg4_field4_rdf( dev1_t *dev )`	Read field
`void dev1_reg4_field4_wrf( dev1_t *dev, ftype4 val )`	Write field
`ftype4_t dev1_reg4_field4_rd_shaddow( dev1_t *dev )`	Read shadow
`int dev1_reg4_pr( char *s, size_t sz, dev1_t *dev )`	Print
Register array declarations	
`const int dev1_array5_length`	Num. registers
`dev1_rtype5_t dev1_reg5_rd( dev1_t *dev, int i )`	Read register
`dev1_rtype5_t dev1_reg5_rawrd( dev1_t *dev, int i )`	Raw read
`void dev1_reg5_wr( dev1_t *dev, int i, dev1_rtype5_t val )`	Write register
`void dev1_reg5_rawwr( dev1_t *dev, int i, dev1_rtype5_t val )`	Raw write
`ftype5_t dev1_reg5_field5_rdf( dev1_t *dev, int i )`	Read field
`void dev1_reg5_field5_wrf( dev1_t *dev, int i, ftype5 val )`	Write field
`ftype5_t dev1_reg5_field5_rd_shaddow( dev1_t *dev, int i )`	Read shadow
`int dev1_reg5_pri( char *s, size_t sz, dev1_t *dev, int i )`	Print
`int dev1_reg5_pr( char *s, size_t sz, dev1_t *dev )`	Print array
Space access functions	
`uint`*x*`_t dev1_space6_read_`*x*`( dev1_t *dev, size_t offset )`	Read a value
`void dev1_space6_write`*x*`( dev1_t *dev, size_t offset, uint`*x*`_t value )`	Write a value