


2005

## Formalization Of Input And Output In Modern Operating Systems: The Hadley Model

Matthew Gerber  
*University of Central Florida*

 Part of the [Computer Sciences Commons](#), and the [Engineering Commons](#)  
Find similar works at: <https://stars.library.ucf.edu/etd>  
University of Central Florida Libraries <http://library.ucf.edu>

This Doctoral Dissertation (Open Access) is brought to you for free and open access by STARS. It has been accepted for inclusion in Electronic Theses and Dissertations, 2004-2019 by an authorized administrator of STARS. For more information, please contact [STARS@ucf.edu](mailto:STARS@ucf.edu).

---

### STARS Citation

Gerber, Matthew, "Formalization Of Input And Output In Modern Operating Systems: The Hadley Model" (2005). *Electronic Theses and Dissertations, 2004-2019*. 324.  
<https://stars.library.ucf.edu/etd/324>

FORMALIZATION OF INPUT AND OUTPUT IN  
MODERN OPERATING SYSTEMS:  
THE HADLEY MODEL

by

MATTHEW BURNETT GERBER  
B.S. University of Central Florida, 1998  
M.S. University of Central Florida, 2000

A dissertation submitted in partial fulfillment of the requirements  
for the degree of Doctor of Philosophy  
in the School of Computer Science  
in the College of Engineering and Computer Science  
at the University of Central Florida  
Orlando, Florida

Spring Term  
2005

Major Professor: John J. Leeson

© 2005 Matthew Burnett Gerber

## **ABSTRACT**

We present the Hadley model, a formal descriptive model of input and output for modern computer operating systems. Our model is intentionally inspired by the Open Systems Interconnection model of networking; I/O as a process is defined as a set of translations between a set of computer-sensible forms, or layers, of information.

To illustrate an initial application domain, we discuss the utility of the Hadley model and a potential associated I/O system as a tool for digital forensic investigators.

To illustrate practical uses of the Hadley model we present the Hadley Specification Language, an essentially functional language designed to allow the translations that comprise I/O to be written in a concise format allowing for relatively easy verifiability.

To further illustrate the utility of the language we present a read/write Microsoft DOS FAT12 and read-only Linux ext2 file system specification written in the new format. We prove the correctness of the read-only side of these descriptions. We present test results from operation of our HSL-driven system both in user mode on stored disk images and as part of a Linux kernel module allowing file systems to be read.

We conclude by discussing future directions for the research.

To Dr. Homer Gerber and Nancy Burnett Gerber, for their love and support.

*Soli Deo Gloria.*

## ACKNOWLEDGMENTS

First and foremost, my thanks above all to the LORD, my God and Savior through his Son, without whom this work and my very life would be less than meaningless.

Dr. John Leeson, my advisor and major professor, guided me through this work at every stage, from beginning to end and as it became something quite different than it was when it began. When I began to seek dissertation topics suitable for the area of digital investigations, the path of research took a wholly unexpected direction, and it was Dr. Leeson's broad knowledge of what could be accomplished that kept the project on track.

Drs. Kien Hua, David Workman, and K. Michael Reynolds, my committee, kept the project focused, and with their diverse expertise provided invaluable insight and direction.

Drs. Amar Mukherjee and Randall Shumaker provided significant insights in the early stages of the project that helped guide it into a productive direction.

My family has been a source of support without which this project could never have happened. Dr. Homer Gerber, my father, gave me invaluable encouragement, inspiration, and someone to discuss obscure results with at all hours. My mother, Mrs. Nancy Gerber, cannot be praised enough for her support and encouragement. My grandparents, Harley and Cora Finley Burnett and Victor and Gertrude Gerber, did not live to see the end of this work with earthly eyes, though Grandma Cora and Grandpa Vic knew it had begun. My memories of them have continually helped to encourage and drive this project, and it is my prayer that they are looking upon it now, at its completion. Dr. John Burnett, a cousin and close friend, has been an encouragement and inspiration.

The Reverend Dr. Jim Henry, my pastor, has been a source of encouragement and inspiration throughout my life, and particularly in the last, critical months of this project.

My academic career began in significant part through the help of two teachers to whom I owe more than I can say. Mrs. Marie Hamrick and Dr. Douglas Brumbaugh guided me through the transition to college, and through my first years of it, with advice, insight, warmth and, at times, much-needed good humor.

The encouragement and support of my friends has been an uplift and a delight. Jeremy, Braden, John, Keith, Jon, Russ, Chris, Jon, Eric, Amy, Jesse, Phil, Larry, Chris, James, Mason, Eric, Carol, Kate, Chris, Travis, and many others helped to make it possible for me to come this far.

Finally, special thanks to Dr. Archibald MacPherson, the Reverend Dr. Henry Parker, Mrs. Rita Mann and Mrs. June Biggs for cherished memories and special contributions to my life.

## TABLE OF CONTENTS

LIST OF FIGURES .....	xiv
LIST OF TABLES .....	xv
LIST OF ACRONYMS .....	xvi
CHAPTER ONE: INTRODUCTION .....	1
CHAPTER TWO: LITERATURE REVIEW .....	7
The OSI Networking Model .....	7
Hardware Standards .....	10
Modeling Languages .....	11
Functional Languages .....	13
Summary .....	14
CHAPTER THREE: RESEARCH DESIGN AND METHODOLOGY .....	15
The Hadley Model .....	15
Definitions .....	15
Layer Models of I/O .....	16
Input and Output in Hardware .....	18
The Layer Model of Hardware .....	19
Uses of the Hardware Model .....	24
Input and Output in Operating Systems .....	25
The Layer Model of Stream Peripherals .....	27
The Layer Model of Random Access Peripherals .....	28
Uses of the Software Model .....	31



The Hadley Specification Language .....	32
Overview .....	32
Spacetypes .....	33
Subspaces .....	33
Addressed References (“Variables”) .....	34
Functions .....	35
Built-In Functions .....	36
Templates .....	38
Lists, Hints and Writing .....	40
Hint Functions .....	40
List Functions .....	41
Expressions .....	42
Working With HSL .....	43
Making HSL Useful .....	43
Writing an HSL Module .....	43
Writing an HSL File System Module .....	44
Using an HSL Module from C .....	45
Initializing HSL .....	46
Registering Template Functions .....	46
Creating the Context .....	47
Calling HSL Functions From C .....	48
Writing Out HSL Results .....	50
Garbage Collection .....	50

Adding a new File System to the Extractor .....	50
Adding a new File System to the VFS Module .....	51
A Note on Concurrency .....	52
Design Notes: The HSL Runtime .....	52
Demonstrating the Correctness of HSL Specifications .....	54
The hadleyfs template .....	54
Demonstrating the Correctness of hadleyfs implementations .....	57
Fundamental Functions .....	57
Directory Functions .....	57
File Functions .....	59
CHAPTER FOUR: FINDINGS .....	60
HSL Test Suite .....	60
Filesystem Extractors .....	60
Universal Filesystem Driver .....	61
Writing Filesystems .....	62
Specification Efficiency .....	62
Performance .....	63
Characteristics of HSL and HSL Demonstrations .....	64
Demonstrations of Correctness .....	64
Major Dependencies .....	64
Definition of FAT12 .....	65
Overview .....	65
Partition Metadata .....	65

Directory Metadata .....	69
The FAT, Clusters and the Data Area .....	72
Demonstration of Correctness for FAT12 .....	73
Definition (FAT12 File ID).....	73
Assertion FAT12-a .....	73
Assertion FAT12-b .....	74
Assertion FAT12-c .....	76
Assertion FAT12-d .....	77
Assertion FAT12-e .....	77
Assertion FAT12-f.....	78
Assertion FAT12-g.....	79
Assertion FAT12-h.....	80
Assertion FAT12-i.....	81
Assertion FAT12-j.....	82
Assertion FAT12-1 .....	83
Assertion FAT12-2b .....	83
Assertion FAT12-k.....	84
Assertion FAT12-m.....	84
Assertion FAT12-n.....	87
Assertion FAT12-o .....	88
Assertion FAT12-p.....	88
Assertion FAT12-3b .....	89
Assertion FAT12-4b.....	91

Assertion FAT12-5b .....	92
Assertion FAT12-6 .....	93
Assertion FAT12-7 .....	94
Assertion FAT12-8 .....	95
Assertion FAT12-9-1 .....	99
Assertion FAT12-9-2 .....	99
Assertion FAT12-9-3 .....	100
Assertion FAT12-9 .....	101
Conclusion .....	103
Definition of ext2fs .....	103
Overview .....	104
Partition Metadata .....	104
File Metadata.....	107
Directories.....	110
Demonstration of Correctness for EXT2FS.....	111
Definition (EXT2FS File ID).....	112
Assertion EXT2FS-a.....	112
Assertion EXT2FS-b .....	113
Assertion EXT2FS-c.....	113
Assertion EXT2FS-d .....	115
Assertion EXT2FS-e.....	115
Assertion EXT2FS-1 .....	117
Assertion EXT2FS-1-1 .....	117

Assertion EXT2FS-f .....	118
Assertion EXT2FS-g .....	118
Assertion EXT2FS-h .....	119
Assertion EXT2FS-i .....	120
Assertion EXT2FS-j .....	122
Assertion EXT2FS-k .....	125
Assertion EXT2FS-l .....	127
Assertion EXT2FS-2b .....	128
Assertion EXT2FS-m .....	128
Assertion EXT2FS-n .....	129
Assertion EXT2FS-3b .....	132
Assertion EXT2FS-4b .....	134
Assertion EXT2FS-5b .....	136
Assertion EXT2FS-6 .....	137
Assertion EXT2FS-7 .....	137
Assertion EXT2FS-8 .....	138
Assertion EXT2FS-9-1 .....	139
Assertion EXT2FS-9-2 .....	139
Assertion EXT2FS-9-3 .....	140
Assertion EXT2FS-9 .....	142
Conclusion .....	143
CHAPTER FIVE: CONCLUSION.....	144
Summary.....	144

Future Directions.....	145
The Model.....	145
The System.....	145
The Tool.....	147
APPENDIX A: HSL SPECIFICATIONS .....	149
APPENDIX B: HADLEY SYSTEM SOURCE CODE .....	165
APPENDIX C: HSL GRAMMAR (YACC FORMAT) .....	211
LIST OF REFERENCES .....	216

## LIST OF FIGURES

Figure 1: The Peripheral Hierarchy .....	6
Figure 2: The OSI Networking Model .....	10
Figure 3: The Hadley Hardware I/O Model .....	25
Figure 4: Software I/O and the Hardware Layers .....	31
Figure 5: The HSL System Design .....	53
Figure 6: Networked File Access.....	146

## LIST OF TABLES

Table 1: ext2fs specification efficiency .....	62
Table 2: FAT12 specification efficiency.....	63
Table 3: Performance results .....	64
Table 4: The FAT12 Boot Sector.....	66
Table 5: The FAT12 BIOS Parameter Block .....	67
Table 6: The FAT12 Extended BIOS Parameter Block.....	68
Table 7: FAT12 Directory Entries .....	70
Table 8: FAT12 Attribute Bytes .....	71
Table 9: FAT12 Reference Values .....	72
Table 10: ext2fs superblock.....	105
Table 11: ext2fs block group .....	106
Table 12: ext2fs inode .....	108
Table 13: ext2fs file formats.....	110
Table 14: ext2fs directory entry.....	111



## LIST OF ACRONYMS

ALGOL	Algorithmic Language
ANSI	American National Standards Institute
ASCII	American Standard Code for Information Interchange
ATA/ATAPI	AT Attachment [with Packet Interface]
BIOS	Basic Input/Output System
CPU	Central Processing Unit
ext2/ext2fs	Second Extended File System
FAT/FAT12	[12-Bit] File Allocation Table File System
hadleyfs	Hadley File System Template
HDD	Hard Disk Drive
HFS/HFS+	[Apple Macintosh] Hierarchical File System [Plus]
HSL	Hadley Specification Language
I/O	Input and Output
IDE	Intelligent Drive Electronics
INCITS	International Committee on Information Technology Standards
ISO	International Standards Organization
MS-DOS	Microsoft Disk Operating System
NTFS	[Microsoft Windows] New Technology File System
OS	Operating System
OSI/OSIE	Open Systems Interconnection [Environment]
PC	Personal Computer

PCI	Peripheral Component Interconnect
RAM	Random Access Memory
SCSI	Small Computer Systems Interface
USB	Universal Serial Bus
VFS	Linux Virtual File System

## CHAPTER ONE: INTRODUCTION

Computers are well understood, but input and output (I/O) are not. A number of mathematical models exist to describe all forms of computation, but no comprehensive model exists for I/O. In fact, networking – a small subset of I/O – has a better-developed formal model than I/O as a whole.

Our immediate application area, digital forensic investigation, deals almost entirely with I/O.; it was, in fact, during investigation of other possibilities for work in this area that we discovered the lack of formalization we seek to remedy. The lack of understanding of I/O presents problems of verifiability and clarity for the forensic investigator. These problems are currently resolved via repeated testing of forensic tools and best-practices technical explanations, but the lack of a standard model still makes the validity of testing constantly questionable and explanations of I/O in court and to other laypersons difficult in an era of increasingly stringent requirements for scientific evidence. (*Daubert, Kumho*)

Digital investigation is one of the most obvious but far from the only area of computer science to suffer from the poor understanding of I/O; modern operating systems contain numerous device drivers, each one intended to broker some aspect of input, output or both, many of them questionably compatible. Scattershot development in an area of computer science is nearly always a key symptom of poor formal understanding of the area. Both hardware and software sides of I/O are affected: every translation among the various forms of data that modern computers recognize is more art than science given the current lack of a model.

We seek to create that model. The presentation of Hadley—and of the initial practical system using it—is offered in hope to eliminate a source of guesswork in computer science: how low-level I/O tools, in the forms of system software and forensic tools, process data.

In the strictest sense, a *computer* is only the microprocessor and primary memory that reside on the motherboard of a typical modern PC. Computers receive input from, and generate output to, a variety of *peripherals*. Examples of peripherals are fixed disk drives, removable magnetic disks, removable optical disks, network interface cards, printers, analog modems, ISDN modems, video cards, sound input devices, sound output devices, keyboards, pointing devices, et cetera. These peripherals store, transmit or receive data; but for anything to be done computationally with this data, it must arrive either in primary memory or at the processor.

All of the above peripherals are also part of a hierarchy that extends both above and below them. To create an example that we will use throughout this paper, a typical input request would be to retrieve a range of bytes from a file on a fixed disk. To do this, a typical PC-architecture computer must interface with and/or understand, in turn:

- The PCI expansion bus.
- The IDE device controller residing on the PCI expansion bus.
- The file system of the partition on the fixed disk drive that contains the file.

At this point, the computer may retrieve the necessary metadata from the file system to find the bytes it is looking for, and retrieve those bytes into primary memory.

The elements of this hierarchy are disparate and scattered. For example, a computer running a modern operating system, such as Microsoft Windows or Linux, will typically have separate device drivers for the PCI expansion bus, the IDE device controller, and the file system that contains the file. These drivers will be unrelated, interacting code: each is dependent on

each of the others, and an error in any of them will cause the others to fail in unexpected—and often nearly untraceable—ways.

The elements of this hierarchy are also highly duplicated. A recent version of the Linux operating system contained file system drivers for over twenty file systems, several of which were over 200K in source size—yet the functionality of every file system is fundamentally similar. Various IDE controllers must have their own drivers as well. If the fixed disk drive resides on a SCSI bus instead of an IDE bus, then the hierarchy of drivers is different yet again, for an essentially similar function. (Torvalds)

As more and different peripherals (such as fixed disks and their controllers) and means of accessing those peripherals (such as file systems) become available, the difficulties faced in accessing them all multiplies. This is a concern to anyone who wishes to create or maintain stable, reliable software systems capable of interfacing with most available peripherals; it is of particular concern to three types of computer professionals.

- Law enforcement officers must be able not only to read information from every conceivable type of peripheral, but to do so in a verifiable, duplicable, and completely non-destructive manner. Current systems are simply inadequate to this task: most serious attempts to produce verifiable results involve rebooting a computer into MS-DOS and running highly expensive tools that work with very few types of peripherals.
- Maintainers of current operating systems find I/O to be increasingly troublesome: Microsoft has gone on record blaming up to 80 percent of crashes in some versions of Windows on device drivers. Since most Windows device drivers are actually written by third parties, it has become increasingly difficult for Microsoft to quality-control their

own operating system with respect to its I/O subsystems; driver certification programs have lessened this problem, but do not solve it in a fundamental sense.

- Designers of new operating systems find I/O to be an even worse problem. Peripherals that work on the most popular operating systems are expected to work on all, and regardless of where the necessary effort lies, the operating system will be blamed if a peripheral fails to work with it. As the number of available peripherals explodes, and as peripheral vendors focus their driver design efforts more and more tightly, it becomes more and more difficult for any operating systems that do not already have drivers to obtain them.

We contend that all of these concerns are symptomatic of the same problem. Lack of verifiability, scattershot design techniques, high amounts of duplication of effort, and low reusability all point to inadequate description. A review of the literature finds, in fact, that little effort has been put into formalization of this particular domain.

Before we can deal with the problems of I/O for modern computers and their implications for forensic investigation, we first have to explain what we mean by computers and what we mean by I/O. There are several useful top-level definitions of a computer: several of them, including the Turing machine and the von Neumann machine, can be encapsulated as “one or more arithmetic and logical execution units able to read to and write from primary memory”. We will use this definition. (Aho 1992)

A usable top-level definition of I/O follows rapidly. Input may be defined as any operation that receives information from a device external to the computer and places it in the primary memory, and output may be defined as any operation that transmits information from

the primary memory to a device external to the computer. Finally, we refer to any device capable of generating input, accepting output, or both, as a peripheral.

Often, input and output go through several peripherals. For example, a hard drive is not read directly; a hard drive controller reads it, and the controller interfaces with a system bus controller, which in turn is the peripheral that actually interfaces with the computer. The graph of peripherals that communicate with each other can be viewed as a tree with the computer as the root node; we refer to this tree as the peripheral hierarchy.

These definitions are far from universal; in fact, there is practically no definition in the realm of input and output that can be called universal. A search of the literature finds a good deal of work on the specification of file systems from the user and developer's point of view, but none that relate to system software. Forensic tools are, as a rule, presently tested based solely on their predictability: a full source code level audit of any tool, let alone any operating system component, to ensure precise and correct operation is basically impossible.

Examining the state of device driver development in operating systems whose source code is available makes this lack of definition even more clear. Typically, each individual node of the peripheral hierarchy receives its own device driver; the Linux operating system comes with over 2,000 device drivers of one type or another (Torvalds). Different programmers or different teams often write these drivers—including drivers with a high degree of similarity, and drivers that coexist on paths within the device hierarchy. The problems this scattershot development can create are both obvious in nature and well documented in fact: Microsoft blames as many as 80 percent of Windows system errors on device drivers, and a recent static compiler analysis study found that in Linux 2.4.1, device drivers had error rates up to seven times higher than the rest of the kernel. (Mason 2002, Chou 2001)

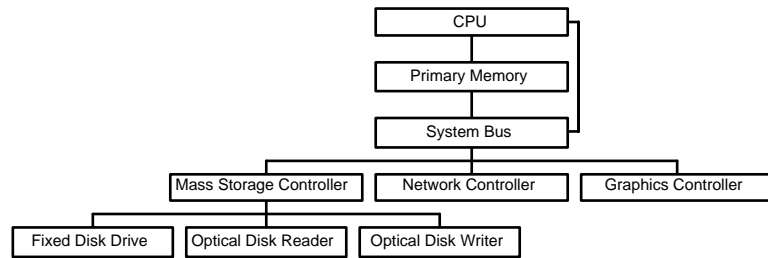


Figure 1: The Peripheral Hierarchy

The general problems of poorly understood peripheral I/O are less obvious but at least as severe. With the exception of a few worthy efforts such as the USB Human Interface Device specification, a company that wishes a device to be supported by more than one combination of processor architecture and operating system must write a device driver for each such combination. Further, some operating systems (notably Windows) change device driver formats between revisions, rendering older device drivers—and, often, older devices—unusable. The constant parade of device driver changes implies a constant parade of changes in the way devices work



## **CHAPTER TWO: LITERATURE REVIEW**

### **The OSI Networking Model**

The sort of problem we are dealing with is not new. When computer-networking professionals faced a similar problem of incompatibility and inconsistency, the community devised a straightforward, unifying basic model.

ISO 7498-1 is the International Standards Organization's reference model of networking: the Open Systems Interconnection model. The document spends its first five sections describing in some detail the assumptions and thought processes that led to the model, which is finally described in the sixth and seventh sections. It is these sections that primarily interest us.

The Open Systems Interconnection Environment (OSIE) is defined in seven layers of operation, numbered highest at the most abstract and lowest at the hardware level. They are discussed in the reverse order of their numbering.

The designers of the OSI model created its layers based on 5 principles:

1. A layer should be created where a different level of abstraction is needed.
2. Each layer should perform a well-defined function.
3. The function of each layer should be chosen with an eye toward defining internationally standardized protocols.
4. The layer boundaries should be chosen to minimize the information flow across the interfaces.

5. The number of layers should be large enough that distinct functions need not be thrown together in the same layer out of necessity, and small enough that the architecture does not become unwieldy.

The first two layers, the most abstract, are the least defined by the document.

- Layer 7 is the **application layer**. It provides services for end-user applications to access the OSIE, and is the only layer that does. No layer of the OSIE sits above the application layer, and the application layer is the only entry point to the OSIE.
- Layer 6 is the **presentation layer**. The presentation layer defines the syntax of information being transferred, translating it between the application layer and the rest of the OSIE.

The middle three layers are the heart of the document's definition of networking. These are the layers that provide the services typically associated with a network by network software developers and users.

- Layer 5 is the **session layer**. This is the layer that has the responsibility for opening, maintaining, and closing connections (*sockets* in TCP/IP).
- Layer 4 is the **transport layer**. This is the layer that handles error detection and correction, sequencing control and reordering, and flow control.
- Layer 3 is the **network layer**. This is the layer that handles network routing, relaying, and gateways between sub-networks.

The bottom two layers, like the top two, are also less thoroughly defined.

- Layer 2 is the **data link layer**. This is the layer that handles data transmission between individual points on the network. It may also handle routing within a sub-network.

- Layer 1 is the **physical layer**. This is the layer that describes the physical communications media.

The layers are very rigidly defined; to the point that, for example, as far as connection-based and connectionless services are defined, it is restrictively specified which layers may convert between the two. The document defines explicitly the services provided by each layer to the layer above it, and the services used by each layer from the layer below it.

The OSI networking model's domain is obviously different from the one we consider here: it concerns one computer communicating with another, while we are interested in a computer communicating with components of itself. However, it is difficult to overstate the effect that the OSI model has had on its domain, the networking community; it has influenced everything from formal reasoning about networks to certification for network professionals.

(Cisco, Barjakatrovic)

<b>Layer 7: Application</b>
<b>Layer 6: Presentation</b>
<b>Layer 5: Session</b>
<b>Layer 4: Transport</b>
<b>Layer 3: Network</b>
<b>Layer 2: Data Link</b>
<b>Layer 1: Physical</b>

Figure 2: The OSI Networking Model

### **Hardware Standards**

As we have intimated, there is a lack of formal description of interface mechanisms; however, there is no such shortage of technical documentation. As examples, the two most common interface systems for fixed disk drives are very well-documented. More generally, the hardware design community is well aware of the need for increased formalization of I/O.

The AT Attachment with Packet Interface (ATA/ATAPI) standard is the technical name of the hard drive interface system commonly known as IDE (for Intelligent Drive Electronics). ATA/ATAPI is maintained by Technical Committee T13 of the International Committee on Information Technology Standards (INCITS); its finalized documents are available from the American National Standards Institute (ANSI) and its drafts may be downloaded via the Web. (INCITS-T13)

The Small Computer Systems Interface (SCSI) standard is maintained by Technical Committee T10 of the INCITS, which handles lower-level interfaces. As with T13, T10's draft

documents are available online and its finished products may be purchased from ANSI.

(INCITS-T13)

Smotherman some time ago developed and promoted a taxonomy of hardware-level I/O modes in use at the time (Smotherman 1989). Shimizu notes the need for formal specification in functional validation of hardware designs (Shimizu 2002) and Hill et al. advocate *Wisconsin I/O*, a framework for describing I/O architectures at the hardware level (Hill 1999).

### **Modeling Languages**

Ciancarini, Fogli and Gaspari describe *Gammalog*, a declarative language for problem solving that includes the concepts of coordination. They illustrate its expressive power by including a simple “operating system” written in the language. Actually calling it an operating system, however, is overstating the case; the functions that actually read and write files are not described in the paper. (Ciancarini et al., 2000)

Heisel describes an attempt to specify the user view of the UNIX file system in the Z modeling language. The attempt is interesting to us because of the formal level at which it models the file system in question. It is, however, concerned with what the user sees, not what is actually stored within the file system itself. Actual discussion of the fine points of the model would require a specification of the Z language, which Heisel does not give and which we will not give here. (Heisel, 1995)

Heydon and Tygar describe *Miró*, a formal system for specifying and checking security constraints under UNIX. *Miró* is not theoretical; its designers intend it to be implemented and used by system administrators. Its domain is limited to security. (Heydon & Tygar, 1994)

Miró defines two languages: an *instance language* to create security *configurations*, which are simply matrices of subjects versus objects on a file system with each cell being “grant” or “deny”; and a *constraint language* for security *policies*, whereby each policy is a set of constraints which is in turn a set of configurations, and a configuration is consistent with a policy if it is within each of that policy’s configurations.

The instance language is a (relatively) simple set of named boxes and lines. A box can specify a set of users or a set of files, and can contain other boxes that are subsets. A “user” box can have a directed line drawn from it to a “file” box, the line specifying either the granting or the denying of certain accesses. More specific arrows have higher priority. Each box has a type that gives it certain attributes; the types are specified in an object-oriented manner, children inheriting attributes from their parents.

A constraint specifies a pattern of instance pictures, just as a language specifies a pattern of strings. A *box pattern* specifies, in a predicate language, the characteristics of the boxes it matches. *Semantics arrows* specify access permissions to be matched between boxes that match the box patterns they connect. *Containment arrows* specify containment relationships to be matched between boxes that match the box patterns they connect.

The remainder of the paper describes the implementation of the software system itself, and is beyond the scope of this review. Miró is interesting to us for its formal specification of a methodology used by file systems, but in the end its precepts and mechanisms are limited to security.

## **Functional Languages**

Considering input and output in formal models of computation is not a new idea; hence, especially as functional languages are often used to generate formal models, neither is considering useful input and output for functional languages. Most attempts to do so have their roots in Landin's demonstration of correspondence between a modified lambda calculus and ALGOL 60 (Landin 1965), but approaches diverge quickly. We consider two of the more popular here.

Data flow models have been part of languages since quite early (Dennis 1975), and remain influential. Gordon some time ago dealt with the problem of data flow in lazy-evaluation functional languages, and Broy more recently considered the creation of an algebra specifically for stream processing functions. (Gordon 1993, Broy 2001)

Other than traditional data flow and variants on it, monads have been the major mode of I/O in functional languages. Wadler first extended them to describe state (Wadler 1990) and Gordon expounds further on them and on other constructs for functional language I/O (Gordon 1994).

Both ordinary data flow and monads are implemented in the programming language Haskell (Peterson), considered one of the most influential functional languages where attempts to deal with I/O are concerned (Gordon 1994).

## **Summary**

All of the work we have found here shares a common thread: it is related but not equivalent to what we seek to accomplish. Monads are models of I/O strictly from the computer's point of view. Hardware specifications for I/O and verification cover only hardware level communications. The OSI model is nearly exactly what we seek, but its domain is limited to networking. Multiple means of integrating I/O into functional languages exist, but no functional language that we have discovered exists with I/O as its primary purpose. In short, the specifications that have the practical use we seek are too specific and the specifications that are general enough are designed to deal with theoretical I/O rather than real bit-moving.

We are free to offer, as our contribution, a general, extensible and constructive model of I/O, and some initial demonstrations of its real practical use.



## CHAPTER THREE: RESEARCH DESIGN AND METHODOLOGY

### The Hadley Model

#### Definitions

We begin by clarifying a few definitions.

**Definition** (Computer): A *computer* is one or more arithmetic and logical execution units able to read both data and instructions from, and write data to, a random-access primary memory.

**Definition** (Input): *Input* is the process of writing data not generated by a computer's arithmetic and logical execution units into that computer's primary memory.

**Definition** (Output): *Output* is the reading of data from a computer's primary memory by anything other than that computer's arithmetic and logical execution units.

**Definition** (Device): A *device* is any mechanism capable of storing, producing, or accepting data.

**Definition** (Logical Device): A *logical device* is a "device" within a computer or within another device that does not have discrete physical components; it is a theoretically unnecessary device that exists for organizational convenience.

**Definition** (Peripheral): A *peripheral* is any device capable of producing input to a computer or reading output from a computer.

**Definition** (System): A *system*, or a *computer system* is a computer together with all its interface mechanisms, excluding peripherals.

The definition of a computer is intentionally somewhat vague. Hadley does not model computers, and it should work with any computer that has memory. In more general terms, a computer is the CPU and RAM, input and output are defined as generally understood, and hard drives and the like are classed as peripherals.

### **Layer Models of I/O**

Hadley is a layered I/O model; more precisely, it is a system of layered models, based on the fundamental observation that all input and output is simply the translation and transport of data. Hence, each layer in a Hadley model represents a physical location through which data is transported in the I/O process or a view of data that appears during the translation component of the I/O process.

The fundamental similarity between Hadley and the OSI model is that in both, each one of these layers represents either a different level of abstraction or a different locus of functionality. For example, as the OSI model treats the physical communication medium and the addressing separately (layers 1 and 2, above) the Hadley model separates the physical storage media and the mechanism of presenting that data digitally (layers H-1 and H-2, below).

The fundamental dissimilarities arise from the fact that each layer of a Hadley model describes a location or representation that already exists, not one that we as the designers of Hadley may define: if Hadley is to be of any use, then its development must be driven by what I/O is, not what we believe it should be. Layers of the OSI model can be clearly associated with specific segments of data in network packets, whereas, since we cannot lay any new requirements on the way data is represented, layers in the Hadley model are better seen as stops

along the path of I/O. The constructive use of Hadley, as we will show, is in the transmission and translation of data between these layers.

I/O is, as we noted, comprised of both transport and translation. I/O is also comprised of both hardware and software components. Hardware I/O is concerned with turning data on peripherals into bits and bytes that a computer can deal with, and the reverse. Software I/O is concerned with what a computer does with those bits and bytes. There are significant translations and transports that happen in hardware before a computer sees bits and bytes; there are significant translations that happen in software before applications see those bits and bytes as sequential files, the preferred I/O model in modern operating systems.

We did not attempt to model the whole of computer I/O with one set of layers. I/O translations and transport take place both in terms of moving data between peripherals and translating data that is already accessible between different usable forms. These two areas are both concerned with providing I/O services, and both lend themselves well to layer diagrams; but they are different enough in scope and nature that distinct layer models are desirable to maintain clarity. In turn, there are two common and distinct modes of software I/O that, though related, are different enough it is clearer to represent them with two distinct layer models than to try to shoehorn them into one.

Stopping at sequential files is somewhat arbitrary, as our models could involve higher and still more meaningful layers. The models are extensible to these, and may be so extended – but these extensions are much less urgent, as compared to I/O itself, file formats are generally well documented and well understood.

In choosing layers of representation for each individual layer model, we follow principles similar to those that animated the choice of layers for the OSI model, and that we believe will provide similar advantages for our purposes.

1. A layer should be created where a different level of abstraction is needed.
2. Each layer should represent a well-defined location or abstraction.
3. Each layer should be chosen with an eye toward representing well-recognized extant components of peripheral I/O.
4. The layer boundaries should be chosen to keep the information flow across interfaces between layers well ordered.
5. The number of layers should be large enough that distinct locations or abstractions need not be thrown together in the same layer out of necessity, and small enough that the model does not become unwieldy.

### **Input and Output in Hardware**

Physical devices external to a computer generate all important input and output. Theoretically, a computer must be able to read and present data, which traditional models provide for only by examining initial and end state. Practically, a computer must also be able to permanently store data, which RAM does not do under typical architectures.

The hardware layers must represent the primary categories of physical components along the I/O chain of modern computer systems. No categorization is perfect, hence devices whose layer is unclear will always exist, but most I/O devices other than the CPU and primary memory can be clearly and more or less distinctly thought of as one of the following:

- Physical peripheral electronics that move data into and out of actual representations in the real world;
- Peripheral interface electronics that broker interactions between peripheral electronics and standard computer system interfaces;
- I/O busses that permit the connection of peripherals to computer systems through standard interfaces, or
- System busses that permit data exchange among I/O busses, primary memory and the CPU.

We choose the layers for the Hadley model of hardware accordingly.

### **The Layer Model of Hardware**

The Hadley model of hardware, shown in Figure 3, considers six layers of physical devices, beginning with the physical peripheral electronics and ending with the components of the traditional Von Neumann machine. We call these layers H-1 through H-6 (for Hardware-1 through Hardware-6). Figure 3 also gives examples of how components of modern computer systems fit into the hierarchy.

In general, we say that components within a given layer may send and receive data in their appropriate native formats to and from other components within that layer.

#### ***Layer H-1: Peripheral Electronics***

Layer H-1 is comprised of *the physical medium by which peripheral data is represented, and the electronics necessary for rendering that data into and/or from digital signals.*

This is by far the least uniform layer of the model: magnetic platters or tape and read/write heads may represent information, as may switches on a keyboard, the print heads of a printer, or the frequency modulators of a modem. In modern computer systems, all physical interaction with the user in the ordinary use of the system takes place with layer 1 components of peripherals. The layer H-1 components are also where more extreme forms of data recovery can begin – with enough effort and sufficient resources, it is possible to extract a remarkable amount of data from the removed platters of seemingly dead fixed disk drives. Digital forensics at this level may require significant resources, including a clean room. (DriveSavers)

Layer H-1 components have physical state as their primary representation of data. At the request of layer H-2 components they translate that state into digital signals to be read by layer H-2 components, or translate digital signals provided by layer 2 components into that state.

### ***Layer H-2: Peripheral Interface Electronics***

Layer H-2 is comprised of *any devices that a peripheral uses to prepare data before offering it to the higher layers of a system, or propagate data to its physical media after being given it by a system*. Its separation from layer 1 is dictated by the primary representation of data as digital information rather than as a physical state.

This layer is nearly as varied as layer H-1; however, in it we start to see uniformity. At the top of it, data must, at the very least, be in a digital format, either having been received from the higher layers of a system or ready to be offered to that system. The easiest examples of layer H-2 components are found on the controller card of an HDD – the card stores no actual data, but communicates with the drive electronics and the IDE bus.

Layer H-2 components may request the physical state of layer H-1 components as digital information, or request that that state be changed according to digital information the layer H-2 components provide.

Layer H-2 components may receive digital data from, and send digital data to, layer H-3 components.

### ***Layer H-3: I/O Bus***

Layer H-3 is comprised of *any devices that are attached to the system bus, do not terminally produce, receive or contain data themselves, and serve to connect to peripheral interface electronics or other I/O bus devices*. Its separation from layer H-2 is dictated by the physical co-location of layer H-2 components with their peripherals, whereas layer H-3 components are part of a computer system.

An I/O bus device must be attached to the system bus, but may not be part of the system bus. It must not terminally provide, receive or contain data itself, except for commands and diagnostic information. (It is allowed, of course, to transiently provide, receive and contain data, either through caches or simply as it transmits and translates signals, but it cannot be that data's authoritative source.)

Most devices that extend system busses serve primarily to allow the system to communicate, not to provide or receive data, and are hence layer H-3 devices—for example, IDE disk controllers and USB serial controllers are layer H-3 components. Generally, physical connections made between systems and peripherals are made between layer H-3 and layer H-2 components.

It should be further noted that the description explicitly permits “chaining” of I/O bus devices. This is also a point of potential ambiguity in the layers: an external ATA hard disk attached via a USB connector likely has within its case a USB-to-ATA device. Such “gateway” devices are sold both separately and as part of external drive packages, so we cannot go by how the device is packaged or marketed. We say therefore that *any* such gateway device remains in layer H-3, as long as it is not unique to a specific peripheral; therefore, controller cards on IDE hard disk drives remain in layer H-2, but a USB-to-ATA enclosure – even one obtained as part of an external drive kit – is in layer H-3.

Layer H-3 components may receive digital data from, and send digital data to, layer H-2 components.

Layer H-3 components may receive digital data from, and send digital data to, layer H-4 components.

#### ***Layer H-4: System Bus***

Layer H-4 is comprised of *any devices in a system that are under the direct control of the CPU and which the CPU can instruct to input data to, or output data from, primary memory.*

The extant practical distinction of system bus components from storage I/O, serial I/O and other peripheral I/O controllers in modern computer system architectures dictate layer H-4’s separation from layer H-3.

For a component to exist in layer H-4, the CPU must be able to communicate with it directly. System busses actually serve many more functions than we consider; for our purposes, they are conduits between I/O busses and primary memory. Layer H-4 is generally comprised of the traditional system bus; for example, Intel PCIs<sup>et</sup>® chipsets are Layer H-4 components.



Layer H-4 components may receive digital data from, and send digital data to, layer H-3 components.

Layer H-4 components may read from and write to primary memory.

Layer H-4 components may send feedback to, and receive commands from, the CPU.

Note that this opens two data paths from layer H-4 components to the primary memory: access by layer H-4 components themselves, or by the CPU storing the results of communication with layer H-4 components to, and reading data for communication with layer H-4 components from, primary memory. The former method is typically known as *direct memory access* (DMA); the latter method is typically known as *programmed I/O* (PIO).

### ***Layer H-5: Primary Memory***

Layer H-5 is the primary memory—the von Neumann RAM—of the computer. All peripheral input is assumed to be written into primary memory either by layer H-5 components or by the CPU while it communicates with layer H-5 components; all peripheral output is assumed to be read from primary memory by layer H-5 components or by the CPU while it communicates with layer H-5 components.

### ***Layer H-6: CPU***

The CPU is the functional arithmetic and logical unit of the von Neumann machine at the heart of the computer. It is the source of all output and the destination of all input. We assume that the CPU can read from and write to primary memory directly, and can send commands to, and receive feedback from, the system bus.

Multiple CPUs have significant effects on implementation issues, but no effect on the model itself, as they simply produce and receive commands and data from and to multiple points instead of one.

### **Uses of the Hardware Model**

The benefits of classification of hardware into layers for computer science purposes are traditional and fairly obvious; there are benefits of clarity to the forensic investigator as well. As one example, acceptance of the Hadley model makes it immediately clear that every component of a fixed disk other than the platters can be switched out for an identical component without affecting the data stored on the platters – a fact that is normally complex to explain.

We can also note that all data is “written down” or “read up”: data must go up the hierarchy to be read by the CPU and down the hierarchy to be written by it. As long as we verify the following three things about a technique, we can say that technique does not modify data:

- none of the commands the CPU sends modifies stored data other than component status,
- the downward write path from the CPU and main memory is never used, and
- the peripherals themselves never change data physically stored at layer 0 without instructions from the CPU.

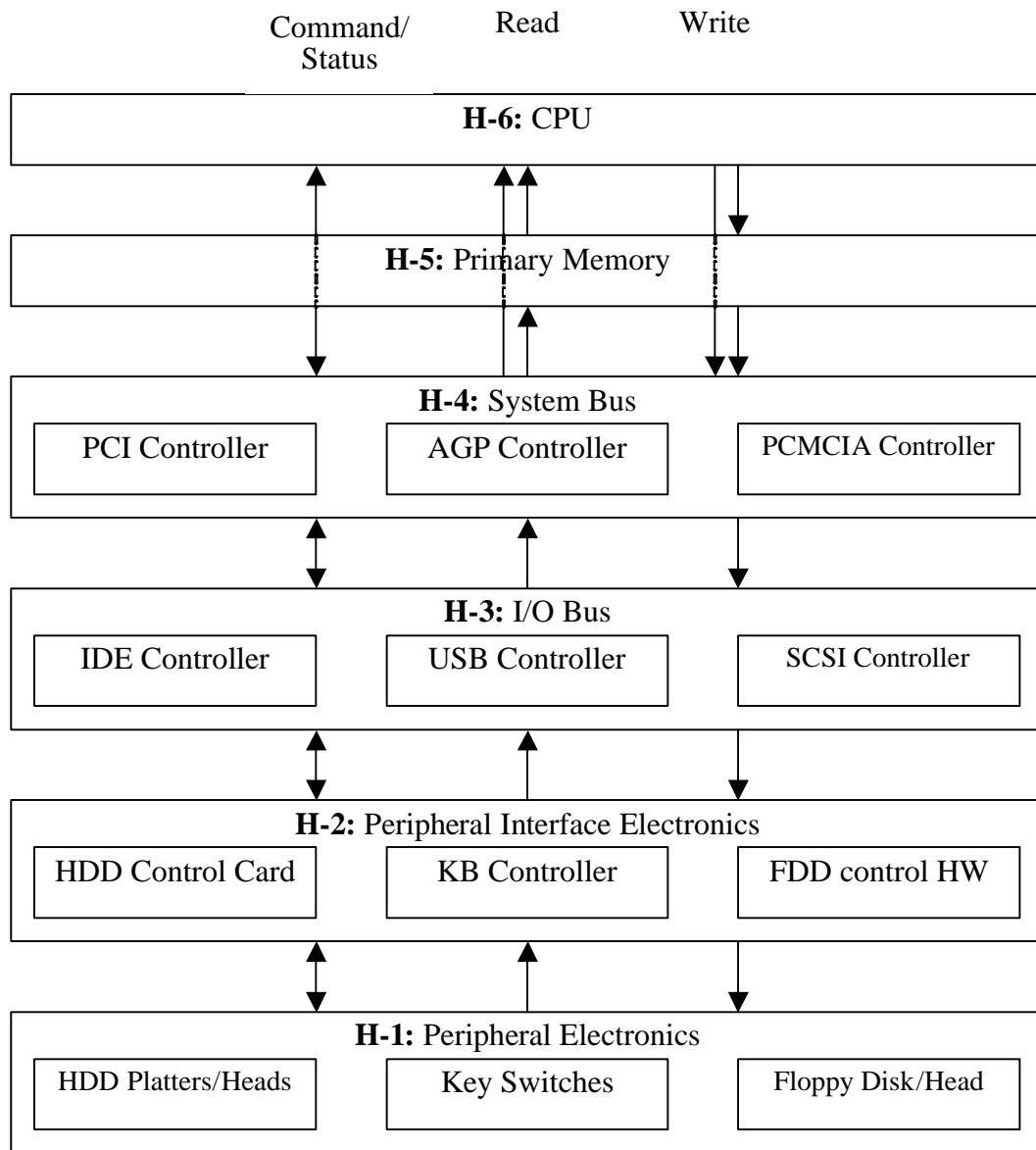


Figure 3: The Hadley Hardware I/O Model

### Input and Output in Operating Systems

The software side of the Hadley model is concerned with the way operating systems obtain input from and generate output to peripherals.

For I/O purposes, any operating system is essentially capable of doing three things:

- reading primary memory,
- writing primary memory,
- sending commands and data through the system bus, and
- receiving information, including interrupts, from the system bus.

Internally, modern operating systems view all peripherals as sequential files: either streams or secondary random access memory spaces. Streams are “files” that act like keyboards – files from which the next bit can always be read and to which the next bit can always be written, if it is ready, but previous bits generally cannot be reread and there is no write location to specify. Secondary random access memory spaces act like traditional files, a space of bits in which any bit is readable or writable at any time given its addressed location.

The application view of I/O in operating systems is also that of streams or spaces: operating systems view peripherals as sequential files, and applications see files within those files in turn. The Hadley model for operating system I/O is concerned with providing the operating system its sequential file view of peripherals, and with providing applications access to the sequential files stored on those peripherals.

This is a model of abstraction layers, sharing similarities with Carrier’s work (Carrier 2002); Carrier suggests the use of abstraction layers as a guide in the development of forensic tools, and the use of these abstraction layers as an intrinsic part of the tool. We differ from Carrier in the scope of our layers – his primary example is layers within a specific file system or document format – and will go further by, in our next section, depicting a universal system in which translation between our abstraction layers can be defined in a predictable, verifiable and immediately usable manner.

Figure 4 shows where software I/O translation fits in with the hardware layers of the Hadley system. Since software I/O translation is actually purely computational, it affects how the CPU reads and writes data from and to primary memory and the system bus, and affects no other layers.

In real world terms, translating between layers of software I/O simply means the translation of data into differing formats among various types of random-access and stream-based data models. These two classes of model are different enough that it is easier to describe them with two different layer models; we will examine the simpler model first.

### **The Layer Model of Stream Peripherals**

Stream peripherals encapsulate simple ordered input and output of digital data. A stream peripheral provides the ability to read digital input from it and/or write digital output to it; it provides nothing else, other than status checking. In particular, a stream peripheral has no ability to go back and retrieve data that has already been read once.

Stream peripherals include human input devices (keyboards, mice, game controllers, et al.) and network interface cards.

The Hadley stream peripherals model consists of layers for direct addressing of the peripheral including commands to be sent to it –the data sent to the peripheral over the I/O and system busses; addressing of the peripheral data as a byte stream; and addressing of that byte stream as a sequential file. It is shown in Figure 4. We refer to its layers as OS-S1 through OS-S3 (for Operating System-Stream 1 through Operating System-Stream 3.)

### ***Layer OS-S1: Peripheral Addressing***

This layer is provided directly by the system bus, and is exposed only within an operating system. It is comprised of the raw stream of data sent to and received from a peripheral through the system bus and the I/O busses after it, including commands that govern that peripheral's operation. Generally, only operating systems deal with this view.

### ***Layer OS-S2: Stream Addressing***

This layer is exposed within an operating system and to system-level programmers. It is access to the stream peripheral as a raw, linear stream of byte data. Generally, only operating systems deal with this view.

### ***Layer OS-S3: Sequential File Addressing***

The API of an operating system exposes this layer to applications. It is access to the stream peripheral filtered in such a way that it may be treated as a non-seeking ordinary sequential file, usually with a location in the operating system's universal file system environment. This is the normal application view of inherently stream-based peripherals, such as serial and parallel ports. There is no user layer for these peripherals, as they typically are not accessed directly by the user.

### **The Layer Model of Random Access Peripherals**

Random access peripherals encapsulate memory-mapped input and output of digital data to arbitrary locations in a finite array of available space: the typical storage disk, or tape, or "flash drive". Any part of a random access peripheral may be read or written at any time, subject

only to performance constraints. Note that performance does not influence whether a peripheral is a random access peripheral or a stream peripheral; a tape drive is as much a random access peripheral as a solid-state memory-based virtual disk drive.

The Hadley random access peripherals model is shown in Figure 4. We refer to its layers as OS-R1 through OS-R5 (for Operating System-Random 1 through Operating System-Random 5). It consists of layers for:

- Direct addressing of the peripheral including commands to be sent to it –the data sent to the peripheral over the I/O and system busses;
- Addressing of the peripheral data as a single flat space of bytes;
- Addressing of the peripheral data as partitions within that flat space;
- Addressing of those partitions according to their respective file system formats; and
- Addressing of the individual sequential files in those file systems.

### ***Layer OS-R1: Peripheral Addressing***

This layer is provided directly by the system bus, and is exposed only within an operating system. It is comprised of the raw stream of data sent to and received from a peripheral through the system bus and the I/O busses after it, including commands that govern that peripheral's operation. Generally, only operating systems deal with this view.

### ***Layer OS-R2: Flat Space Addressing***

This layer is exposed within an operating system and to system-level programmers. It is access to the random-access peripheral as a single large linear array of bytes. For storage media, this is the raw image of the entire physical disk. A software-based RAID system would combine

the flat spaces of more than one physical storage device into a single logical flat space at this level.

### ***Layer OS-R3: Partitions in Flat Space***

Most modern operating systems permit large random access devices to be *partitioned* into several smaller logical random access peripherals. This layer, exposed within an operating system and to system-level programmers, represents each of those partitions as a randomly-accessible linear array of bytes, offset within the device's flat space. For storage media, these are images of the disk's partitions.

### ***Layer OS-R4: Formatted Partitions***

Partitions are typically *formatted* with a system for the organization of sequential files on that partition. This layer represents the formatted partition, including partition metadata such as file catalogs. The API of an operating system exposes this layer to applications. This is the normal programmer view of the file system.

### ***Layer OS-R5: Random-Access Sequential Files***

A partition format typically makes provision for storing an arbitrary number of sequential files within a partition, so long as the sum of the size of the files within the partition does not approach the partition size itself too closely. This layer represents each file within the partition as a randomly accessible linear array of bytes, segmented within the partition's flat space. The API of an operating system exposes this layer to applications. This is the normal user view of the file system.



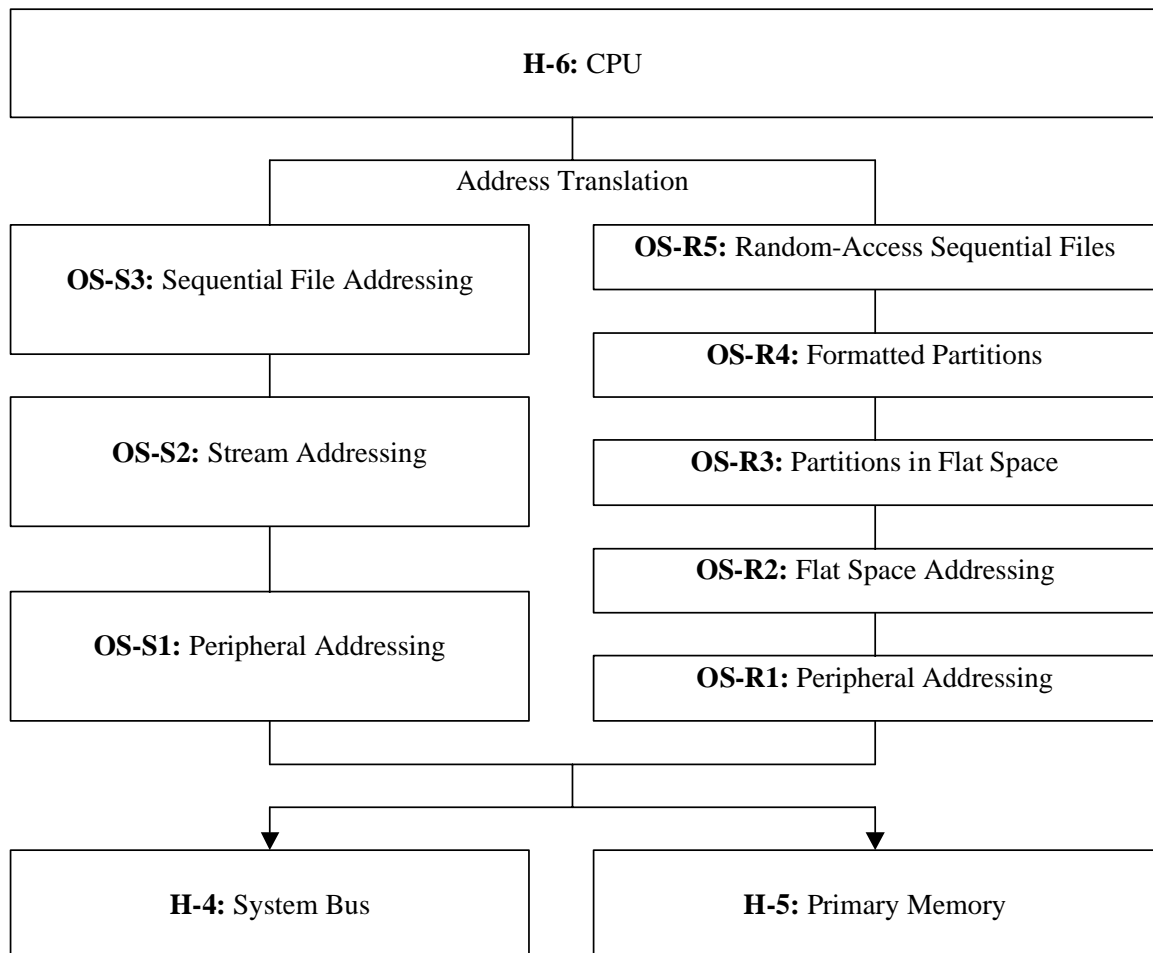


Figure 4: Software I/O and the Hardware Layers

### Uses of the Software Model

The Hadley model of software I/O, of course, does not provide the investigator any new abilities to pull data from a fixed disk drive – there are already programs to easily obtain the image of an entire drive. The Hadley model can, however, give the investigator two things.

The first benefit of the Hadley software model is clarity, similar to the examples given for the hardware model. Examining the Hadley model, it becomes intuitively clear how a “deleted”

file can actually still remain – it has been removed from the fourth and fifth layers of the model, but still exists in the third on down. The past existence of data that has been completely removed can be inferred from information from the metadata in the fourth layer. Considering the software layers to lie atop the hardware layers, data that has been removed from the entire software-based system might still exist within the hardware, recoverable from the magnetic platters if poorly erased or, in exotic cases, still in the buffers of a fixed disk drive.

## **The Hadley Specification Language**

### **Overview**

HSL is a context-free, essentially functional language that compiles readily to C; this is its intended mode of practical use. In its current incarnation it intentionally has only one control construct (the conditional return value) and very few forms of address. As the development of the language continues, we will add more features only as found to be either necessary or convenient for representation.

We describe HSL as *essentially functional* because, though it follows the general form of a functional language, it allows for persistent variable declarations to represent data at specific locations. It does not allow assignment to these variables, so these declarations could be rewritten as functions themselves; however, for the sake of clarity and efficiency of specification, we have not done so.

An HSL module's purpose is to provide translation functions that extract desired data from memory spaces. HSL captures the storage characteristics of peripherals: the locations of

sensible data in their memory spaces, and how they provide access to their connected peripherals in turn.

An HSL module is comprised of one or more declarations of spacetypes. A spacetype is meant to represent a data area, whether literally contiguous or conceptually organized; it is considered contiguous internally. It may contain variables defined in terms of locations within its area and functions defined in terms of those variables. Any of these terms may be comprised of complex expressions and be evaluated at runtime.

## Spacetypes

*Spacetypes* are HSL's fundamental units of spatial organization. A spacetype is analogous to a class. All non-built-in functions and variables must be declared within spacetypes.

```
spacetype <spacetype-name> {  
  
    . . .  
  
}
```

Declaring a spacetype is as simple as giving its name; the meat of the declaration is the subspaces, variables and functions.

## Subspaces

Hadley spaces can contain other spaces.

```
use <spacetype-name>.  
  
subspace <subspace-name> is <spacetype-name>  
    at <location>.
```

The *use* directive loads the headers necessary for one spacetype to understand another. A spacetype must be used before subspaces of that type can be declared directly or used in *with* clauses (described below, under *Functions*). The *subspace* directive declares a subspace within the current space, giving it a name and location.

### Addressed References (“Variables”)

“*Variables*”, or addressed references, are HSL’s fundamental units of data. They are read-only and may not be assigned. As such the term “variable” is a misnomer, initially chosen for the familiarity of the **var** mnemonic and because of HSL addressed references’ syntactic similarity to variables in languages such as C. The term will be dropped in later versions of HSL.

```
var <variable-name> is [public]  
    <variable-type> [ [<array-length>] ]  
    [<endian>] [width <width>] at <location>  
    [minor <bits>].
```

Each reference has a type; currently, this type can be the primitive *integer* or *char*. Single bits may be handled by an integer of width 1, as described below. Integers are unsigned; characters, theoretically untyped, have undefined signness. We will discuss the internal representation of integers and chars when we discuss the current HSL runtime.

Each reference has a specific offset location at which it occurs in its enclosing space, and may have, optionally, a minor location specifying at which bit it starts within its offset byte (i.e., octet).

Integer references have a width in bits and a given endianness. Technically endianness may be left undefined, but doing so for an integer of other than 8 bits' width, or that occurs anywhere other than on a byte boundary, leaves the value of that reference undefined as well.

A char reference or an integer reference of any width may have an array count. Array counts work as typical in C-style languages, except they may be defined by any valid integer expression.

Addressed references are not strictly necessary in HSL, as they simply collect several function definitions based around *intAt* and *charAt*, described below, that make it easy to access data at specified locations. However, they greatly simplify the syntax of typical HSL modules.

A *public* reference simply defines a function named after itself, eliminating the necessity of writing an accessor function. Currently only integer references may be public; this will be extended in later versions of HSL.

## Functions

*Functions* are HSL's fundamental units of execution organization. A function on a spacetype has access to all of that spacetype's variables and other functions when called. Functions are analogous to methods, but are true functions—they cannot have side effects.

```

function <function-name> is <function-type>
    ([<parameter-name> is <parameter-type>
    [, <parameter-name> is <parameter-type>
    [...]]])
[with <subspace-name> is <subspace-type>
    at <location>
    [, <subspace-name> is <subspace-type>
    at <location>
    ][, ...]
]

{<function-body>}

```

*function-type* can be *integer* or *char* as can each *parameter-type*. Function and parameter types, unlike variables, may also be *list*; lists are described in more detail below. Explicitly passing arrays to and from functions is not permitted, but the *char* values passed may be of any length.

*With* clauses declare subspaces accessible to a function at specific location offsets. These locations may be in terms of the function's parameters; the typical application is to pass the location offset of a subspace to a function as an integer.

## **Built-In Functions**

HSL 0.1 provides built-in arithmetic, comparison, logical and direct data examination functions.

The built-in arithmetic functions are:

```
@sys.add is integer(a is integer, b is integer)
@sys.sub is integer(a is integer, b is integer)
@sys.mul is integer(a is integer, b is integer)
@sys.div is integer(a is integer, b is integer)
@sys.mod is integer(a is integer, b is integer)
```

Respectively, they perform addition, subtraction, multiplication, quotient (integer division), and modulus.

The built-in logical arithmetic functions are:

```
@sys.equ is integer(a is integer, b is integer)
@sys.grt is integer(a is integer, b is integer)
@sys.lst is integer(a is integer, b is integer)
@sys.gre is integer(a is integer, b is integer)
@sys.lse is integer(a is integer, b is integer)
```

Respectively, they return nonzero for  $a$  equal to, greater than, less than, greater than or equal to, and less than or equal to  $b$ .

The built-in logical functions are:

```
@sys.and is integer(a is integer, b is integer)
@sys.or  is integer(a is integer, b is integer)
@sys.not is integer(a is integer, b is integer)
```

They perform logical and, or and not, respectively. As in C, a value in HSL is logically true if and only if it evaluates to zero.

The built in string functions are:

```

@sys.clipcat is char(a is char, b is char)

@sys.subclip is char(a is char, start is integer,
                    length is integer)

@sys.clipclip is integer(a is char, b is char)

@sys.cliplen is integer(a is char)

@sys.clipequ is integer(a is char, b is char)

```

These perform concatenation, substring, substring indexing (returning the offset of  $b$  within  $a$ ), string length, and string equality, respectively. Several of these generate garbage, and the application programmer must occasionally perform garbage collection as described below (in *Working with HSL*).

Finally, the built-in direct data examination functions are:

```

@sys.intAt is integer( l is integer,
                    m is integer,
                    w is integer,
                    e is integer)

@sys.charAt is char(l is integer, n is integer)

```

*intAt* returns the integer value at offset location  $l$ , minor location  $m$ , with width  $w$  and endianness  $e$ , where 0 is little-endian and 1 is big-endian. *charAt* returns the string at location  $l$  of length  $n$ .

## Templates

*Templates* are HSL's method of abstracting similar functionality between individual specifications. There are two halves to templates on the HSL side: the template itself, which is



an entity unto itself like a spacetype, and the implementations of the template within HSL specifications. On the user program side, templates provide powerful registration capabilities that we will discuss in the interfacing section.

```
template <name> {  
    <function-header>  
    [function_header  
    [...]]  
}
```

The syntax of templates is quite straightforward; simply name the template then give its associated functions. Function headers are identical to function declarations, except with no **with** clauses, body or enclosing braces.

Implementing templates is equally straightforward.

```
spacetype foo {  
    implement <template-name> {  
        <function-name> [function-name [...]]  
    }  
}
```

Name the template to be implemented, then call out the functions the spacetype implements it with. The functions must be of the same declared type, including parameters, *and the same order* as the functions in the template declaration.

## Lists, Hints and Writing

As a functional language, HSL cannot support writing directly. However, it may return information that tells the user – or the system software – where to write. The constructs it uses to do this are called *lists* and *hints*.

*Lists* allow HSL functions to return arbitrary collections of *hints*, special *char* structures that have location information embedded for writing.

### Hint Functions

The following functions are available to create hints.

```
@sys.makecharhint is char(c is char, loc is integer)
```

```
@sys.makeinthint is char(value is integer,  
                           location is integer,  
                           minorloc is integer,  
                           width is integer,  
                           endian is integer)
```

```
@sys.makecharhintfill is char(c is char,  
                              size is integer,  
                              location is integer)
```

The first two functions generate hints to write character data at a given location, and to write an integer with a given value and a given location, minor bitwise location, width and endianness (nonzero for big, zero for little). The third function is like *makecharhint* except it

generates as many iterations of  $c$  in a row as necessary to fill up  $size$ . All of these functions create garbage.

All hints – even integer hints – are of type char, and may be passed as such.

## **List Functions**

The list functions are as follows.

```
@sys.newlist is list(c is char)

@sys.listadd is list(l is list, c is char)

@sys.listaddend is list(l is list, c is char)

@sys.listcon is list(l1 is list, l2 is list)

@sys.listfrom is list(l is list, f is integer)

@sys.listrepeat is list(l is list, times is integer,
                        space is integer)
```

The first function creates a new list with the given hint. It is worth noting that in this version of HSL, empty lists are not legal. The second function adds a hint to a list in undefined order. The third function adds a hint to the end of a list – if this function is used, the Hadley runtime is required to commit the contents of the added hint *after* the contents of the prior ones. The fourth function concatenates two lists. The fifth function modifies all hints in a list to increment their writing locations by  $f$ ; the intended use of this is to “bump” location hints given by a subspace to their proper locations in the outer space, and several examples of this use may be seen in the **fat12.hsl** specification. The last function repeats the same list of hints a given

number of times, with a specified space between the repetitions; this can be used to generate complex array structures.

## Expressions

*Expressions* are HSL's fundamental units for evaluating results, quantitatively and logically. An expression evaluates to logical true if and only if it evaluates to a nonzero integer. An expression may be a conditional construct, a function call, a variable reference, a variable's byte location, a variable's bit location within its byte, or a literal.

```
<literal value>

<function-parameter-name>

$<variable-name>

&!<variable-name>

&.<variable-name>

@<subspace>.<function-name>([<parameter>[ , ...]])

if (<test-expression>) {<body-expression>}

    else {<else-expression>}
```

All parameters to functions and test expressions in conditional expressions are themselves expressions. A function's body is a single expression. Functions a space is calling on itself rather than on a subspace are called using `@this`; system functions are called using `@sys`. Literal strings must be quoted. If necessary, they can be immediately followed by a literal integer, which will lock their width at that number – this is the only way to include a null (ASCII 0) character in a literal string.

## **Working With HSL**

### **Making HSL Useful**

HSL 0.1 reads memory spaces and random-access files from user space, and its VFS kernel module can read block devices. Future versions will allow generalized access to block and character devices. Before functions from an HSL module are called, the HSL runtime library must be pointed to its memory space or file if in user space, on pain of undefined results; the VFS kernel module handles this upon mount. Since function results are always completely re-evaluated, more than one memory space may be worked with at a time in the user space runtime by keeping track of which memory space should be examined when.

Two major examples of programs that use HSL specifications are included along with the HSL 0.1 runtime: an extractor program which iterates through and extracts every file in a file system disk image, and a Linux Virtual File System interface in the form of a kernel module. The best way to learn interfacing with HSL is to use and extend these programs.

We will discuss four cases of practical HSL development, then the necessary steps in detail.

### **Writing an HSL Module**

Writing an HSL module is quite straightforward:

- Declare the spacetype. A typical module will have only one; however, closely related subtypes should be kept within the same module file.

- Declare **use** statements for any subspace types the spacetype needs.
- Declare any references (“variables”) that represent statically located information in the spacetype.
- Declare functions that return dynamic information about the spacetype. To directly return the values of integer variables, declare them as public instead of defining trivial “getter” functions.
- Compile the module with **hslc**.

At this point, the module is done, and can be called from a C program as described below.

### Writing an HSL File System Module

Writing an HSL file system module is partly backward from writing a module from scratch: instead of defining the spacetype, references, and variables from scratch, the process is to examine the file system templates and figure out how to implement them.

- Decide which of the **hadleyfs** (file system reading), **hadleyfsw** (file system writing) and **hadleyugo** (UNIX file permission model) templates should be implemented. (Initially, implement only the **hadleyfs** template, test the module to make sure it works, then move on to **hadleyugo** (if appropriate) and **hadleyfsw**.)
- Declare the spacetypes. Consider separate spacetypes for the file system’s superblock, directory entries, and (if appropriate) inodes.
- Declare references to static structures, especially the superblock.

- Declare each of the file system template functions in terms of the data gained from the superblock and other static structures.
- Compile the module with **hslc**.

The process requires significant effort, but should not be any more complex than the definition of the file system itself infers. The **fat12.hsl** and **ext2.hsl** source files, in Appendix A, provide examples for file systems quite different in character.

At this point, the module is done, and can be added to the extractor or used in the VFS Linux kernel module.

### Using an HSL Module from C

Preparing to directly call an HSL module from C is fairly complex. It has the following steps:

- Initialize HSL.
- Register any template functions.
- Define a context.
- Call HSL functions on that context.
- Write out results as necessary.
- Garbage collect whenever the program is finished with the results of an HSL function.

Each of these steps is described in detail below.

## Initializing HSL

The header library for the HSL core runtime, **hadley.h**, must be included, and the core initialization function, *hsl\_CORE\_init()*, must be called before any use of other HSL functions. This function performs HSL runtime library initialization work, notably including detecting the endianness of the host machine. The results of calling HSL functions before calling the core initialization function are undefined – in HSL 0.1, the library will operate, but lacking information about the endianness of the host machine, may return radically wrong values for integers, leading to undefined results and, most likely, a quick segmentation fault.

## Registering Template Functions

Functions implemented by template may be called more easily than the standard HSL C interface, but the template functions must be registered and requested by the user program first. There are four steps to this process:

- Include the *.h* file named after the template.
- Include the *.h* files for the implementing spacetypes.
- Declare a template data type.
- Register the implementing spacetypes.
- Request a template implementation to fill the template data type.

Template data types are named based on their templates. Declarations, to use an actual example, appear as follows:



```
hsl_hadleyfs_template *fs = NULL;
```

*fs*, after registration and request, will store an implementation of the *hadleyfs* template.

To register the implementing spacetypes, their specific registration functions must be called.

```
hsl_fat12_register_hadleyfs();
```

```
hsl_ext2_register_hadleyfs();
```

Once the implementing spacetypes are registered, no further spacetype-specific functions need be called; the implementation may be requested and further function calls may be handled through the template, without the usual noise in HSL C interface function names. The dummy context parameter must still be passed.

```
fs = get_hsl_hadleyfs_implementation("ext2");
```

```
rootid = fs->RootDirectoryID(0);
```

See below for general information about calling HSL functions from C.

## Creating the Context

The context is the space on which the HSL runtime operates: the raw data which HSL specifications interpret. User space programs can use either a memory space or a file as a context. There are four HSL core library functions used to set up a context.

```
void hsl_CORE_set_context_memory_mode();
```

```
void hsl_CORE_set_context_memory_location(void *p);
```

```
void hsl_CORE_set_context_file_mode();
```

```
void hsl_CORE_set_context_file(FILE *f);
```

If the core runtime is passed a memory space, that space must already be allocated and filled with whatever information is targeted for reading. If it is passed a file, that file must

already be open for reading if only reading is desired, and for read-write if writing is desired. In HSL 0.1, the context mode, location or file may be changed at any time.

The VFS kernel module creates its own context at mount time.

### Calling HSL Functions From C

When an HSL spacetype is compiled, it produces a .c file and a .h file. The .h file contains prototypes that allow any of the HSL spacetype's functions to be called from C. An integer function returns a *long long int*; a char function returns a filled-out instance of the following structure—

```
struct clip {  
    char *location;  
    long long int value;  
    long long int length;  
    long long int outputloc;  
    long long int outputminorloc;  
    long long int endian;  
    int isAllocated;  
};
```

—and a list function returns a pointer to a linked list structured as follows:

```

struct cliplist {
    void *c;
    struct cliplist *next;
};

```

In non-pathological cases, *c* will be a *struct clip*.

Clips returned from HSL functions are currently safe to access until the calling program initiates garbage collection (described below); this may change in future revisions of HSL. Two translator functions exist for the *clip* structure:

```

struct clip hsl_CORE_stringtoclip(char *s);
char *hsl_CORE_cliptostring(struct clip c);

```

*hsl\_CORE\_cliptostring* returns an allocated string that the calling program should free when it is done with it. The result of *hsl\_CORE\_stringtoclip* is safe to use until garbage collection, and the calling program must *not* free it.

Functions are given the C names *hsl\_<sname>\_function\_<fname>* where *sname* is the name of the spacetype and *fname* is the name of the function. Hence the *nthBlock* function in the *fat12* module would be *hsl\_fat12\_function\_nthBlock*. Function parameters occur in the same order as they do in their HSL files, and are translated according to the same rules as function types.

Non-public HSL variables *cannot* be directly referenced from C; they must always be wrapped in functions.

## Writing Out HSL Results

List-type HSL functions may return lists meant to be committed to the underlying space.

To commit the results, the following function is provided:

```
void hsl_CORE_write_clips(struct cliplist *l);
```

This function *will write the committed changes to disk* if the HSL runtime is operating in file context mode; it should not be called unless this behavior is desired!

## Garbage Collection

HSL shares with most functional languages the need to perform garbage collection; it allocates temporary storage on the heap when running in user space, and in kernel space when running as a VFS module. The calling program determines when the HSL runtime library garbage collects.

```
void hsl_CORE_garbagecollect(void);
```

*hsl\_CORE\_garbagecollect* is inexpensive, so should be called whenever the calling program has copied everything it needs from an HSL result.

## Adding a new File System to the Extractor

Compiling an extractor for a new file system is straightforward:

- Create an HSL file system module for the new file system.
- Edit **test-temp.c** to include the header file for, and to register, the new file system.

- Create a copy of **test-ext2temp.c** renamed for the new file system; e.g., **test-newfstemp.c**. Edit this copy appropriately to replace **ext2** and **ext2disk.in** with appropriate values.
- Edit the **Makefile**. Copy and edit the section for **ext2.o** to create a new section appropriate to the new file system. Also add its object files to the section for **fs.a**.
- Run **make** to rebuild all the extractors.

### Adding a new File System to the VFS Module

Compiling a new VFS module is less straightforward than compiling a new extractor and more dangerous in terms of system stability. A basic understanding of Linux device driver development and compilation is required. Given that, the steps are as follows:

- Create an HSL file system module for the new file system.
- Verify that the HSL file system module works correctly by testing it several times in the extractor. *An invalid file system module will panic the kernel, leading to a probable system crash and possible loss of data if the filesystem has not recently been synchronized.*
- Create copies of **linux-ext2.c**, **makekernelmodule-ext2.sh** and **trykernelmodule-ext2.sh** for the new file system. Edit them appropriately. In **linux-<newfs>.c**, add the header for the new file system, register it and change the **get\_hsl\_hadleyfs\_implementation** call to reference it instead of **ext2**. Editing the **makekernelmodule** and **trykernelmodule** scripts requires familiarity with device driver compilation under the host Linux distribution.

- Use the new **makekernelmodule** script to create the kernel module.

### **A Note on Concurrency**

HSL currently contains no support for concurrency or locking of any kind. Attempting to access a file system with HSL and any other method, including HSL, at the same time leads to completely undefined behavior.

We will discuss additional implications of concurrency for HSL below, as we discuss future work.

### **Design Notes: The HSL Runtime**

Having explored the use of the HSL runtime, it is worth commenting on its structure.

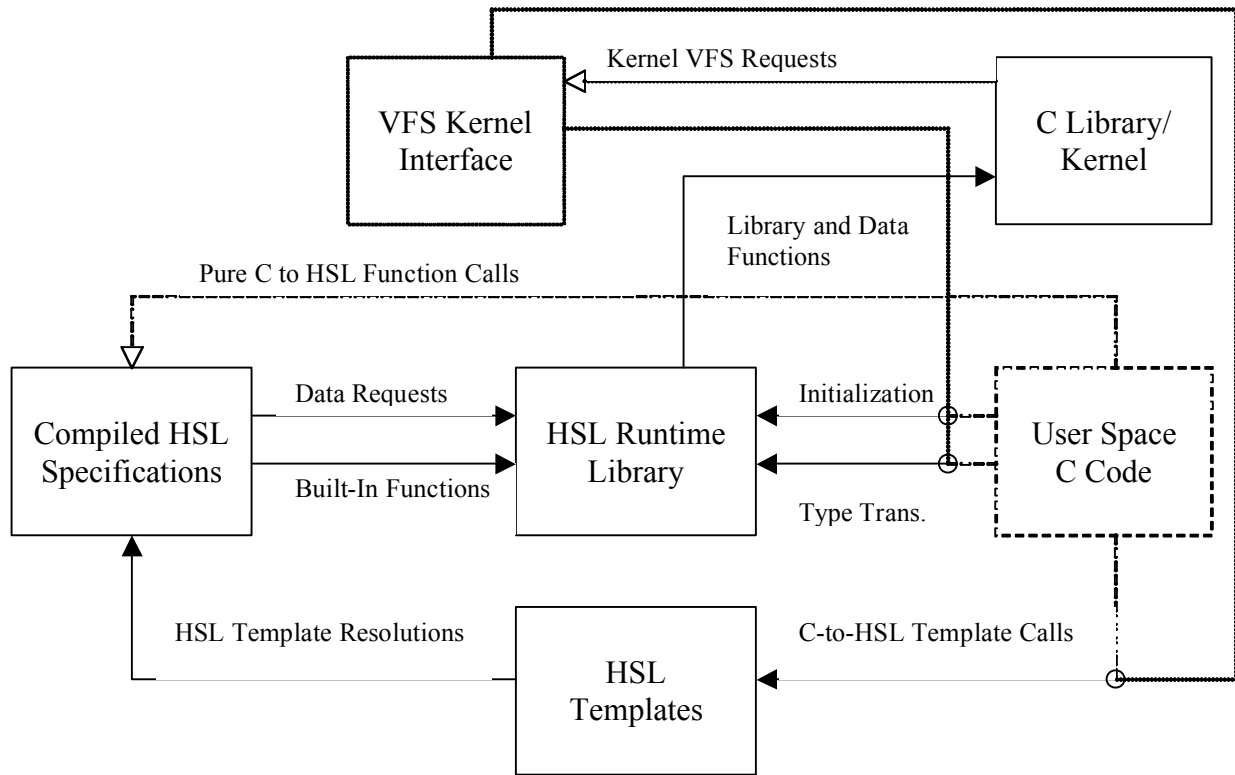


Figure 5: The HSL System Design

Figure 5 shows the design of the initial HSL system, both in VFS kernel module mode and in user space mode. In all modes, compiled HSL specifications interface with the HSL runtime library to request data and call built-in functions. The runtime in turn interfaces with the system's C library or directly with the kernel, depending on whether it is in user space or kernel space.

Dashed paths show unique user space functionality and interfaces. User space C code can call HSL specifications directly, which the VFS kernel module does not do.

Dotted paths show unique kernel mode functionality and interfaces. The kernel can call the VFS kernel interface to answer VFS device driver requests from applications.

In either user space or kernel space, the C code or VFS interface calls the HSL runtime library to initialize it and perform any type translations, and other than that uses the HSL template libraries as interfaces to reach the compiled HSL template specifications.

### **Demonstrating the Correctness of HSL Specifications**

#### **The hadleyfs template**

The hadleyfs template defines the essential functions for reading data from ordinary hierarchical file systems; that is, plain sequential files stored within one or more directories, inclusive of a root directory. More precisely:

*Definition (ordinary hierarchical file system):* An ordinary hierarchical file system is a tree in which:

1. Each node other than the root has an associated name.
2. Each node may be a Directory node, a File node, or of unspecified type unique to the file system.
3. Each File node has an associated regular sequential file.
4. Only Directory nodes may be nonleaves.
5. The root is a Directory node.

The third type of node in the second bullet of the definition exists to encapsulate types of file entries, such as symbolic links, that exist in modern file systems but that we do not cover formally here.



The hadleyfs template is built around the concept of unique identifiers for nodes called File IDs. A File ID must be a value that contains enough information for the specification to uniquely identify and locate that file. For FAT12 and ext2fs, a single integer suffices. We refer to File IDs as Directory IDs when they are already known to refer to directory nodes. Every Directory ID is also a File ID.

The hadleyfs template consists of nine functions:

- RootDirectoryID() returns the File ID of the root node.
- FilesIn(), given a Directory ID, returns the count of its node's children.
- nthFileIn(), given a Directory ID *d* and an integer *n*, returns the File ID of the *n*th child of *d*'s node.
- FileIsDirectory(), given a File ID, returns true iff its node is a Directory node.
- FileIsRegular(), given a File ID, returns true iff its node is a File node.
- FileName(), given a valid File ID other than the root, returns its node's associated name.
- FileSize(), given the File ID of a File node, returns its size in bytes.
- BlockSize() returns the fundamental block size of the file system in bytes.
- FileBlock(), given the File ID *f* of a valid sequential file and an integer *n*, returns the *n*th block of *f*'s node's associated sequential file; of size BlockSize() if the *n*th block is not the last block, and of size in the interval [1..BlockSize()] if it is.

We want to demonstrate that these functions are sufficient to read the name of every directory, the name of every regular sequential file, and the data of every regular sequential file in an ordinary hierarchical file system.

To show that we can visit any node and find its associated name, we observe that:

- An ordinary hierarchical file system is a tree; hence, any successful traversal of the tree will find each of its nodes.
- The hadleyfs template provides for successful traversal, as follows:
  - FileIsDirectory() and FilesIn() together determine leaves versus nonleaves. A node with File ID  $f$  is a nonleaf iff  $\text{FileIsDirectory}(f)$  and  $\text{FilesIn}(f) > 0$ .
  - RootDirectoryID() provides the File ID for the root node.
  - FilesIn() and nthFileIn() inductively provide the File IDs for the child nodes of any directory; and hence any nonleaf, since all nonleaves are directories.
  - FileName() provides the associated name of any nonroot node.

To show that we can retrieve the associated sequential files of all File nodes, we observe that:

- From above, we can visit any node.
- FileIsRegular() determines nodes with associated sequential files.
- FileSize() provides the size of a node's associated sequential file.
- A node's sequential file has  $\text{FileSize()} \bmod \text{BlockSize()}$  blocks.
- FileBlock() provides each of these blocks, randomly or in turn.

*Conclusion (1):* The functions provided by the hadleyfs template are sufficient to read the name of every directory, the name of every regular sequential file, and the data of every regular sequential file in an ordinary hierarchical file system.

*Conclusion (2):* Given correct operation of the functions provided by the hadleyfs template, the name of every regular sequential file, and the data of every regular sequential file in an ordinary hierarchical file system can be correctly read.

It follows that a file system specification which implements the hadleyfs template is shown to allow correct reading of its file system if it is shown that its implementation of each hadleyfs function is correct. A template for such demonstrations follows in turn.

### **Demonstrating the Correctness of hadleyfs implementations**

All indexes begin at zero, and in all cases we assume that a properly-operating system is operating upon a valid file system of the defined type.

#### **Fundamental Functions**

**Assertion FS-1:** BlockSize() correctly returns the file system's fundamental block size. (This is not the same thing as the sector size of the underlying media.)

#### **Directory Functions**

Two paths can be taken for the directory-related functions, depending on how the root directory is dealt with in a particular file system.

### ***Path 1***

**Assertion FS-2a:** RootDirectoryID() correctly returns the Directory ID of the root directory.

**Assertion FS-3a:** FilesIn() correctly returns the number of files in the directory with the DirectoryID passed to it.

**Assertion FS-4a:** nthFileIn() correctly returns the File ID of the nth file in the directory with the DirectoryID passed to it.

**Assertion FS-5a:** FileIsDirectory() correctly returns a nonzero value iff the File ID passed to it in FileID is the File ID of a directory.

### ***Path 2***

**Assertion FS-2b:** RootDirectoryID() returns a unique "magic number" for the root directory that cannot otherwise refer to a valid directory or file.

**Assertion FS-3b:** FilesIn() correctly returns the number of files in the root directory if passed the "magic number" for the root directory in DirectoryID, and otherwise correctly returns the number of files in the directory with the DirectoryID passed to it.

**Assertion FS-4b:** nthFileIn() correctly returns the File ID of the nth file in the root directory if passed the "magic number" for the root directory in DirectoryID, and otherwise correctly returns the File ID of the nth file in the directory with the DirectoryID passed to it.

**Assertion FS-5b:** FileIsDirectory() correctly returns a nonzero value iff the File ID passed to it in FileID is the "magic number" for the root directory or otherwise is the File ID of a directory.

## **File Functions**

**Assertion FS-6:** FileIsRegular() correctly returns a nonzero value iff the File ID passed to it in FileID is the File ID of a regular sequential file.

**Assertion FS-7:** FileSize() correctly returns the size of the regular file whose File ID is passed to it in FileID.

**Assertion FS-8:** FileName() correctly returns the name of the regular file or non-root directory whose File ID is passed to it in FileID.

**Assertion FS-9:** FileBlock() correctly returns the BlockNumth block of data of the regular file whose File ID is passed to it in FileID.

## **CHAPTER FOUR: FINDINGS**

### **HSL Test Suite**

There are two essential platforms for testing HSL: the extractor and the universal file system driver.

The extractor is a utility that exhaustively traverses a file system stored in an image file, and reproduces each directory and regular file in an output directory.

The universal file system driver is a shell that, compiled and linked with the object files produced by the HSL system, provides a Linux kernel Virtual File System (VFS) interface to HSL. Creating versions for variant file systems requires only changing one constant in the driver source file and recompiling; future versions will permit accessing multiple file systems with no recompilation necessary at all.

The Second Extended File System module written in HSL 0.1 also contains facilities for reading file slack space. HSL is not yet able to distinguish among RAM slack (the slack between the end of a file and the end of its last sector) and file system slack (the slack between the end of a file's last sector and the end of its last block); it treats all slack equally.

### **Filesystem Extractors**

The compiled FAT12 extractor was used to examine the FreeDOS Installation Boot Disk image as downloaded directly from the Internet. (Hall) It successfully read all relevant metadata from the disk image and properly extracted all files. We tested text files against their versions

extracted using ordinary operating system methods through inspection and binary files (including one large fragmented file) through the UNIX **diff** command.

The compiled Second Extended File System extractor was used to examine two images: a root image from User Mode Linux and a much smaller root image from Romanian Mini Linux with the /dev directory removed. (User Mode Linux, Anton) It successfully read all relevant metadata from both images and properly extracted all files. We tested text files against their versions extracted using ordinary operating system methods through inspection and binary files through the UNIX **diff** command.

It should be noted that the extractor is a currently usable, albeit slow, tool. It extracts all legal regular files from a filesystem and benefits from the demonstrations given later in this document; the core means by which it interprets filesystems have been shown to be correct. The extractor will eventually be offered as an experimental forensic investigation tool.

### **Universal Filesystem Driver**

HSL universal file system drivers in FAT12 and ext2fs variants for Linux were used to examine all three of the images that the extractors were run on. They correctly read files and essential metadata where kernel limitations were not encountered. We tested files against their versions extracted using ordinary operating system methods through inspection and through the UNIX **diff** command.

The Linux kernel has an enforced small stack size with no provided way to change it – 4K in most 32-bit configurations, 8K in most 64-bit configurations. Large O(n) recursive

operations quickly become untenable given this operating system limitation. This is the controlling finding for the eventual need to include iteration in HSL.

### **Writing Filesystems**

The compiled FAT12 module written in HSL 0.1 was used to add directories and files to the FreeDOS Installation Boot Disk image. It successfully created and re-read directories and files with essential associated metadata – the file name, extension, size, attribute block and cluster chains. Dates are not handled by the initial version of HSL, but are set to a valid zero with respect to MS-DOS’s epoch (January 1, 1980).

### **Specification Efficiency**

The HSL module for ext2fs is approximately 16K: around 12% the size of the .C files in the ext2fs subsystem in Linux as of kernel version 2.4.26 (Torvalds). This is not an apples-to-apples comparison: HSL does not currently support file system writing or some ext2 advanced features. We are confident that these additions will increase the size of the code far less than the over 800% necessary to catch up with Linux, and that the code will remain more readable.

Table 1: ext2fs specification efficiency

	Lines	Words	Characters
ext2.hsl	423	1756	15971
Linux ext2 total	4521	15957	128186



The results for FAT are both more and less accurate: Hadley does not yet support FAT formats other than FAT12, and FAT lacks support for most modern file system features such as permissions. However, the Hadley module for FAT12 supports limited writeback, so the complexity of the Hadley module is close to what would be necessary to support the rest of Linux's FAT12 features.

Table 2: FAT12 specification efficiency

	Lines	Words	Characters
fat12.hsl	505	1773	17861
Linux FAT total	3263	10412	85799

### **Performance**

The comparative performance of the Hadley extractor and universal filesystem driver versus its Linux kernel counterparts are shown below. As expected, the Linux kernel's filesystem interface retains a dominant performance advantage. Much optimization will be necessary before HSL may see practical use directly in system software.

Table 3: Performance results

	HSL extraction utility	HSL Linux driver	Linux
FAT12 list	N/A	3.789s	.006s
FAT12 copy	6.7s	Stack Overflow	.019s
small ext2 list	N/A	.441s	.004s
small ext2 copy	.143s	.140s	.016s
large ext2 copy	16.7s	Stack Overflow	.039s

### **Characteristics of HSL and HSL Demonstrations**

The YACC grammar for HSL is presented in Appendix C. Simple inspection shows that it is a context-free language.

Since HSL is a purely functional language, direct demonstrations may be used.

### **Demonstrations of Correctness**

#### **Major Dependencies**

In our demonstrations of correctness, we assume the correct operation of the HSL runtime library and the underlying C runtime libraries. The empirical steps we took to verify correct operation of the HSL runtime library were described above. The most critical dependencies on the C runtime library are the basic memory functions **malloc**, **free**, **memcpy** and **memcmp**; the string library functions, particularly **strlen**; and the file functions **fread**,

**fwrite** and **fseek**. The Linux VFS kernel module is additionally dependent on **kmalloc**, **kfree**, **bread** and **brelse** for management of kernel space memory and direct reading of block devices.

## **Definition of FAT12**

Primary sources for this section include (Kjoernes 2000) and (Microsoft).

### **Overview**

FAT12 is an ordinary hierarchical file system with the characteristics described here. All integers are little-endian.

### **Partition Metadata**

The FAT12 filesystem has three partition-level metadata structures, one of which encloses another.

## *The Boot Sector*

Table 4: The FAT12 Boot Sector

Object	Length	Offset
Jump Instruction	3	0
OEM Name	8	3
BIOS Parameter Block	25	11
Extended BPB	26	36
Bootstrap Code	448	62
End of Sector Marker	2	510

The Jump Instruction is an artifact that points to the Bootstrap Code; the OEM Name is intended to store the operating system that created the partition; and the End of Sector Marker is a constant. The BIOS Parameter Block is what we are interested in; it is structured as follows.

## ***BIOS Parameter Block***

Table 5: The FAT12 BIOS Parameter Block

Object	Length	Offset
Bytes per Sector	2	0
Sectors per Cluster	1	2
Reserved Sectors	2	3
Number of FATs	1	5
Root Entries	2	6
Sectors in Volume	2	8
Media Type	1	10
Sectors per FAT	2	11
Sectors per Track	2	13
Heads per Cylinder	2	15
Hidden Sectors	4	17
Sectors in Volume	4	21

Bytes per Sector indicates the number of sectors on the underlying media. For most non-exotic hardware, this is 512.

Sectors per Cluster indicates the number of sectors per fundamental block. The fundamental block size is the bytes per sector multiplied by the sectors per cluster.

Reserved Sectors indicates the number of sectors from the Partition Boot Sector to the start of the first FAT. It is guaranteed to be at least 1.

The file system stores some number of copies of the FAT for redundancy. Typically, two are stored.

The root directory in FAT12 is at a fixed location and permits a constant number of entries.

There are two different locations for the number of sectors in a volume: a 2-byte and 4-byte unsigned integer. If the number of sectors is less than or equal to 65,535, it will be stored in the 2-byte integer; otherwise, 0 will be stored in the 2-byte integer and the real value will be stored in the 4-byte integer.

The media type, sectors per track and heads per cylinder refer to hardware information about the underlying media. Hidden Sectors refers to the partition's physical location on the underlying media.

For completeness, we briefly describe the Extended BIOS Parameter Block as well.

### ***Extended BIOS Parameter Block***

Table 6: The FAT12 Extended BIOS Parameter Block

Object	Length	Offset
Physical Disk Number	1	0
Current Head	1	1
Signature	1	2
Volume S/N	4	3
Volume Label	11 (String)	7
System ID	8 (String)	18

The physical disk number refers to underlying hardware. The current head is unused in the FAT filesystem. The signature byte is required to be either 28h or 29h hexadecimal. The volume serial number is a unique number created when the partition is formatted.

The volume label is usually stored in a file rather than here, but it is legal to store it here. If it is stored here, it should be padded with spaces.

The system ID is required to be "FAT12", padded out to eight characters with spaces.

### **Directory Metadata**

Directories in the FAT12 file system are juxtaposed lists of directory entries. The root directory of the FAT12 filesystem is always located immediately following the last FAT and has a fixed number of entries; this means it is limited in size. Subdirectories are located by directory entries, and are treated as ordinary files, so may consist of more than one block.

Directory entries have the following structure.

## *Directory Entry*

Table 7: FAT12 Directory Entries

Object	Length	Offset
Name	8	0
Extension	3	8
Attribute	1	11
Reserved	1	12
VFAT Creation msec/10	1	13
VFAT Creation Time	2	14
VFAT Creation Date	2	16
VFAT Access Date	2	18
FAT32 High Cluster #	2	20
Update Time	2	22
Update Date	2	24
First Cluster #	2	26
File Size	4	28

The required file name and optional extension are padded to their maximum length with spaces. The attribute byte is described below. The high cluster number is relevant only for FAT32 file systems, listed here only to clarify that that space is now filled. The time and date fields are self-explanatory; several of them are only present in VFAT revisions of the file system. The file size is a 32 bit unsigned integer.



The first character of the name may have one of three special values, as well as legal characters: 00h to indicate that neither this entry nor any following entry are in use, 05h to indicate that the first character of the name is actually E5h, or E5h to indicate that the entry has been erased and is available for use.

The first cluster points both to the file's first cluster of data and to the beginning of the file's chain in the FAT.

### ***Attribute Byte***

The attribute byte is packed as follows:

Table 8: FAT12 Attribute Bytes

Read-Only	1bit	0bits
Hidden	1bit	1bits
System	1bit	2bits
Volume	1bit	3bits
Directory	1bit	4bits
Archive	1bit	5bits

Read-Only and Hidden attributes are self-explanatory. System files are special files that should not be tampered with by user-mode programs. The Volume file is the volume name for the partition. Directories always have a file size of zero, but have cluster chains containing their directory entries.

### **The FAT, Clusters and the Data Area**

There are several identical FAT structures, located one after the other beginning after the Reserved Sectors as specified in the BIOS Parameter Block. Each FAT is an array of twelve-bit integers arranged in little-endian fashion.

The 0th and 1st FAT entries are reserved. The remaining entries have values as follows:

Table 9: FAT12 Reference Values

Value	Description
000	Available (empty)
001	Reserved
002-FF6	Next Cluster
FF7	Bad Cluster
FF8-FFF	Last Cluster

Hence, cluster chains can be traversed inductively as follows:

- The first cluster of a file has the cluster number given by the First Cluster number in its directory entry.
- For the  $n$ th cluster of a file with cluster number  $k$ , if  $FAT[k]$  is:
  - 002 through FF6, then the  $n+1$ th cluster has cluster number  $FAT[k]$ .
  - FF8 through FFF, then the  $n$ th cluster is the final one.

## ***The Data Area***

The actual file data area begins immediately following the end of the root directory. Cluster number 2 begins at this offset, and the clusters proceed from there in linear order.

## **Demonstration of Correctness for FAT12**

All variables and functions are with respect to the HSL **fat12** spacetype, unless noted otherwise. Correct operation of the Hadley system and its built-in functions is assumed at all times. All division is assumed to be integer.

### **Definition (FAT12 File ID)**

The File ID for the HSL **fat12** spacetype is either the offset of the directory entry of the associated file from the start of the partition or the number zero, indicating the root directory.

### **Assertion FAT12-a**

BytesPerSector, SectorsPerBlock, ReservedSectors, NumberOfFats, MaxRootDirEntries, SectorsInPartition, and SectorsPerFAT specify, respectively, the bytes per sector, sectors per cluster, reserved sectors, number of FATs, maximum root directory entries, number of sectors in the partition, and sectors per FAT in a FAT12 file system.

```
var BytesPerSector    is integer littleend width 16 at 11.
var SectorsPerBlock   is integer littleend width 8  at 13.
var ReservedSectors   is integer littleend width 16 at 14.
var NumberOfFats       is integer littleend width 8  at 16.
var MaxRootDirEntries  is integer littleend width 16 at 17.
var SectorsInPartition is integer littleend width 16 at 19.
var SectorsPerFAT      is integer littleend width 16 at 22.
```

### **Demonstration**

By the definition of FAT12, the BIOS parameter block occurs at offset 11 in the partition. The bytes per sector, sectors per cluster, reserved sectors, number of FATs, maximum root directory entries, number of sectors in the partition, and sectors per FAT occur at offsets 0, 2, 3, 5, 6, 8 and 11 in the BIOS parameter block respectively, and are 2, 1, 2, 1, 2, 2 and 2 bytes wide respectively. Arithmetic places them at offsets 11, 13, 14, 16, 17, 19 and 22 respectively. Simple inspection then demonstrates the equivalence of the variable specifications with their corresponding locations in the FAT12 structure. □

### **Assertion FAT12-b**

With regard to the **fat12dirent** spacetype, `FileNameFirstChar`, `FileName`, `FileExt`, `FileIsReadOnly`, `FileIsHidden`, `FileIsSystem`, `FileIsVolumeLabel`, `FileIsDirectory`, `FileIsArchive`, `FirstBlock` and `FileSize` specify, respectively, the first character of the file name, the space-padded file name, the space-padded file extension, the read-only attribute flag, the hidden attribute flag, the system attribute flag, the volume label attribute flag, the directory attribute flag, the archive attribute flag, the first cluster number, and the file size field of a directory entry.

var FileNameFirstChar	is integer littleend width 8 at 0.
var FileName	is char[8] at 0.
var FileExt	is char[3] at 8.
var FileIsReadOnly	is public integer littleend width 1 at 11 minor 0.
var FileIsHidden	is public integer littleend width 1 at 11 minor 1.
var FileIsSystem	is public integer littleend width 1 at 11 minor 2.
var FileIsVolumeLabel	is public integer littleend width 1 at 11 minor 3.
var FileIsDirectory	is public integer littleend width 1 at 11 minor 4.
var FileIsArchive	is public integer littleend width 1 at 11 minor 5.
var FirstBlock	is public integer littleend width 16 at 26.
var FileSize	is public integer littleend width 32 at 28.

### **Demonstration**

For FileName, FileExt, FirstBlock and FileSize, by the definition of FAT12, the space-padded file name and extension, the first cluster number, and the file size occur at offsets 0, 8, 26 and 28 in a directory entry respectively, and are 8, 3, 2 and 4 bytes wide respectively. Simple inspection demonstrates their equivalence with the variable specifications.

For FileNameFirstChar, by the definition of a string, the first character of a string is a single character collocated with the string itself. Hence the first character of the file name is a single character collocated with the file name. By the definition of FAT12, the space-padded file

name occurs at offset 0 in a directory entry, and hence its first character also occurs at offset 0.

Simple inspection demonstrates equivalence with the `FileNameFirstChar` specification.

For the attribute flags, by the definition of FAT12, the attribute byte occurs at offset 11 in a directory entry, and the read-only, hidden, system, volume label, directory and archive bits are the 0<sup>th</sup>, 1<sup>st</sup>, 2<sup>nd</sup>, 3<sup>rd</sup>, 4<sup>th</sup> and 5<sup>th</sup> bits of that byte respectively. Simple inspection demonstrates equivalence with the `FileIsReadOnly`, `FileIsHidden`, `FileIsSystem`, `FileIsVolumeLabel`, `FileIsDirectory` and `FileIsArchive` bits. □

### **Assertion FAT12-c**

`RootDirectorySize()` returns the size in bytes of the root directory.

```
function RootDirectorySize is integer() {  
    @sys.mul($MaxRootDirEntries, 32)  
}
```

### **Demonstration**

By the definition of FAT12, the root directory is as large as its maximum number of entries times the size of a directory entry. By inspection of the definition of FAT12, a directory entry is 32 bytes long. By assumed proper operation of `@sys.mul`, **`RootDirectorySize()`** returns **`MaxRootDirEntries`** times 32; from the above and Assertion FAT12-a, this is the size of the root directory. □

### **Assertion FAT12-d**

`RootDirectoryStart()` returns the offset of the root directory from the beginning of the partition.

```
function RootDirectoryStart is integer() {  
    @sys.mul(@sys.add($ReservedSectors,  
                      @sys.mul($NumberOfFats, $SectorsPerFAT)),  
            $BytesPerSector)  
}
```

### **Demonstration**

By the definition of FAT12, the root directory begins immediately after the final FAT. Also by the definition of FAT12, the FATs begin immediately after the reserved sectors. Hence by arithmetic the root directory begins at sector  $n$  where  $n$  is the number of reserved sectors plus the total number of sectors taken by the FATs. By arithmetic the number of sectors taken by the FATs is the number of FATs times the number of sectors per FAT.

By Assertion FAT12-a and assumed proper operation of `@sys.mul` and `@sys.add`, **RootDirectoryStart()** returns  $k + lm * n$ , where  $k$  is the number of reserved sectors,  $l$  is the number of FATs,  $m$  is the sectors per FAT and  $n$  is the bytes per sector, as desired.  $\square$

### **Assertion FAT12-e**

`data()` returns the offset  $k$  characters from the beginning of the partition's data area.

```
function data is integer(k is integer) {  
    @sys.add(k, @sys.add(@this.RootDirectoryStart(),  
        @this.RootDirectorySize()))  
}
```

### **Demonstration**

By Assertions FAT12-c and FAT12-d and assumed proper operation of **@sys.add**, **data** returns  $k + l + m$  where  $l$  is the offset of the root directory from the beginning of the partition and  $m$  is the size of the root directory. By the definition of FAT12 the data area begins immediately after the root directory; hence, at  $l + m$ . Hence **data( $k$ )** returns the offset  $k$  characters from the beginning of the partition's data area.  $\square$

### **Assertion FAT12-f**

**DataBlocks()** returns the number of blocks in the data area.



```

function NonDataSectors is integer() {
    @sys.div(@this.data(0), $BytesPerSector)
}

function DataSectors is integer() {
    @sys.sub($SectorsInPartition, @this.NonDataSectors())
}

function DataBlocks is integer() {
    @sys.div(@this.DataSectors(), $SectorsPerBlock)
}

```

### **Demonstration**

By Assertions FAT12-e and FAT12-a, and assumed proper operation of **@sys.div**, **NonDataSectors()** returns the offset of the data area divided by the size of each sector. Since by definition of FAT12 the data area is required to begin at a sector boundary, this value is the number of sectors before the data area.

By the above, Assertion FAT12-a, and assumed proper operation of **@sys.sub**, **DataSectors()** returns the number of sectors in the partition minus the number of sectors before the data area. By arithmetic, this is the number of sectors in the data area.

By the above, Assertion FAT12-a, and assumed proper operation of **@sys.div**, **DataBlocks()** returns the number of sectors in the data area divided by the number of sectors per block. By arithmetic, this is the number of blocks in the data area.  $\square$

### **Assertion FAT12-g**

FAT[k] is the  $k$ th entry in the first FAT of the partition.

```

var FAT is integer[@sys.add(@this.DataBlocks(), 2)]

    littleend width 12 at

    @sys.mul($ReservedSectors, $BytesPerSector).

```

### **Demonstration**

By the definition of FAT12, the first FAT is an array of 12-bit integers in little-endian format, one for each block in the data area plus two reserved entries at the beginning, occurring immediately after the reserved sectors.

By inspection, and by assumed proper operation of variable declarations, **@sys.add** and **@sys.mul**, FAT is an array of **DataBlocks() + 2** 12-bit integers in little-endian format, occurring at **ReservedSectors × BytesPerSector**. By arithmetic and Assertion FAT12-a, FAT occurs immediately after the reserved sectors; by arithmetic and Assertion FAT12-f, FAT is an array of one integer for each block in the data area plus two more. □

### **Assertion FAT12-h**

**FirstBlock()**, passed a File ID of a regular file or a directory other than the root, correctly returns the first block number of the file associated with the File ID.

```

function FirstBlock is integer(FileID is integer)

    with entry is fat12dirent at FileID

    {

        @entry.FirstBlock()

    }

```

By the definition of a FAT12 File ID, unless it is the root directory’s “magic number”, **FileID** is the offset of the directory entry of its associated file from the beginning of the partition. On a valid call of **FirstBlock()**, **FileID** cannot refer to the root. Hence by assumed proper

operation of **with**, **entry** is the directory entry associated with **FileID**. By assumed proper operation of function indirection it now suffices to show that **fat12dirent**'s **FirstBlock()** function returns the first block of the file as stored in the directory entry.

```
var FirstBlock is public integer littleend width 16 at 26.
```

By assumed proper operation of a public integer, **fat12dirent**'s **FirstBlock()** function returns the value of the **FirstBlock** variable. Hence by Assertion FAT12-b, it returns the first cluster number, i.e., the first block number.  $\square$

### Assertion FAT12-i

**NextBlock()**, passed a block number, correctly returns the next block number in the cluster chain from the FAT or 0 if there is no next block number.

```
function NextBlock is integer(k is integer) {  
    if(@sys.and(@sys.lst($FAT[k], 4087),  
        @sys.grt($FAT[k], 1))) { $FAT[k] } else { 0 }  
}
```

### Demonstration

By the definition of FAT12, the next block number in a cluster chain from a given block number  $k$ , if that cluster chain exists, is the  $k$ th entry of the FAT. A number between FF8 and FFF indicates that the current block number is the end of the cluster chain, and numbers 0, 1 and FF7 are used for system purposes; all other values are valid next block numbers. Hence it suffices to show that **NextBlock()** returns the  $k$ th entry of the FAT if that  $k$ th entry is less than FF7 (4087) and greater than 1; otherwise return 0.

By assumed proper operation of **if/else**, **@sys.and**, **@sys.lst** and **@sys.grt**, **NextBlock()** returns **FAT[k]** if  $1 < \text{FAT}[k] < 4087$ , 0 otherwise.

By Assertion FAT12-g, **FAT[k]** is the  $k$ th entry of the FAT.

Hence **NextBlock()** returns the  $k$ th entry of the FAT if that  $k$ th entry is less than FF7 (4087) and greater than 1; otherwise it returns 0.  $\square$

### **Assertion FAT12-j**

**nthBlock()**, passed a non-root File ID and a number  $n$ , correctly returns the block number of the  $n$ th block in the associated File ID's cluster chain.

```
function nthBlock is integer(FileID is integer, n is integer) {
    if(@sys.equ(n, 0)) { @this.FirstBlock(FileID) }
    else { @this.NextBlock(@this.nthBlock(FileID,
                                @sys.sub(n, 1))) }
}
```

### **Demonstration**

Consider  $n = 0$ . Then by assumption of proper operation of **if** and **@sys.equ**, **nthBlock()** returns **FirstBlock(FileID)**, which by Assertion FAT12-h is equal to the first (i.e., zeroeth) block number in **FileID**'s associated file's cluster chain.

Choose  $n \geq 1$  and assume that **nthBlock(FileID,  $n-1$ )** returns the  $n-1$ th block number in **FileID**'s associated file's cluster chain. Then by assumption of proper operation of **else**, **@sys.equ** and **@sys.sub**, **nthBlock()** returns **NextBlock(nthBlock(FileID,  $n-1$ ))**. By the induction hypothesis this is **NextBlock()** called on the  $n-1$ th block number in **FileID**'s associated

file's cluster chain; by Assertion FAT12-i, this is the  $n$ th block number in **FileID**'s associated file's cluster chain.  $\square$

### **Assertion FAT12-1**

BlockSize() correctly returns the file system's fundamental block size.

```
function BlockSize is integer() { @sys.mul($BytesPerSector,  
                                           $SectorsPerBlock) }
```

### **Demonstration**

By the definition of FAT12, the sectors per cluster value indicates the number of sectors per fundamental block. Hence by assertion FAT12-a, BytesPerSector specifies the number of bytes per sector and SectorsPerBlock specifies the number of sectors per fundamental block. Hence by multiplication,  $(\text{BytesPerSector} \times \text{SectorsPerBlock})$  is equal to the fundamental block size. Hence by assumed correct operation of **@sys.mul**, BlockSize() returns the fundamental block size.  $\square$

### **Assertion FAT12-2b**

RootDirectoryID() returns a unique "magic number" for the root directory that cannot otherwise refer to a valid directory or file.

```
function RootDirectoryID is integer() { 0 }
```

### **Demonstration**

By the definition of FAT12, the boot sector occurs at offset 0. Hence a valid directory entry cannot occur at 0. Hence by the definition of the FAT12 File ID, 0 cannot be a valid File ID. By trivial syntax, RootDirectoryID() always returns 0. Hence RootDirectoryID() returns a unique “magic number” for the root directory that cannot otherwise refer to a valid directory or file. □

### **Assertion FAT12-k**

DirEntsPerBlock() returns the number of directory entries per block.

```
function DirEntsPerBlock is integer() {  
    @sys.div(@this.BlockSize(), 32)  
}
```

### **Demonstration**

By inspection of the definition of FAT12, a directory entry is 32 bytes long. By Assertion FAT12-1, **BlockSize()** is the block size. Hence by assumed proper operation of **@sys.div**, **DirEntsPerBlock()** returns the block size divided by the length of a directory entry, as desired. □

### **Assertion FAT12-m**

DirectoryEntryLoc() returns the offset of the nth entry in the directory whose ID is passed to it in DirectoryID.

```

function DirectoryEntryLoc is integer(DirectoryID is integer,
                                     n is integer) {

    if(@sys.equ(DirectoryID, 0)) {

        @sys.add(@this.RootDirectoryStart(), @sys.mul(n, 32))

    }

    else { @this.data(@sys.add(

        @sys.mul(

            @sys.sub(

                @this.nthBlock(

                    DirectoryID,

                    @sys.div(n, @this.DirEntsPerBlock()),

                    2),

                @this.BlockSize()),

            @sys.mul(@sys.mod(n, @this.DirEntsPerBlock()), 32)

        )) }

    }
}

```

### **Demonstration**

By the definition of a FAT12 Directory ID, there are two cases.

#### **Case 1: DirectoryID is 0.**

By the definition of a FAT12 Directory ID, it suffices to show that **DirectoryEntryLoc()** returns the offset of the **n**th entry in the root directory.

**DirectoryID** is 0. Hence by assumed proper operation of **if** and **@sys.equ**, **DirectoryEntryLoc()** returns **@sys.add(@this.RootDirectoryStart(),@sys.mul(n, 32))**.

By Assertion FAT12-d, **RootDirectoryStart()** returns the offset of the root directory in the partition.

Hence by assumed proper operation of **@sys.add** and **@sys.mul**, **DirectoryEntryLoc()** returns the offset  $32n$  bytes into the root directory.

By the definition of FAT12, a directory is an array of juxtaposed directory entries 32 bytes long.

Hence **DirectoryEntryLoc()** returns the offset of the  $n$ th entry of the root directory in the partition.

**Case 2: DirectoryID is the offset of a directory's directory entry in the partition.**

By the definition of FAT12, a nonroot directory is an array of juxtaposed directory entries 32 bytes long, stored otherwise as a sequential file.

By arithmetic and by the above, the  $n$ th directory entry occurs  $32n$  bytes into a directory's data.

By definition of a sequential file as stored in FAT12, a directory may contain more than one data block of entries, and each block contains a finite number  $m$  of directory entries. Hence by arithmetic the  $n$ th directory entry occurs in the  $(n/m)$ th data block of the directory, and further, it occurs at offset  $32(n \bmod m)$  in that block.

It now suffices to show that **DirectoryEntryLoc** returns the offset  $32(n \bmod m)$  in the  $(n/m)$ th data block of **DirectoryID**'s associated directory.

By Assertion FAT12-2b, **DirectoryID** cannot be the root directory ID. Hence by assumed proper operation of **else**, **DirectoryEntryLoc()** returns the **else**-enclosed block. By assumed proper operation of **@sys.add**, **@sys.mul**, **@sys.sub** and **@sys.mod**, and by Assertion FAT12-k, we rewrite this as:

$$\mathbf{data( (nthBlock(DirectoryID, n/m) - 2) \times BlockSize() + (n \bmod m) \times 32 ).}$$

By Assertions FAT12-j and FAT12-1, this is



$$\text{data}( (k - 2) \times b + (n \bmod m) \times 32 )$$

where  $b$  is the block size and  $k$  is the cluster number of the  $(n/m)$ th block of **DirectoryID**'s directory.

By the definition of FAT12, the cluster number  $k$  actually refers to the  $k$ -2th cluster, numbered from zero, in the data area. Hence by arithmetic, cluster number  $k$  begins at offset  $(k-2)$  times the block size in the data area.

Hence by the above ad Assertion FAT12-e, **DirectoryEntryLoc()** returns the offset  $(n \bmod m) \times 32$  bytes from the beginning of the  $(n/m)$ th block of **DirectoryID**'s directory.  $\square$

### Assertion FAT12-n

With regard to the **fat12dirent** spacetype, **IsFinalEnt()** correctly returns nonzero iff the directory entry is a final-entry placeholder in its directory, i.e., if no in-use directory entries follow it.

```
function IsFinalEnt is integer() { @sys.equ($FileNameFirstChar,
                                     0) }
```

By the definition of FAT32, a final-entry placeholder in a directory has the null character (0) as the first character of its filename, and no other file may. By Assertion FAT12-b, **FileNameFirstChar** is the first character of the directory entry's filename. By proper operation of **@sys.equ**, **IsFinalEnt()** returns nonzero iff the first character of the directory entry's filename is 0, which from above happens iff it is a final-entry placeholder.  $\square$

### **Assertion FAT12-o**

With regard to the **fat12dirent** spacetype, **IsErased()** correctly returns nonzero iff the directory entry is marked as erased.

```
function IsErased is integer() { @sys.equ($FileNameFirstChar,  
                                         229) }
```

By the definition of FAT32, an erased file has character 229 as the first character of its filename, and no other file may. By Assertion FAT12-b, **FileNameFirstChar** is the first character of the directory entry's filename. By proper operation of **@sys.equ**, **IsErased()** returns nonzero iff the first character of the directory entry's filename is character 229, which from above happens iff it is an erased file. □

### **Assertion FAT12-p**

**DirectoryEntryUsed()** correctly returns nonzero iff the **n**th entry of the directory with the **DirectoryID** passed to it is a used entry, i.e., it is the File ID of either a file or a directory.

```

function DirectoryEntryUsed is integer(DirectoryID is integer,
                                     n is integer)

  with entry is fat12dirent at
    @this.DirectoryEntryLoc(DirectoryID, n)
  {
    if( @sys.or(@sys.equ(@entry.FileAttr(), 15),
                  @sys.or(@entry.IsFinalEnt(),
                        @entry.IsErased())) { 0 }
    else { 1 }
  }

```

By the definition of FAT12, an entry is used if it is neither erased nor a final-entry placeholder. We also consider an entry to be unused if it is a VFAT Long Filename component with a characteristic attribute value of 15.

By assumed proper operation of **with** and Assertion FAT12-m, **entry** is the **n**th directory entry in the directory indicated by **DirectoryID**.

By assumed proper operation of **if/else**, **@sys.or** and **@sys.equ**, and Assertions FAT12-b, FAT12-n and FAT12-o, **DirectoryEntryUsed()** returns nonzero iff the directory entry has an attribute value of 15, is a final-entry placeholder or is marked erased, which occurs iff it is unused. □

### **Assertion FAT12-3b**

**FilesIn()** correctly returns the number of files in the root directory if passed the "magic number" for the root directory in **DirectoryID**, and otherwise correctly returns the number of files in the directory with the **DirectoryID** passed to it.

```

function FilesInStep is integer (DirectoryID is integer, n is
                                integer, count is integer)

    with entry is fat12dirent at

        @this.DirectoryEntryLoc(DirectoryID, n)
    {
        if(@entry.IsFinalEnt()) { count }

        else { @this.FilesInStep(DirectoryID, @sys.add(n, 1),

            if(@this.DirectoryEntryUsed(DirectoryID, n))

                { @sys.add(count, 1) }

            else {count})

        }

    }

function FilesIn is integer(DirectoryID is integer) {

    @this.FilesInStep(DirectoryID, 0, 0)

}

```

First consider **FilesInStep()**. By assumed proper operation of **with** and Assertion FAT12-m, **entry** is the **n**th directory entry in the directory indicated by **DirectoryID**. By assumed proper operation of **if** and Assertion FAT12-n, **FilesInStep()** returns **count** if **entry** is a final-entry placeholder. Otherwise, by assumed proper operation of **if/else**, **@sys.add** and Assertion FAT12-p, **FilesInStep()** returns **FilesInStep(DirectoryID, n+1, count+1)** if **entry** is a used entry, and **FilesInStep(DirectoryID, n+1, count)** if not. **FilesIn()** returns **FilesInStep(DirectoryID, 0, 0)** which by trivial recursion returns the number of used entries, and hence the number of files, in the directory indicated by **DirectoryID**.  $\square$

### Assertion FAT12-4b

`nthFileIn()` correctly returns the File ID of the Nth file in the root directory if passed the "magic number" for the root directory in `DirectoryID`, and otherwise correctly returns the File ID of the Nth file in the directory with the `DirectoryID` passed to it.

```
function FileStep is integer(DirectoryID is integer, n is
    integer, count is integer) {
    if(@this.DirectoryEntryUsed(DirectoryID, n)) {
        if (@sys.grt(count, 0)) {
            @this.FileStep(DirectoryID, @sys.add(n, 1),
                @sys.sub(count, 1))
        } else {
            @this.DirectoryEntryLoc(DirectoryID, n)
        }
    } else {
        @this.FileStep(DirectoryID, @sys.add(n, 1), count)
    }
}

function nthFileIn is integer(DirectoryID is integer,
    N is integer) {
    if(@sys.gre(n, @this.FilesIn(DirectoryID))) { -1 }
    else { @this.FileStep(DirectoryID, 0, N) }
}
```

First consider **FileStep()**. By assumed proper operation of **if/else** and **@sys.grt**, Assertion FAT12-p and Assertion FAT12-m, **FileStep()** returns **FileStep(DirectoryID, n+1, count)** if the **nth** entry in **DirectoryID**'s indicated directory is unused, **FileStep(DirectoryID,**

**n+1, count-1**) if it is used but **count** > 0, and the location of the **n**th entry in **DirectoryID**'s indicated directory if it is used and **count** is 0.

Hence by simple recursion, **FileStep()** steps through the entries in **DirectoryID**'s indicated directory, decrementing **count** each time it encounters a used entry and returning the next used entry it encounters once **count** is 0.

**nthFileIn** returns **FileStep(DirectoryID, 0, N)**, which by simple recursion returns the **N**th used entry in **DirectoryID**'s indicated directory, hence the **N**th file.  $\square$

### Assertion FAT12-5b

**FileIsDirectory()** correctly returns a nonzero value iff the File ID passed to it in **FileID** is the "magic number" for the root directory or otherwise is the File ID of a directory.

```
function FileIsDirectory is integer(FileID is integer)
  with entry is fat12dirent at FileID
  { if(@sys.equ(FileID, @this.RootDirectoryID())) { 1 }
    else { @entry.FileIsDirectory() } }
```

### Demonstration

By assumed proper operation of **if/else** and **@sys.equ**, and by Assertion FAT12-2b, **FileIsDirectory()** returns 1 if it is passed the "magic number" for the root directory and **@entry.FileIsDirectory()** otherwise.

By the definition of a FAT12 File ID, unless it is the root directory's "magic number", **FileID** is the offset of the directory entry of its associated file from the beginning of the partition. Hence by assumed proper operation of **with**, **entry** is the directory entry associated with **FileID**.

By assumed proper operation of function indirection it now suffices to show that **fat12dirent's FileIsDirectory()** function returns true iff the entry is a directory.

```
var FileIsDirectory is public integer littleend width 1
                        at 11 minor 4.
```

By assumed proper operation of a public integer, **fat12dirent's FileIsDirectory()** function returns the value of the **FileIsDirectory** variable. Hence by Assertion FAT12-b, it returns true iff the entry is a directory.  $\square$

### Assertion FAT12-6

**FileIsRegular()** correctly returns a nonzero value iff the File ID passed to it in **FileID** is the File ID of a regular sequential file.

```
function FileIsRegular is integer(FileID is integer)
    with entry is fat12dirent at FileID
    { @sys.and(@sys.not(@this.FileIsDirectory(FileID)),
              @sys.not(@entry.FileIsVolumeLabel())) }
```

### Demonstration

The definition of FAT12 has no explicit specification for a regular file. We must note by process of elimination that a regular file is a file that is neither a directory nor a volume label; hence, it suffices to show that **FileIsRegular()** returns nonzero iff **FileID** is not the File ID of a directory or a volume label.

By Assertion FAT12-5b and assumed proper operation of **@sys.not**, **@sys.not(@this.FileIsDirectory(FileID))** returns nonzero iff **FileID** is not the File ID of a directory.

Hence by assumed proper operation of **@sys.and**, **FileIsRegular()** returns nonzero iff **FileID** is not the File ID of a directory and **@sys.not(@entry.FileIsVolumeLabel())** returns nonzero. By assumed proper operation of **@sys.not** it now suffices to show that **@entry.FileIsVolumeLabel()** returns nonzero iff **FileID** is not the File ID of a volume label.

By the definition of a FAT12 File ID, unless it is the root directory's "magic number", **FileID** is the offset of the directory entry of its associated file from the beginning of the partition. From above, and by short-circuit evaluation, **FileID** cannot refer to a directory at all. Hence by assumed proper operation of **with**, **entry** is the directory entry associated with **FileID**. By assumed proper operation of function indirection it now suffices to show that **fat12dirent**'s **FileIsVolumeLabel()** function returns true iff the entry is a volume label.

```
var FileIsVolumeLabel is public integer littleend width 1
                        at 11 minor 3.
```

By assumed proper operation of a public integer, **fat12dirent**'s **FileIsVolumeLabel()** function returns the value of the **FileIsVolumeLabel** variable. Hence by Assertion FAT12-b, it returns true iff the entry is a volume label.  $\square$

### Assertion FAT12-7

**FileSize()** correctly returns the size of the regular file whose File ID is passed to it in **FileID**.



```
function FileSize is integer(FileID is integer)
    with entry is fat12dirent at FileID
{ @entry.FileSize() }
```

### **Demonstration**

By the definition of a FAT12 File ID, unless it is the root directory's "magic number", **FileID** is the offset of the directory entry of its associated file from the beginning of the partition. On a valid call of FileSize(), **FileID** must refer to a regular file, hence cannot refer to a directory at all. Hence by assumed proper operation of **with**, **entry** is the directory entry associated with **FileID**. By assumed proper operation of function indirection it now suffices to show that **fat12dirent's FileSize()** function returns the size of the file as stored in the directory entry.

```
var FileSize is public integer littleend width 32 at 28.
```

By assumed proper operation of a public integer, **fat12dirent's FileSize()** function returns the value of the **FileSize** variable. Hence by Assertion FAT12-b, it returns the file's size. □

### **Assertion FAT12-8**

FileName() correctly returns the name of the regular file or non-root directory whose File ID is passed to it in FileID.

```
function FileName is char(FileID is integer)

    with entry is fat12dirent at FileID

    { @entry.FileName() }
```

### **Demonstration**

By the definition of a FAT12 File ID, unless it is the root directory's "magic number", **FileID** is the offset of the directory entry of its associated file from the beginning of the partition. On a valid call of **FileName()**, **FileID** cannot refer to the root directory. Hence by assumed proper operation of **with**, **entry** is the directory entry associated with **FileID**. By assumed proper operation of function indirection it now suffices to show that **fat12dirent**'s **FileName()** function returns the file name.

```
function FileName is char() {

    @sys.clipcat(@this.FileNamePart(), @this.FileExtPart())

}
```

By assumed proper operation of **@sys.clipcat**, **FileName** returns **FileNamePart()** concatenated with **FileExtPart()**.

Multiple valid interpretations of FAT's unique 8.3 filename format to general strings exist. We choose to translate to the non-optional filename, concatenated with a period and the extension iff there is an extension; dropping all space padding in all cases.

It is hence sufficient to show that **FileNamePart()** returns the filename without padding, and **FileExtPart()** returns a period concatenated with the extension without padding should the extension exist, and the null string otherwise.

```
function FileNamePart is char() { $FileName<0, @this.FNLength()> }
```

By assumed proper operation of the substring operator, **FileNamePart()** returns the first **FNLength()** characters of the filename. It now suffices to show for **FileNamePart()**'s purposes that **FNLength()** returns the length of the the file name.

```
function FNLengthStep is integer(k is integer) {
    if(@sys.equ(k, 8)) { 8 }
    else {
        if(@sys.clipequ($FileName[k], " ")) { k }
        else { @this.FNLengthStep(@sys.add(k, 1)) }
    }
}

function FNLength is integer() { @this.FNLengthStep(0) }
```

By assumed proper operation of **if** and **@sys.equ**, **FNLengthStep(k)** = 8 if  $k = 8$ .

Otherwise, by assumed proper operation of **@sys.equ** and array functionality, **FNLengthStep(k)** =  $k$  if **FileName[k]** is a space. Otherwise, by assumed proper operation of **@sys.add**, **FNLengthStep(k)** = **FNLengthStep(k+1)**. Hence we have *Assertion FAT12-8-1*:

**FNLengthStep(k)** =  $k$  if  $k = 8$  or **FileName[k]** is a space, **FNLengthStep(k+1)** otherwise.

Now assume without loss of generality that **FileName** has length  $n$ . By definition of FAT12, either  $n = 8$  or **FileName[n]** is a space, and for all  $k < n$ , **FileName[n]** must be a non-space. Hence by Assertion FAT12-8-1, **FNLengthStep(k)** =  $k$  if  $n = k$ , **FNLengthStep(k+1)** otherwise. It follows by trivial recursion that **FNLengthStep(0)** =  $n$  and by assumed proper operation of function calls, **FNLength()** =  $n$  = the length of the file name.

It remains to prove that **FileExtPart()** returns a period concatenated with the extension without padding should the extension exist, and the null string otherwise.

```

function FELengthStep is integer(k is integer) {
    if(@sys.equ(k, 3)) { 3 }
    else {
        if(@sys.clipequ($FileExt[k], " ")) { k }
        else { @this.FELengthStep(@sys.add(k, 1)) }
    }
}

function FELength is integer() { @this.FELengthStep(0) }

```

We first note that **FELength()** returns the length of the extension; the demonstration is essentially identical to that for **FNLength()**. Now consider **FileExtPart()**.

```

function HasExt is integer()
{ @sys.not(@sys.equ(@this.FELength(), 0)) }

function FileExtPart is char() {
    if(@this.HasExt()) {
        @sys.clipcat(".", $FileExt<0, @this.FELength(>))
    } else { "" }
}

```

By the definition of FAT12, if the length of the extension is 0, there is no extension present. By assumed proper operation of **@sys.not** and **@sys.equ** and demonstrated proper operation of **FELength()**, **HasExt()** returns nonzero iff **FELength()** = 0 iff there is no extension present.

Hence by assumed proper operation of **if/else**, **FileExtPart()** returns the null string if there is no extension present. Otherwise, by assumed proper operation of **@sys.clipcat** and the substring operator, it returns a period concatenated with the first **FELength()** characters of the extension – that is, the extension, as desired.  $\square$

### **Assertion FAT12-9-1**

FileHasEvenBlocks() returns a nonzero value iff the regular file associated with the passed FileID has a file size integrally divisible by the block size.

```
function FileHasEvenBlocks is integer(FileID is integer) {  
    if(@sys.grt(@sys.mod(@this.FileSize(FileID),  
                        @this.BlockSize()), 0)) { 0 }  
    else { 1 }  
}
```

### **Demonstration**

By Assertion FAT12-7, **FileSize(FileID)** returns the size of the regular file associated with File ID. By Assertion FAT12-1, **BlockSize()** returns the block size. By assumption of proper implementation of **@sys.mod**, **@sys.mod(@this.FileSize(FileID), @this.BlockSize())** returns greater than zero if and only if the file size is not integrally divisible by the block size. Hence by assumption of proper implementation of **if** and **@sys.grt**, **FileHasEvenBlocks()** returns nonzero iff the file size is integrally divisible by the block size. □

### **Assertion FAT12-9-2**

DataForBlock() returns character data from the data block associated with the passed cluster number, with the given size.

```

function DataForBlock is char(k is integer, size is integer) {
    @sys.charAt(@this.data(@sys.mul(@sys.sub(k, 2),
        @this.BlockSize()))), size)
}

```

### **Demonstration**

By the definition of FAT12, the cluster number  $k$  actually refers to the  $k-2$ th cluster, numbered from zero, in the data area. Hence by arithmetic, cluster number  $k$  begins at offset  $(k-2)$  times the block size in the data area.

By Assertion FAT12-e, **data()** returns the offset within the partition for an offset relative to the data area. Hence by the above, assumption of the proper operation of **@sys.mul** and **@sys.sub**, and Assertion FAT12-1, **@this.data(@sys.mul(@sys.sub(k, 2), @this.BlockSize()))** returns the offset within the partition of the data block with cluster number  $k$ .

Hence by assumption of the proper operation of **@sys.charAt**, **DataForBlock()** returns the data of the passed **size** beginning at the offset of the data block with cluster number  $k$ .  $\square$

### **Assertion FAT12-9-3**

**BlocksInFile()** returns the number of blocks in the cluster chain of the regular file associated with the passed FileID.

```

function BlocksInFile is integer(FileID is integer) {
    @sys.add(@sys.div(@this.FileSize(FileID),
        @this.BlockSize()),
        if(@this.FileHasEvenBlocks(FileID)) { 0 } else { 1 }
    )
}

```

### **Demonstration**

By arithmetic, a given file has  $n$  blocks where  $n$  equals the ceiling of the file size  $f$  divided by the block size  $b$ .

By assumption of proper operation of **@sys.div**, Assertion FAT12-7 and Assertion FAT12-1, **@sys.div(@this.FileSize(FileID), @this.BlockSize()) =  $f / b$** . We can obtain the ceiling by adding 1 iff  $f \bmod b > 0$ ; hence by assumption of proper operation of **if** and **@sys.add**, and Assertion FAT12-9-1, **BlocksInFile** returns  $\lceil f / b \rceil$  as desired.  $\square$

### **Assertion FAT12-9**

FileBlock() correctly returns the BlockNumth block of data of the regular file whose File ID is passed to it in FileID.

```

function FileBlock is char(FileID is integer, n is integer) {
    if(@sys.and(@sys.equ(@sys.add(n, 1),
                        @this.BlocksInFile(FileID)),
        @sys.not(@this.FileHasEvenBlocks(FileID)))
    { @this.DataForBlock(@this.nthBlock(FileID, n),
        @sys.mod(@this.FileSize(FileID)
        @this.BlockSize())) }
    else { @this.DataForBlock(@this.nthBlock(FileID, n),
        @this.BlockSize()) }
}

```

### **Demonstration:**

By arithmetic, a given file has  $n$  blocks where  $n$  equals the ceiling of the file size  $f$  divided by the block size  $b$ . The file's last block has the partition's full block size iff  $f \bmod b = 0$ ; otherwise, it has size  $f \bmod b$ . Hence, we need to show two cases:

**Case 1:  $n$  is the last block of the file associated with  $\text{FileID}$ , and the file's size is not integrally divisible by the block size.**

Given that  $n$  is the last block of  $\text{FileID}$ 's file, by Assertion FAT12-9-3,  $n = (\text{BlocksInFile}(\text{FileID}) - 1)$ .

Given that  $\text{FileID}$ 's file's size is not integrally divisible by the block size, by Assertion FAT12-9-1,  $\text{FileHasEvenBlocks}(\text{FileID})$  returns zero.

Hence by assumed proper operation of **if**, **@sys.and**, **@sys.equ**, **@sys.add** and **@sys.not**, **FileBlock()** returns **@this.DataForBlock(@this.nthBlock(FileID, n), @sys.mod(@this.FileSize(FileID), @this.BlockSize()))**.

By Assertion FAT12-j, **nthBlock(FileID, n)** is the block number of the  $n$ th block of the file associated with **FileID**.



By Assertion FAT12-7, **FileSize(FileID)** is the file size.

By Assertion FAT12-1, **BlockSize()** is the block size.

Hence by Assertion FAT12-9-2 and assumed proper operation of **@sys.mod, FileBlock()** returns data of size  $f \bmod b$  where  $f$  is the file size and  $b$  is the fundamental block size from **FileID**'s file's last cluster, as desired.

### **Case 2: Otherwise.**

By assumption of proper operation of **else, FileBlock()** returns **@this.DataForBlock(@this.nthBlock(FileID, n), @this.BlockSize())**.

By Assertion FAT12-j, **nthBlock(FileID, n)** is the block number of the  $n$ th block of the file associated with **FileID**.

By Assertion FAT12-1, **BlockSize()** is the block size.

Hence by Assertion FAT12-9-2, **FileBlock()** returns data of the fundamental block size from **FileID**'s file's  $n$ th cluster, as desired.  $\square$

### **Conclusion**

*Result HSL-FAT12:* By Assertions FAT12-1, FAT12-2b, FAT12-3b, FAT12-4b, FAT12-5b, FAT12-6, FAT12-7, FAT12-8, FAT12-9 and Conclusion 2, the **fat12** HSL specification properly reads the FAT12 file system as defined above.  $\square$

### **Definition of ext2fs**

Primary sources for this section include (Card) and (Poirer).

## **Overview**

The Second Extended File System, or ext2fs, is an ordinary hierarchical file system with the following characteristics.

All integers are little-endian.

## **Partition Metadata**

The ext2 filesystem has two relevant pieces of partition-level metadata. One is the Superblock; the second is the set of group descriptors.

The Superblock occurs several places in the partition, but always at offset 1024, and contains the following data:

Table 10: ext2fs superblock

Offset	Length	Object
0	4	Inodes in the file system
4	4	Blocks in the file system
8	4	Blocks reserved for superuser
12	4	Free blocks
16	4	Free inodes
20	4	First data block
24	4	Block size shift
28	4	Fragment size shift
32	4	Blocks per group
36	4	Fragments per group
40	4	Inodes per group
44	4	Last mount time
48	4	Last write time
52	2	Times mounted since check
54	2	Maximum times to mount before check
56	2	Magic number (0xEF53)
58	2	File system state
60	2	Error behavior
62	2	Minor revision level
64	4	Time of last filesystem check
68	4	Maximum interval between filesystem checks
72	4	Creator OS
76	4	Revision level
80	940	Reserved

The first data block is the ID of the block that contains the superblock structure. No other partition data occurs before the superblock. ext2 does not use block numbers relative to a data area - all block numbers are relative to the partition.

The block size shift indicates the file system's block size; the size is  $1024 \ll \text{block size}$  shift.

The fragment size shift likewise indicates the file system's fragment size; a negative value can cause a right shift. In practice, except in highly exotic implementations, ext2fs's block size and fragment size are identical. For simplicity's sake, we will assume this is the case.

The group descriptors occur one after the other immediately after the superblock, and there are as many of them as necessary given the block count and blocks per group.

Table 11: ext2fs block group

Offset	Length	Object
0	4	Block bitmap location (block number)
4	4	Inode bitmap location (block number)
8	4	Inode table location (block number)
12	2	Free blocks
14	2	Free inodes
16	2	Used directories
18	2	Pad
20	12	Reserved

`bg_block_bitmap`, `bg_inode_bitmap`, and `bg_inode_table` are the block numbers of the block bitmap, inode bitmap and inode table for the group. Note that ext2 does not use block numbers relative to a data area - all block numbers are relative to the partition.

The inode table contains the inodes for the group described by a given descriptor. If  $k$  is the number of inodes per group, then the first group descriptor has a pointer to a table containing inodes 1 to  $k$ , the second has a pointer to a table containing inodes  $k+1$  to  $2k$ , and so on.

### **File Metadata**

Inode tables are arrays of inodes, which take the following form.

Table 12: ext2fs inode

Offset	Length	Object
0	2	File mode
2	2	User ID
4	4	File size
8	4	Access time
12	4	Creation time
16	4	Modification time
20	4	Deletion time
24	2	Group ID
26	2	Link count
28	4	Number of blocks
32	4	Behavior flags
36	4	Reserved
40	4x15	Block numbers
100	4	File version
104	4	File ACL
108	4	Directory ACL
112	4	Location of last fragment
116	12	Reserved

The first twelve block numbers directly point at file data.

The thirteenth block number is for single indirect addressing; for a file larger than 12 blocks, it points to a block of  $(\text{block size} / 4)$  additional block numbers.

The fourteenth block number is for double indirect addressing; for a file larger than  $(12 + \text{block size} / 4)$  blocks, it points to  $(\text{block size} / 4)$  blocks \*each\* containing  $(\text{block size} / 4)$  additional block numbers.

The fifteenth block number is for triple indirect addressing; for a file larger than  $(12 + (\text{block size} / 4) * (\text{block size} / 4 + 1))$  blocks, it points to  $(\text{block size} / 4)$  blocks each containing pointers to  $(\text{block size} / 4)$  blocks each containing  $(\text{block size} / 4)$  additional block numbers.

Indirect addressing is additive - the inode's twelve internal blocks are still used when single indirect addressing is in play, which is still used when double indirect addressing is in play, which is still used when triple indirect addressing is in play.

Each inode has an inode number; each inode table stores `s_inodes_per_group` inodes. inode indexes begin at 1, not at 0; hence, to find an inode's group and offset, subtract 1 before dividing by `s_inodes_per_group` to get the group and performing modulo arithmetic to get the offset.

The root directory has the reserved inode number 2. Inode number 0 is never used.

The twelve least-significant bits of the mode block concern file permissions. The upper four are the file format, one of the following:

Table 13: ext2fs file formats

Value	Format
1	FIFO
2	Character Device
4	Directory
6	Block Device
8	Regular File
10	Symbolic Link
12	Socket

## **Directories**

Directories are stored as files that contain directory entries as their data. Directory entries contain a file's name and its inode. In newer versions of ext2fs they also contain an easier way to get the file mode, but as it isn't well documented which versions this is true for, we don't consider it reliable.

They are structured as follows:



Table 14: ext2fs directory entry

Offset	Length	Object
0	4	Inode Number
4	2	Record Length
6	1	Name Length
8	Name Length	Name

Directory entries are, as the above implies, not of constant length. The record length gives the total length of a given directory entry (and is *\*not\** guaranteed to be seven plus the name length). A directory entry with inode number 0 is unused and should be ignored. The last directory entry in a block is extended to fill the block, so determining when to move on to the next block is straightforward.

### Demonstration of Correctness for EXT2FS

All variables and functions are with respect to the HSL **ext2** spacetype, unless noted otherwise. Correct operation of the Hadley system and its built-in functions is assumed at all times. All division is assumed to be integer. We note that through the assumed proper operation of **use** clauses, the **ext2** spacetype is able to declare **ext2dirent**, **ext2inode**, **ext2superblock** and **ext2groupdescriptor** subspaces, and that **ext2inode** is able to declare **ext2filemode** subspaces.

### **Definition (EXT2FS File ID)**

The File ID for the HSL **ext2** spacetype is either the offset of the directory entry of the associated file from the start of the partition or the number one, indicating the root directory.

### **Assertion EXT2FS-a**

**@superblock.BlocksCount()**, **@superblock.LogBlockSize()**, **@superblock.BlocksPerGroup()**, and **@superblock.InodesPerGroup()** return the number of blocks, the block size shift, the number of blocks per group, and the number of inodes per group for the partition, respectively.

```
subspace superblock is ext2superblock at 1024.
```

### **Demonstration**

By assumed proper operation of subspace, **superblock** is a space of type **ext2superblock** at offset 1024 in the partition. By the definition of ext2fs the superblock occurs at offset 1024 in the partition. Hence it suffices to show that **ext2superblock** indicates the desired values properly with respect to the ext2fs superblock structure.

```
var BlocksCount      is public integer littleend width 32 at 4.  
var LogBlockSize     is public integer littleend width 32 at 24.  
var BlocksPerGroup   is public integer littleend width 32 at 32.  
var InodesPerGroup   is public integer littleend width 32 at 40.
```

By the definition of ext2fs the number of blocks, block size shift, number of blocks per group and number of inodes per group occur at offsets 4, 24, 32, and 40 in the superblock respectively and are each 4 bytes wide. Simple inspection then demonstrates the equivalence of

the variable specifications with their corresponding locations in the superblock structure, and by assumed proper operation of public functions the corresponding functions return the variables' values. □

#### **Assertion EXT2FS-b**

Given an **ext2groupdescriptor** structure describing an ext2fs group descriptor, the function **InodeTable()** returns the first block number of the inode table for the group.

<pre>var InodeTable is public integer littleend width 32 at 8.</pre>
--

#### **Demonstration**

By the definition of ext2fs the inode table block number occurs at offset 8 in the group descriptor and is 4 bytes wide. Simple inspection demonstrates the equivalence of the variable specification. By assumed proper operation of a public function the corresponding function returns the variable's value. □

#### **Assertion EXT2FS-c**

Given an **ext2inode** structure describing an ext2fs inode, the functions **Size()**, **Blocks()**, **IsRegularFile()**, and **IsDirectory()** return the size of the inode's associated file, the block count of the inode's associated file or directory, nonzero iff the inode refers to a sequential file and nonzero iff the inode refers to a directory, respectively.

```
var Size      is public integer littleend width 32 at 4.  
var Blocks    is public integer littleend width 32 at 28.
```

### Demonstration

By the definition of ext2fs the file size and block count occur at offsets 4 and 28 in the inode, and both are 4 bytes wide. Simple inspection demonstrates the equivalence of the variable specifications. By assumed proper operation of public functions the corresponding functions return the variables' values.

```
subspace Mode      is ext2filemode at 0.  
  
function IsRegularFile is integer() { @Mode.IsRegularFile() }  
  
function IsDirectory   is integer() { @Mode.IsDirectory() }
```

By the definition of ext2fs the file mode occurs at offset 0 in the inode. It now suffices to show that with given an **ext2filemode** structure describing an ext2fs file mode block, the functions **IsRegularFile()** and **IsDirectory()** return nonzero iff the inode refers to a sequential file and iff the inode refers to a directory, respectively.

```
var FileMode is public integer littleend width 4 at 1 minor 4.  
  
function IsRegularFile is integer() { @sys.equ($FileMode, 8) }  
  
function IsDirectory   is integer() { @sys.equ($FileMode, 4) }
```

By the definition of ext2fs the file format occurs in the four most significant bits of the mode block. Simple inspection demonstrates the equivalence of the **FileMode** specification. By the definition of ext2fs an inode refers to a regular file iff the file format value is 8, and a directory iff the file format value is 4. Hence by assumed proper operation of **@sys.equ** **IsRegularFile()** returns nonzero iff the file mode value is 8 iff the inode refers to a sequential file, and **IsDirectory()** returns nonzero iff the file mode value is 4 iff the inode refers to a directory. □

### **Assertion EXT2FS-d**

Given an **ext2inode** structure describing an ext2fs inode, the function **Block()** passed a number  $n$  with a value 0 through 11 returns the  $n$ th block number for the file, passed the number 12 it returns the single-indirect block number for the file, passed the number 13 it returns the double-indirect block number for the file and passed the number 14 it returns the triple-indirect block number for the file.

```
var Block is public integer[15] littleend width 32 at 40.
```

### **Demonstration**

By the definition of ext2fs the block numbers occur at offset 40 in the inode, in an array of 15, each 4 bytes wide. The first twelve entries are the first twelve block numbers for the file, and the next three are the single-, double-, and triple-indirect block numbers respectively. Simple inspection shows that the variables **Block[0..11]** correspond to the first twelve block numbers and **Block[12..14]** correspond to the respective indirect block numbers. By assumed proper operation of a public function **Block()** returns these values as desired.  $\square$

### **Assertion EXT2FS-e**

Given an **ext2dirent** structure describing an ext2fs directory entry, the functions **Inode()**, **RecordLength()**, **NameLength()**, **Name()** and **IsUsed()** return the associated inode number, the length of the entry, the length of the name, the name, and nonzero iff the directory entry is used, respectively.

```
var Inode          is public integer littleend width 32 at 0.  
var RecordLength  is public integer littleend width 16 at 4.  
var NameLength    is public integer littleend width 8  at 6.
```

By the definition of ext2fs the inode number, record length and name length occur at offsets 0, 4 and 6 in the directory entry, and have widths of 4, 2 and 1 byte(s), respectively. Simple inspection demonstrates the equivalence of the variable specifications and by assumed proper operation of public functions the corresponding functions return the variables' values.

```
var Name          is      char[$NameLength]      at 8.  
  
function Name is char() {  
    $Name<0, $NameLength>  
}
```

By the definition of ext2fs the file name occurs at offset 8 in the directory entry, and from the above it has string length corresponding to the value of **NameLength**. Simple inspection demonstrates the equivalence of the variable specification, hence by assumed proper operation of the multiple array operator **Name()** returns the file name.

```
function IsUsed is integer() {  
    @sys.not(@sys.equ($Inode, 0))  
}
```

By the definition of ext2fs a directory entry is used iff its inode number is other than zero. From above, **Inode** is the inode number. Hence by assumed proper operation of **@sys.not** and **@sys.equ**, **IsUsed()** returns true iff the inode number is other than zero iff the directory entry is used.  $\square$

### **Assertion EXT2FS-1**

BlockSize() correctly returns the file system's fundamental block size. (This is not the same thing as the sector size of the underlying media.)

```
function BlockSize is integer() {  
    @sys.mul(1024, @sys.2to(@superblock.LogBlockSize()))  
}
```

### **Demonstration**

By Assertion EXT2FS-a, **@superblock.LogBlockSize** is equal to the block size shift. From the definition of ext2fs, the block size is equal to  $1024 \times 2^{(\text{block size shift})}$ . By Assertion EXT2FS-a and assumed proper operation of **@sys.2to**, **BlockSize()** returns  $1024 \times 2^{(\text{block size shift})}$ .  $\square$

### **Assertion EXT2FS-1-1**

BlockSize4() returns the block size divided by 4.

```
function BlockSize4 is integer() {  
    @sys.div(@this.BlockSize(), 4)  
}
```

### **Demonstration**

From Assertion EXT2FS-1 and assumed proper operation of **@sys.div**, the desired result immediately follows.  $\square$

### **Assertion EXT2FS-f**

IntExtract(), passed a block number  $n$  for a block of 4-byte integers and an integer number  $k$ , extracts the  $k$ th 4-byte integer from that block.

```
function IntExtract is integer(n is integer, k is integer) {  
    @sys.intAt(@sys.add(@sys.mul(n, @this.BlockSize()),  
                        @sys.mul(k, 4)), 0, 32, 0)  
}
```

### **Demonstration**

By assumed proper operation of @sys.intAt(), IntExtract() returns a little-endian 4-byte integer from @sys.add(@sys.mul(n, @this.BlockSize()), @sys.mul(k, 4)). By assumed proper operation of @sys.add and @sys.mul this is  $(n \times \text{BlockSize}) + 4k$ . By Assertion EXT2FS-1 this is the offset  $4k$  bytes into block  $n$ , hence the location of the desired integer.  $\square$

### **Assertion EXT2FS-g**

SIndMax() and DIndMax() return the maximum number of file blocks for single and double indirect addressing, hence the lowest file block numbers requiring double and triple indirect addressing, respectively.



```

function SIndMax is integer() {
    @sys.add(@this.BlockSize4(), 12)
}

function DIndMax is integer() {
    @sys.add(@sys.mul(@this.BlockSize4(), @this.BlockSize4()),
            @this.SIndMax())
}

```

### **Demonstration**

By the definition of ext2fs, double indirect addressing is needed for files larger than  $(12 + \text{block size} / 4)$  blocks. By proper operation of **@sys.add** and Assertion EXT2FS-1-1, **SIndMax()** returns this value.

By the definition of ext2fs, triple indirect addressing is needed for files larger than

$$(12 + (\text{block size} / 4) * (\text{block size} / 4 + 1)) \text{ blocks}$$

$$= (12 + (\text{block size} / 4) + (\text{block size} / 4)^2) \text{ blocks} \quad \textit{arithmetic}$$

$$= (\mathbf{SIndMax()}) + (\text{block size} / 4)^2 \text{ blocks} \quad \textit{above}$$

and by assumed proper operation of **@sys.add** and **@sys.mul**, **DIndMax()** returns this value.  $\square$

### **Assertion EXT2FS-h**

DoubleIndirect(), passed a source block number  $n$  and an offset block number  $k$ , where  $k$  is less than  $s^2$  where  $s$  is the partition block size divided by 4, returns the  $k$ th block pointed to by  $n$ 's double-indirect addressing.

```

function DoubleIndirect is integer(n is integer, k is integer) {
    @this.IntExtract(
        @this.IntExtract(n, @sys.div(k, @this.BlockSize4())),
        @sys.mod(k, @this.BlockSize4())
    )
}

```

### **Demonstration**

By the definition of ext2fs double indirect addressing,  $n$  is a block number pointing to a block containing an array of  $s$  integers, where  $s$  is the partition's block size divided by 4. These integers each in turn are block numbers pointing to blocks containing arrays of  $s$  integers as well. Hence  $n$ 's double-indirect addressing points to  $s^2$  blocks, of which we want the  $k$ th. By arithmetic  $k = ls + m$  where  $l = k/s$  and  $m = k \bmod s$ , since  $k < s^2$   $l < s$  and  $m < s$ , and our result must be the  $m$ th integer in the  $l$ th block pointed to by block  $n$ .

By assumed proper operation of **@sys.div** and **@sys.mod**, and Assertion EXT2FS-1-1, **DoubleIndirect()** returns **IntExtract(IntExtract( $n$ ,  $l$ ),  $m$ )**. By Assertion EXT2FS-f, this is the  $m$ th integer in the block with the number that is the  $l$ th integer in  $n$  – that is, the  $m$ th integer in the  $l$ th block pointed to by block  $n$ .  $\square$

### **Assertion EXT2FS-i**

TripleIndirect(), passed a source block number  $n$  and an offset block number  $k$ , where  $k$  is less than  $s^3$  where  $s$  is the partition block size divided by 4, returns the  $k$ th block pointed to by  $n$ 's triple-indirect addressing.

```

function TripleIndirect is integer(n is integer, k is integer) {
    @this.IntExtract(
        @this.IntExtract(
            @this.IntExtract(
                n,
                @sys.div(@sys.div(k, @this.BlockSize4()),
                    @this.BlockSize4())
            ),
            @sys.mod(@sys.div(k, @this.BlockSize4()),
                @this.BlockSize4())
        ),
        @sys.mod(k, @this.BlockSize4())
    )
}

```

### **Demonstration**

By the definition of ext2fs triple indirect addressing,  $n$  is a block number pointing to a block containing an array of  $s$  integers, where  $s$  is the partition's block size divided by 4. These integers each in turn are block numbers pointing to blocks containing arrays of  $s$  integers as well, and these integers finally are block numbers pointing to blocks containing arrays of  $s$  integers themselves. Hence  $n$ 's triple-indirect addressing points to  $s^3$  blocks, of which we want the  $k$ th.

By arithmetic decompose  $k$  into base  $s$ .  $k = ls + q$  where  $l = k/s$  and  $q = k \bmod s$ , and further,  $l = os + p$  where  $o = l/s$  and  $p = l \bmod s$ . Since  $k < s^3$ ,  $o$ ,  $p$  and  $q$  are all less than  $s$ . Our result must be the  $q$ th integer in the block pointed to by the  $p$ th integer in the  $o$ th block pointed to by block  $n$ .

By assumed proper operation of **@sys.div** and **@sys.mod**, and Assertion EXT2FS-1-1, **TripleIndirect()** returns **IntExtract(IntExtract(IntExtract(*n*, *o*), *p*), *q*)**. By repeated application of Assertion EXT2FS-f, this is the *q*th integer in the block pointed to by the *p*th integer in the *o*th block pointed to by *n*.  $\square$

### **Assertion EXT2FS-j**

**TranslateBlock()**, given the location of an inode and a number *n*, returns the partition block number for that inode's *n*th block.

```

function TranslateBlock is integer(inodeloc is integer,
                                n is integer)

    with inode is ext2inode at inodeloc

{
    if(@sys.lst(n, 12)) { @inode.Block(n) } else {
        if(@sys.lst(n, @this.SIndMax())) {
            @this.IntExtract(@inode.Block(12),
                            @sys.sub(n, 12))
        } else {
            if(@sys.lst(n, @this.DIndMax())) {
                @this.DoubleIndirect(@inode.Block(13),
                                    @sys.sub(n, @this.SIndMax()))
            } else {
                @this.TripleIndirect(@inode.Block(14),
                                    @sys.sub(n, @this.DIndMax()))
            }
        }
    }
}

```

### **Demonstration**

By the assumed proper operation of **with**, **inode** is an **ext2inode** structure describing the inode at the passed location. By the definition of ext2fs block addressing, there are four possible cases.

#### **Case 1:** Direct addressing.

By the definition of ext2fs block addressing, if the block number is lower than 12, indirect addressing is used based on one of the first twelve entries in the inode's block list. By

assumed proper operation of **if** and **@sys.lst**, if the block number is lower than 12, then

**TranslateBlock()** returns **@inode.Block(n)**. By Assertion EXT2FS-d this is the partition block number for the inode's  $n$ th block as desired.

**Case 2:** Single-indirect addressing.

By the definition of ext2fs block addressing, if the block number is at least twelve but low enough to be handled by single indirect addressing, single indirect addressing is used. By assumed proper operation of **if/else**, **@sys.lst** and **@sys.sub**, and Assertion EXT2FS-g, if the block number is at least twelve but low enough to be handled by single indirect addressing, then **TranslateBlock()** returns **IntExtract(@inode.Block(12),  $n-12$ )**. By Assertions EXT2FS-d and EXT2FS-f, this is the  $n-12$ th entry in the single-indirect block for the inode, as desired.

**Case 3:** Double-indirect addressing.

By the definition of ext2fs block addressing, if the block number is too high to be handled by single indirect addressing but low enough to be handled by double indirect addressing, double indirect addressing is used. By assumed proper operation of **if/else**, **@sys.lst** and **@sys.sub**, and Assertion EXT2FS-g, if the block number is too high to be handled by single indirect addressing but low enough to be handled by double indirect addressing, then **TranslateBlock()** returns **DoubleIndirect(@inode.Block(13),  $n-SIndMax()$ )**. By Assertions EXT2FS-d and EXT2FS-f, this is the  $n-s$ th entry in the double-indirect block structure for the inode, where  $s$  is the number of blocks handled by single-indirect addressing, as desired.

**Case 4:** Triple-indirect addressing.

By the definition of ext2fs block addressing, if the block number is too high to be handled by double indirect addressing, triple indirect addressing is used. By assumed proper operation of **if/else**, **@sys.lst** and **@sys.sub**, and Assertion EXT2FS-g, if the block number is too

high to be handled by double indirect addressing, then **TranslateBlock()** returns **TripleIndirect(@inode.Block(14),  $n - \mathbf{DIndMax}()$ )**. By Assertions EXT2FS-d and EXT2FS-f, this is the  $n - d$ th entry in the triple-indirect block structure for the inode, where  $d$  is the number of blocks handled by double-indirect addressing, as desired.  $\square$

### **Assertion EXT2FS-k**

FindInodeNumber(), given an inode number, returns the offset of that inode in the partition.

```

function FindInodeNumber is integer (InodeID is integer)

  with groupdescriptor is ext2groupdescriptor at

    @sys.add(2048,

      @sys.mul(32,

        @sys.div(@sys.sub(InodeID, 1),

          @superblock.InodesPerGroup()))))

  {

    @sys.add(

      @sys.mul(@groupdescriptor.InodeTable(),

        @this.BlockSize()),

      @sys.mul(@sys.mod(@sys.sub(InodeID, 1),

        @superblock.InodesPerGroup()), 128)

    )

  }

```

### **Demonstration**

By the definition of ext2fs,  $s$  inodes are in the table pointed to by each group descriptor, and the very first inode is numbered one rather than zero. Hence, by zero-indexing, we actually want the  $i$ th inode, where  $i = \mathbf{InodeID} - 1$ . By arithmetic,  $i = k + l$  where  $k = i / s$  and  $l = i \bmod s$  and we want the  $l$ th inode in group  $k$ .

By the definition of ext2fs, the group descriptors occur immediately after the superblock, which occurs at offset 1024 and has length 1024. Hence the group descriptors begin at offset 2048. Also by the definition of ext2fs, the group descriptor structure is 32 bytes long. Hence the group descriptor whose inode table we want is at  $32k$  bytes from offset 2048 in the partition.

By assumed proper operation of **@sys.add**, **@sys.mul**, **@sys.div** and **@sys.sub**, and Assertion EXT2FS-a, **groupdescriptor**'s location, assuming proper operation of **with**, places it



at  $2048 + 32((\text{InodeID} - 1) / s) = 2048 + 32k$ . Hence by assumed proper operation of **with**, **groupdescriptor** indicates group  $k$  and it remains to show that we return the  $l$ th node from its table.

By the definition of an ext2fs group descriptor, the inode table entry is a partition block number; hence the offset for the inode table of a group is the inode table entry for that group times the block size. Hence by Assertions EXT2FS-b and EXT2FS-1, and the assumption of proper operation of **@sys.add** and **@sys.mul**, **FindInodeNumber()** returns the offset of **groupdescriptor**'s inode table plus **@sys.mul(@sys.mod(@sys.sub(InodeID, 1), @superblock.InodesPerGroup()), 128)**. By assumption of proper operation of **@sys.mul**, **@sys.mod** and **@sys.sub**, and Assertion EXT2FS-a, this is  $128l$ . By definition of an ext2 inode, inodes are 128 bytes long. Hence **FindInodeNumber()** returns the offset of **groupdescriptor**'s inode table plus  $128l$ , or the  $l$ th inode in group  $k$ , as desired.  $\square$

### Assertion EXT2FS-l

**GetRootDirectoryInodeLoc()** returns the offset in the partition of the root directory's inode.

```
function GetRootDirectoryInodeLoc is integer() {
    @this.FindInodeNumber(2)
}
```

### Demonstration

By the definition of ext2fs, the root directory has inode number 2. Hence by Assertion EXT2FS-k **GetRootDirectoryInodeLoc()** returns the location of the root directory's inode.  $\square$

### **Assertion EXT2FS-2b**

RootDirectoryID() returns a unique "magic number" for the root directory that cannot otherwise refer to a valid directory or file.

```
function RootDirectoryID is integer() {  
    1  
}
```

### **Demonstration**

By the definition of ext2fs, the superblock occurs at offset 1024, and no other data can appear before the superblock. Hence no directory entry can appear at offset 1. Hence, by definition of an EXT2FS File ID, 1 cannot be a valid File ID other than the Root Directory ID. □

### **Assertion EXT2FS-m**

FindInodeForFileID(), given a File ID, returns the offset in the partition of the inode for the associated file.

```

function FindInodeForFileID is integer (FileID is integer)

    with dirent is ext2dirent at FileID

{
    if(@sys.equ(FileID, @this.RootDirectoryID())) {
        @this.GetRootDirectoryInodeLoc()
    } else {
        @this.FindInodeNumber(@dirent.Inode())
    }
}

```

### **Demonstration**

By the definition of an EXT2FS File ID, there are two cases. In the first case, the File ID is the ID of the root directory. In this case, by assumed proper operation of **if** and Assertion EXT2FS-2b, **FindInodeForFileID()** returns **GetRootDirectoryInodeLoc()**, and by Assertion EXT2FS-l, that is as desired. In the second case, the File ID is the location of a directory entry. Hence by assumed proper operation of **with**, **dirent** is the **ext2dirent** structure for **FileID**'s associated file, and by assumed proper operation of **else**, Assertion EXT2FS-k and Assertion EXT2FS-e, we return the location of its inode.  $\square$

### **Assertion EXT2FS-n**

DirectoryEntryLoc() returns the location of the  $n$ th directory entry in the directory with the DirectoryID passed to it, or 0 if  $n \geq$  the number of entries in that directory.

```

function DirectoryEntryLocStep is integer(inodeloc is integer,
                                          count is integer,
                                          block is integer,
                                          loc is integer)

with inode is ext2inode at inodeloc,
     dirent is ext2dirent
     at @sys.add(
         @sys.mul(
             @this.TranslateBlock(inodeloc, block),
             @this.BlockSize()),
         loc)
{
    if(@sys.equ(block, @inode.Blocks())) { 0 }
    else {
        if(@sys.equ(count, 0)) {
            @sys.add(@sys.mul(@this.TranslateBlock(inodeloc, block),
                @this.BlockSize()),
                loc)
        } else {
            if(@sys.equ(@sys.add(loc, @dirent.RecordLength()),
                @this.BlockSize())) {
                @this.DirectoryEntryLocStep(inodeloc, @sys.sub(count, 1),
                    @sys.add(block, 1), 0)
            } else {
                @this.DirectoryEntryLocStep(inodeloc, @sys.sub(count, 1),
                    block,
                    @sys.add(loc,
                        @dirent.RecordLength()))
            }
        }
    }
}

function DirectoryEntryLoc is integer(DirectoryID is integer,
                                     n is integer)
with dirent is ext2dirent at DirectoryID
{
    @this.DirectoryEntryLocStep(
        @this.FindInodeForFileID(DirectoryID), n, 0, 0)
}

```

### **Demonstration**

We first show that **DirectoryEntryLocStep()**, passed the offset of the inode of a directory in **inodeloc**, a **count** of desired entries forward, the **block** of the directory in which to begin, and the **location** in that block of a valid directory entry, returns:

0 if the block is higher than the last block in the directory,

The partition offset of the directory entry that **loc** points to if **count** is 0,

**DirectoryEntryLocStep(inodeloc, count – 1, block + 1, 0)** if **loc** points to the last directory entry in the **block**, and

**DirectoryEntryLocStep(inodeloc, count - 1, block, loc')** where **loc'** is the location in the **block** of the next directory entry after **loc**, otherwise.

By assumed proper operation of **@sys.add** and **@sys.mul**, Assertion EXT2FS-j, Assertion EXT2FS-1, arithmetic, and the definition of ext2fs partition blocks, **@sys.add(@sys.mul(@this.TranslateBlock(inodeloc, block), @this.BlockSize()), loc)** is the partition offset of the location with offset **loc** in block **block** of the inode associated with **inodeloc**.

By assumed proper operation of **with**, **inode** is an **ext2inode** structure describing the inode associated with **inodeloc**.

By assumed proper operation of **with** and from above, **dirent** is an **ext2dirent** structure describing the directory entry at **loc** in block **block** of the inode associated with **inodeloc**.

By assumed proper operation of **if/else**, there are four cases.

**Case 1:** By assumed proper operation of **@sys.equ** and Assertion EXT2FS-c, **block** is greater than or equal to the number of blocks in the inode referred to by **inodeloc**. Then **block** is a higher number than the last block in the directory, and **DirectoryEntryLocStep()** returns 0 as desired.

**Case 2:** By assumed proper operation of **@sys.equ**, **count** is 0. Then from above, **DirectoryEntryLocStep** returns the partition offset of the location with offset **loc** in block **block** of the inode associated with **inodeloc**, as desired.

**Case 3:** By assumed proper operation of **@sys.equ** and **@sys.add**, and Assertions EXT2FS-e and EXT2FS-1, **loc** plus the record length of the directory entry it points to equals the

block size. Hence **loc** clearly points to the last directory entry in the block. By assumed proper operation of **@sys.add** and **@sys.sub** we return **DirectoryEntryLocStep(inodeloc, count – 1, block + 1, 0)** as desired.

**Case 4:** By presumed proper operation of **else**, none of the above. By assumed proper operation of **@sys.sub**, **@sys.add** and Assertion EXT2FS-e, we return **DirectoryEntryLocStep(inodeloc, count - 1, block, loc')** as desired.

Hence by simple recursion, **DirectoryEntryLocStep()** steps through the entries in the directory data pointed to by **inodeloc**, decrementing **count** for each entry, and returning the entry it reaches when **count** reaches 0 or returning 0 if it runs out of entries before that.

By Assertion EXT2FS-m, **DirectoryEntryLoc()** returns **DirectoryEntryLocStep(inodeloc, n, 0, 0)** where **inodeloc** is the location of the inode associated with **DirectoryID**. By simple recursion this is either the **n**th directory entry in the directory associated with **DirectoryID** or 0 if that entry does not exist. □

### **Assertion EXT2FS-3b**

**FilesIn()** correctly returns the number of files in the root directory if passed the "magic number" for the root directory in **DirectoryID**, and otherwise correctly returns the number of files in the directory with the **DirectoryID** passed to it.

```

function FilesInStep is integer(DirectoryID is integer,
                                n is integer,
                                count is integer)

    with dirent is ext2dirent
        at @this.DirectoryEntryLoc(DirectoryID, n)
    {
        if(@sys.equ(@this.DirectoryEntryLoc(DirectoryID, n), 0)) {
            count
        } else {
            @this.FilesInStep(DirectoryID, @sys.add(n, 1),
                              if(@dirent.IsUsed()) { @sys.add(count, 1) }
                              else { count })
        }
    }

function FilesIn is integer(DirectoryID is integer) {
    @this.FilesInStep(DirectoryID, 0, 0)
}

```

First consider **FilesInStep()**. By assumed proper operation of **with** and Assertion EXT2FS-n, **dirent** is the **n**th directory entry in the directory indicated by **DirectoryID**. By assumed proper operation of **if** and Assertion EXT2FS-n, **FilesInStep()** returns **count** if this entry does not exist. Otherwise, by assumed proper operation of **if/else**, **@sys.add** and Assertion EXT2FS-e, **FilesInStep()** returns **FilesInStep(DirectoryID, n+1, count+1)** if **dirent** is a used entry, and **FilesInStep(DirectoryID, n+1, count)** if not. **FilesIn()** returns **FilesInStep(DirectoryID, 0, 0)** which by trivial recursion returns the number of used entries, and hence the number of files, in the directory indicated by **DirectoryID**.  $\square$

**Assertion EXT2FS-4b**

nthFileIn() correctly returns the File ID of the Nth file in the root directory if passed the "magic number" for the root directory in DirectoryID, and otherwise correctly returns the File ID of the Nth file in the directory with the DirectoryID passed to it.



```

function nthFileInStep is integer(DirectoryID is integer,
                                n is integer, count is integer)

    with dirent is ext2dirent
        at @this.DirectoryEntryLoc(DirectoryID, n)
    {
        if(@dirent.IsUsed()) {
            if(@sys.equ(count, 0)) {
                @this.DirectoryEntryLoc(DirectoryID, n)
            } else {
                @this.nthFileInStep(DirectoryID,
                                    @sys.add(n, 1), @sys.sub(count, 1))
            }
        } else {
            @this.nthFileInStep(DirectoryID,
                                @sys.add(n, 1), count)
        }
    }

function nthFileIn is integer(DirectoryID is integer, n is integer)
{
    @this.nthFileInStep(DirectoryID, 0, n)
}

```

First consider **nthFileInStep()**. By assumed proper operation of **with** and Assertion EXT2FS-n, **dirent** is the **nth** directory entry in the directory indicated by **DirectoryID**. By assumed proper operation of **if/else**, **@sys.add** and **@sys.sub**, Assertion EXT2FS-e and Assertion EXT2FS-n, **nthFileInStep ()** returns the location of the **nth** entry in **DirectoryID**'s

indicated directory if it is used and **count** is 0, **nthFileInStep(DirectoryID, n+1, count-1)** if it is used but **count** > 0, and **nthFileInStep(DirectoryID, n+1, count)** if it is unused.

Hence by simple recursion, **nthFileInStep()** steps through the entries in **DirectoryID**'s indicated directory, decrementing **count** each time it encounters a used entry and returning the next used entry it encounters once **count** is 0.

**nthFileIn()** returns **nthFileInStep(DirectoryID, 0, N)**, which by simple recursion returns the Nth used entry in **DirectoryID**'s indicated directory, hence the Nth file.  $\square$

#### **Assertion EXT2FS-5b**

**FileIsDirectory()** correctly returns a nonzero value iff the File ID passed to it in **FileID** is the "magic number" for the root directory or otherwise is the File ID of a directory.

```
function FileIsDirectory is integer(FileID is integer)
    with inode is ext2inode at @this.FindInodeForFileID(FileID)
    {
        @inode.IsDirectory()
    }
```

#### **Demonstration**

By Assertion EXT2FS-m and assumed proper operation of **with**, **inode** is an **ext2inode** structure describing the inode associated with **FileID**'s file.

By Assertion EXT2FS-c, **FileIsDirectory()** returns true iff **inode** refers to a directory.  $\square$

### **Assertion EXT2FS-6**

FileIsRegular() correctly returns a nonzero value iff the File ID passed to it in FileID is the File ID of a regular sequential file.

```
function FileIsRegular is integer(FileID is integer)
    with inode is ext2inode at @this.FindInodeForFileID(FileID)
    {
        @inode.IsRegularFile()
    }
```

### **Demonstration**

By Assertion EXT2FS-m and assumed proper operation of **with**, **inode** is an **ext2inode** structure describing the inode associated with **FileID**'s file.

By Assertion EXT2FS-c, **FileIsRegular()** returns true iff **inode** refers to a regular sequential file. □

### **Assertion EXT2FS-7**

FileSize() correctly returns the size of the regular file whose File ID is passed to it in FileID.

```

function FileSize is integer(FileID is integer)

    with inode is ext2inode at @this.FindInodeForFileID(FileID)

    {

        @inode.Size()

    }

```

### **Demonstration**

By Assertion EXT2FS-m and assumed proper operation of **with**, **inode** is an **ext2inode** structure describing the inode associated with **FileID**'s file.

By Assertion EXT2FS-c, **FileSize()** returns the size of the file to which **inode** refers. □

### **Assertion EXT2FS-8**

**FileName()** correctly returns the name of the regular file or non-root directory whose File ID is passed to it in **FileID**.

```

function FileName is char(FileID is integer)

    with dirent is ext2dirent at FileID

    {

        @dirent.Name()

    }

```

### **Demonstration**

By definition of an EXT2FS File ID and assumed proper operation of **with**, **dirent** is an **ext2dirent** structure describing the directory entry associated with **FileID**'s file.

By Assertion EXT2FS-e, **FileName()** returns the name of the file to which **dirent** refers.

□

### **Assertion EXT2FS-9-1**

FileHasEvenBlocks() returns a nonzero value iff the regular file associated with the passed FileID has a file size integrally divisible by the block size.

```
function FileHasEvenBlocks is integer(FileID is integer) {  
    if(@sys.grt(@sys.mod(@this.FileSize(FileID),  
                        @this.BlockSize()), 0)) { 0 }  
    else { 1 }  
}
```

### **Demonstration**

By Assertion EXT2FS-7, **FileSize(FileID)** returns the size of the regular file associated with **FileID**. By Assertion EXT2FS-1, **BlockSize()** returns the block size. By assumption of proper implementation of **@sys.mod**, **@sys.mod(@this.FileSize(FileID), @this.BlockSize())** returns greater than zero if and only if the file size is not integrally divisible by the block size. Hence by assumption of proper implementation of **if** and **@sys.grt**, **FileHasEvenBlocks()** returns nonzero iff the file size is integrally divisible by the block size.  $\square$

### **Assertion EXT2FS-9-2**

BlocksInFile() returns the number of blocks in the regular file associated with the passed FileID.

```

function BlocksInFile is integer(FileID is integer) {
    @sys.add(@sys.div(@this.FileSize(FileID),
        @this.BlockSize()),
        if(@this.FileHasEvenBlocks(FileID)) { 0 } else { 1 }
    )
}

```

### **Demonstration**

By arithmetic, a given file has  $n$  blocks where  $n$  equals the ceiling of the file size  $f$  divided by the block size  $b$ .

By assumption of proper operation of **@sys.div**, Assertion EXT2FS-7 and Assertion EXT2FS-1, **@sys.div(@this.FileSize(FileID), @this.BlockSize())**  $= f / b$ . We can obtain the ceiling by adding 1 iff  $f \bmod b > 0$ ; hence by assumption of proper operation of **if** and **@sys.add**, and Assertion EXT2FS-9-1, **BlocksInFile** returns  $\lceil f / b \rceil$  as desired.  $\square$

### **Assertion EXT2FS-9-3**

SizeOfBlock() returns the size of the  $n$ th block of the file indicated by the FileID passed to it.

```

function SizeOfBlock is integer(FileID is integer, n is integer) {
    if(@sys.and(@sys.equ(@sys.add(n, 1), @this.BlocksInFile(FileID)),
        @sys.not(@this.FileHasEvenBlocks(FileID)))) {
        @sys.mod(@this.FileSize(FileID), @this.BlockSize())
    } else { @this.BlockSize() }
}

```

By arithmetic, a given file has  $n$  blocks where  $n$  equals the ceiling of the file size  $f$  divided by the block size  $b$ . The file's last block has the partition's full block size iff  $f \bmod b = 0$ ; otherwise, it has size  $f \bmod b$ . Hence, we need to show two cases:

**Case 1:  $n$  is the last block of the file associated with FileID, and the file's size is not integrally divisible by the block size.**

Given that  $n$  is the last block of FileID's file, by Assertion EXT2FS-9-2,  $n = (\text{BlocksInFile}(\text{FileID}) - 1)$ .

Given that FileID's file's size is not integrally divisible by the block size, by Assertion EXT2FS-9-1, **FileHasEvenBlocks(FileID)** returns zero.

Hence by assumed proper operation of **if**, **@sys.and**, **@sys.equ**, **@sys.add** and **@sys.not**, **SizeOfBlock()** returns **@sys.mod(@this.FileSize(FileID), @this.BlockSize())**.

Hence by Assertion EXT2FS-9-2 and assumed proper operation of **@sys.mod**, **SizeOfBlock()** returns  $f \bmod b$  where  $f$  is the file size and  $b$  is the fundamental block size, as desired.

**Case 2: Otherwise.**

By assumption of proper operation of **else**, **SizeOfBlock()** returns **@this.BlockSize()**.

Hence by Assertion EXT2FS-1, **FileBlock()** returns the fundamental block size, as desired.  $\square$

### **Assertion EXT2FS-9**

FileBlock() correctly returns the BlockNumth block of data of the regular file whose File ID is passed to it in FileID.

```
function FileBlock is char(FileID is integer, BlockNum is integer)
    with dirent is ext2dirent at FileID
{
    @sys.charAt(
        @sys.mul(
            @this.TranslateBlock(
                @this.FindInodeNumber(@dirent.Inode()), BlockNum),
            @this.BlockSize()),
        @this.SizeOfBlock(FileID, BlockNum))
}
```

### **Demonstration**

By definition of an EXT2FS File ID and assumed proper operation of **with, dirent** is an **ext2dirent** structure describing the directory entry associated with **FileID**'s file.

By Assertions EXT2FS-j, EXT2FS-k and EXT2FS-e, **@TranslateBlock(@InodeNumber(@dirent.Inode()), BlockNum)** is the partition block number of the **BlockNumth** block of **FileID**'s file.

Hence by assumed proper operation of **@sys.mul** and **@sys.charAt**, and Assertion EXT2FS-1, **FileBlock()** returns the data beginning at the partition block number of the **BlockNumth** block of data in **FileID**'s file, times the block size. By definition of ext2fs partition blocks this is the offset of the **BlockNumth** block of data in **FileID**'s file.



By assertion EXT2FS-9-3, `@this.SizeOfBlock(FileID, n)` is the size of the **n**th block of **FileID**'s file. Hence by assumed proper operation of `@sys.charAt, FileBlock()` returns data of a size equivalent to the size of the **n**th block of **FileID**'s file.  $\square$

## **Conclusion**

Result HSL-EXT2FS: By Assertions EXT2FS-1, EXT2FS-2b, EXT2FS-3b, EXT2FS-4b, EXT2FS-5b, EXT2FS-6, EXT2FS-7, EXT2FS-8, EXT2FS-9 and Conclusion 2, the **ext2fs** HSL specification properly reads the Second Extended File System as defined above.

## CHAPTER FIVE: CONCLUSION

### Summary

We have discussed and researched the lack of formalization of I/O subsystems in modern operating systems, and concluded that it is problematic for operating systems designers, digital investigators, and computer scientists as a whole.

We have proposed, based on the OSI model of networking, a generalized, layered translation model of input and output, based on control of the data flow between layers of a computer and layers within the computer's operating system. We have suggested that I/O can be defined in terms of translation steps between those layers.

We have presented the Hadley Specification Language as an initial tool to enable such definitions to be constructive. Within this tool, we have written functions to define translations of the FAT12 and Second Extended file systems from raw partition data to directory hierarchies containing sequential files, and demonstrated their effectiveness in operation.

We have demonstrated the above translations to be correct, demonstrating the verifiability of specifications written in HSL and the verifiability of I/O as constructed using the Hadley approach.

## **Future Directions**

### **The Model**

One major area of future work is extension of the Hadley model, to include further exploration of I/O subtypes amenable to the translation layer model and their incorporation, either by explanation or by extension, into the Hadley model itself.

### **The System**

The second major area for future work is extension of HSL. In its current state, it is significant, but not nearly sufficient to be used as an operating system's sole method of input and output. At present, every place in the code where a choice between efficiency and clarity presented itself, clarity was chosen, so there are major opportunities for optimization of the code. Extension of the language to include time primitives will allow a far greater number of devices, virtual and otherwise, to be represented, as well as allow HSL to begin to communicate directly with hardware.

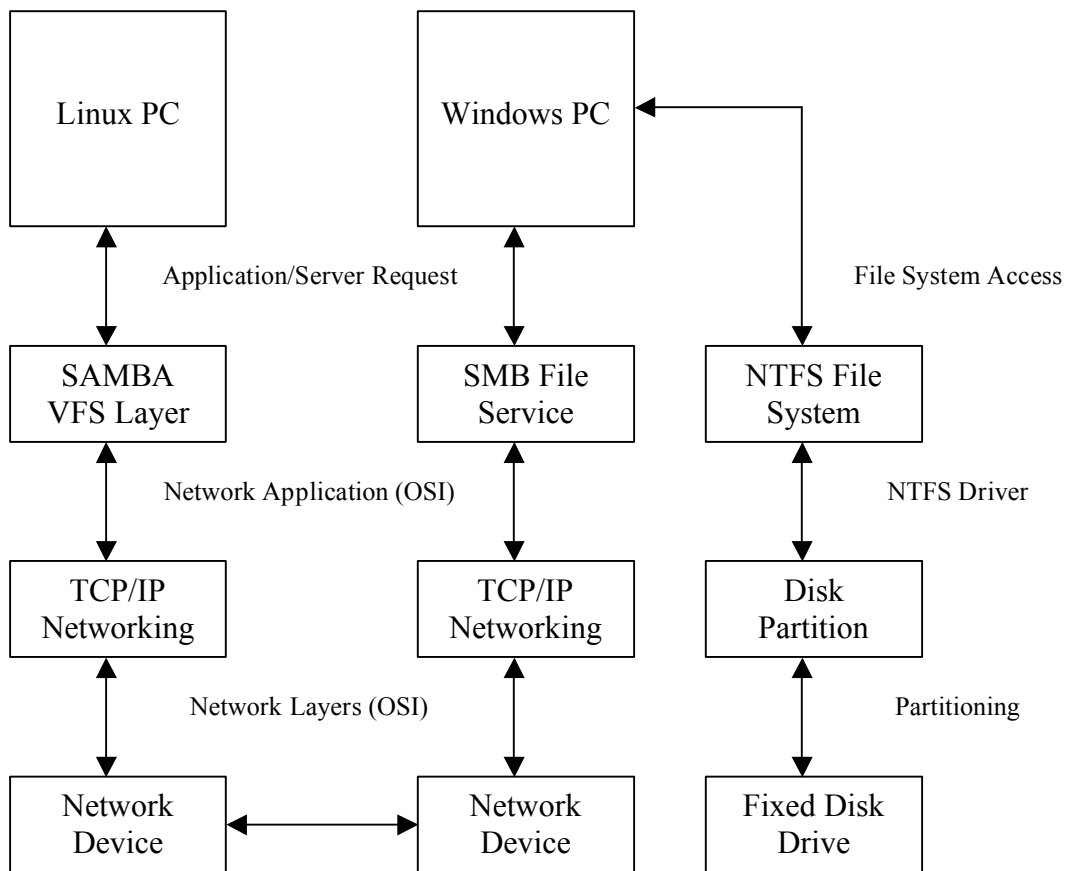


Figure 6: Networked File Access

Figure 6 shows an example of a request that is common enough to be taken for granted, but actually quite complex: a Linux PC accessing a Windows PC's file system over a network. The requests pass through file system to network translators, the networking subsystem, and the Windows PC's file system access methods. Every one of these transactions can be modeled with a combination of the Hadley model as it exists and the OSI model – the Hadley software I/O layers for the VFS and SMB file services, the Hadley software and hardware I/O layers for the Windows PC's file access, and the OSI model for the networking requests. (It is worth noting that when it comes time to model networking for Hadley, we are very likely to simply use OSI. Where a perfectly good layer model already exists, we should use it.) Handling the necessary

translations and compositions for complex requests like this should be the Hadley system's eventual goal.

Concurrency will be a major issue for HSL in the future. It does not come naturally to a functional language designed from the ground up to see all operations as atomic. Initially we will handle the overall reliability issues with aggressive locking, but this is a workaround, not a solution. Aggressive atomicity requirements for writing – such as requiring all written lists to leave a file system in a valid state when they are complete – may be a more workable approach. These are initial ideas; concurrency is widely known as a major, live field of study in and of itself, and determining how to best apply its principles to a new form of I/O subsystem will in turn be a major project in and of itself.

### **The Tool**

The third major area of future work is Hadley as a forensic investigative tool. In this area, it is closest to its goal.

Fundamentally and simply, digital forensic investigators need three things:

- To get hold of any conceivable form of computer-sensible data,
- To not modify it while they get hold of it, and
- To be able to convince a judge and jury they got hold of it right and didn't modify it.

The Hadley extractors, and even the Hadley VFS module once it is stabilized, are well designed for all three of these goals. The Hadley system supports any file systems that specifications have been written for, pulls all the standard data out of those file systems, has been

shown to do this right in a formal sense, and can easily be completely forbidden to do any writing whatsoever.

The most obvious extension needed is more supported file systems; the tool will not be taken seriously until it supports NTFS and, probably, HFS+ as well.

The tool must also be extended to handle at least some cases of “slack space data” or data between the end of a file and the end of its last block, lost block chains and inodes, and hidden information.

## **APPENDIX A: HSL SPECIFICATIONS**

```
-----
Source file: ext2.hsl
-----
```

```
spacetype ext2filemode {
    # Regular File, Directory, Character Device, FIFO

    var FileMode          is public integer littleend width 4 at 1 minor 4.

    # SetUID, SetGID, Sticky
    # These bits might well be wrong.

    var ISUID              is public integer littleend width 1 at 1 minor 3.
    var ISGID              is public integer littleend width 1 at 1 minor 2.
    var ISVTX              is public integer littleend width 1 at 1 minor 1.

    # UNIX permission bits.  These are right.

    var IRUSR              is public integer littleend width 1 at 1 minor 0.
    var IWUSR              is public integer littleend width 1 at 0 minor 7.
    var IXUSR              is public integer littleend width 1 at 0 minor 6.
    var IRGRP              is public integer littleend width 1 at 0 minor 5.
    var IWGRP              is public integer littleend width 1 at 0 minor 4.
    var IXGRP              is public integer littleend width 1 at 0 minor 3.
    var IROTH              is public integer littleend width 1 at 0 minor 2.
    var IWOTH              is public integer littleend width 1 at 0 minor 1.
    var IXOTH              is public integer littleend width 1 at 0 minor 0.

    function IsSocket      is integer() { @sys.equ($FileMode, 12) }
    function IsSymLink     is integer() { @sys.equ($FileMode, 10) }
    function IsRegularFile is integer() { @sys.equ($FileMode, 8) }
    function IsBlockDevice is integer() { @sys.equ($FileMode, 6) }
    function IsDirectory   is integer() { @sys.equ($FileMode, 4) }
    function IsCharDevice  is integer() { @sys.equ($FileMode, 2) }
    function IsFIFO        is integer() { @sys.equ($FileMode, 1) }
}

spacetype ext2superblock {
    var InodesCount      is public integer littleend width 32 at 0.
    var BlocksCount      is public integer littleend width 32 at 4.
    var RBlocksCount     is public integer littleend width 32 at 8.
    var FreeBlocksCount  is public integer littleend width 32 at 12.
    var FreeInodesCount  is public integer littleend width 32 at 16.
    var FirstDataBlock   is public integer littleend width 32 at 20.
    var LogBlockSize     is public integer littleend width 32 at 24.
    var LogFragSize      is public integer littleend width 32 at 28.
    var BlocksPerGroup   is public integer littleend width 32 at 32.
    var FragsPerGroup    is public integer littleend width 32 at 36.
    var InodesPerGroup   is public integer littleend width 32 at 40.
    var Mtime            is public integer littleend width 32 at 44.
    var Wtime            is public integer littleend width 32 at 48.
    var MntCnt           is public integer littleend width 16 at 52.
    var MaxMntCount      is public integer littleend width 16 at 54.
    var Magic            is public integer littleend width 16 at 56.
    var State            is public integer littleend width 16 at 58.
    var Errors           is public integer littleend width 16 at 60.
    var Pad              is public integer littleend width 16 at 62.
    var LastCheck        is public integer littleend width 32 at 64.
    var CheckInterval    is public integer littleend width 32 at 68.
    var CreatorOS        is public integer littleend width 32 at 72.
    var RevLevel         is public integer littleend width 32 at 76.
}

spacetype ext2groupdescriptor {
    var BlockBitmap      is public integer littleend width 32 at 0.
```



```

var InodeBitmap      is public integer littleend width 32 at 4.
var InodeTable       is public integer littleend width 32 at 8.
var FreeBlocksCount  is public integer littleend width 16 at 12.
var FreeInodesCount  is public integer littleend width 16 at 14.
var UsedDirsCount    is public integer littleend width 16 at 16.
}

spacetype ext2inode {
    use ext2filemode.

    subspace Mode      is ext2filemode at 0.

    var UID            is public integer littleend width 16 at 2.
    var Size           is public integer littleend width 32 at 4.
    var ATime          is public integer littleend width 32 at 8.
    var CTime          is public integer littleend width 32 at 12.
    var MTime          is public integer littleend width 32 at 16.
    var DTime          is public integer littleend width 32 at 20.
    var GID            is public integer littleend width 16 at 24.
    var LinksCount     is public integer littleend width 16 at 26.
    var Blocks         is public integer littleend width 32 at 28.
    var Flags          is public integer littleend width 32 at 32.
    var Version        is public integer littleend width 32 at 100.
    var FileACL        is public integer littleend width 32 at 104.
    var DirACL         is public integer littleend width 32 at 108.
    var FAddr          is public integer littleend width 32 at 112.
    var Frag           is public integer littleend width 8  at 116.
    var FSize          is public integer littleend width 8  at 117.

    var Block          is public integer[15] littleend width 32 at 40.

    function IsSocket      is integer() { @Mode.IsSocket() }
    function IsSymLink     is integer() { @Mode.IsSymLink() }
    function IsBlockDevice is integer() { @Mode.IsBlockDevice() }

    function IsRegularFile is integer() { @Mode.IsRegularFile() }
    function IsDirectory   is integer() { @Mode.IsDirectory() }
    function IsCharDevice  is integer() { @Mode.IsCharDevice() }
    function IsFIFO        is integer() { @Mode.IsFIFO() }

    function IsSticky      is integer() { @Mode.ISVTX() }
    function IsSetUID      is integer() { @Mode.ISUID() }
    function IsSetGID      is integer() { @Mode.ISGID() }

    function OtherCanRead  is integer() { @Mode.IROTH() }
    function OtherCanWrite is integer() { @Mode.IWOTH() }
    function OtherCanExec  is integer() { @Mode.IXOTH() }
    function GroupCanRead  is integer() { @Mode.IRGRP() }
    function GroupCanWrite is integer() { @Mode.IWGRP() }
    function GroupCanExec  is integer() { @Mode.IXGRP() }
    function UserCanRead   is integer() { @Mode.IRUSR() }
    function UserCanWrite  is integer() { @Mode.IWUSR() }
    function UserCanExec   is integer() { @Mode.IXUSR() }
}

spacetype ext2dirent {
    var Inode          is public integer littleend width 32 at 0.
    var RecordLength   is public integer littleend width 16 at 4.
    var NameLength     is public integer littleend width 8  at 6.
    var Name           is char[$NameLength] at 8.

    function Name is char() {
        $Name<0, $NameLength>
    }

    function IsUsed is integer() {
        @sys.not(@sys.equ($Inode, 0))
    }
}

```

```

spacetype ext2 {
    use ext2dirent.
    use ext2inode.
    use ext2superblock.
    use ext2groupdescriptor.

    subspace superblock is ext2superblock at 1024.

    # Fundamental block size.

    function BlockSize is integer() {
        @sys.mul(1024, @sys.2to(@superblock.LogBlockSize()))
    }

    function BlockSize4 is integer() {
        @sys.div(@this.BlockSize(), 4)
    }

    # Inode indirect addressing translation.

    function IntExtract is integer(n is integer, k is integer) {
        @sys.intAt(@sys.add(@sys.mul(n, @this.BlockSize()),
            @sys.mul(k, 4)), 0, 32, 0)
    }

    function SIndMax is integer() {
        @sys.add(@this.BlockSize4(), 12)
    }

    function DIndMax is integer() {
        @sys.add(@sys.mul(@this.BlockSize4(), @this.BlockSize4()), @this.SIndMax())
    }

    function DoubleIndirect is integer(n is integer, k is integer) {
        @this.IntExtract(
            @this.IntExtract(n, @sys.div(k, @this.BlockSize4())),
            @sys.mod(k, @this.BlockSize4())
        )
    }

    function TripleIndirect is integer(n is integer, k is integer) {
        @this.IntExtract(
            @this.IntExtract(
                @this.IntExtract(
                    n,
                    @sys.div(@sys.div(k, @this.BlockSize4()), @this.BlockSize4())
                ),
                @sys.mod(@sys.div(k, @this.BlockSize4()), @this.BlockSize4())
            ),
            @sys.mod(k, @this.BlockSize4())
        )
    }

    function TranslateBlock is integer(inodeloc is integer, n is integer)
    with inode is ext2inode at inodeloc
    {
        if(@sys.lst(n, 12)) { @inode.Block(n) } else {
            if(@sys.lst(n, @this.SIndMax())) {
                @this.IntExtract(@inode.Block(12), @sys.sub(n, 12))
            } else {
                if(@sys.lst(n, @this.DIndMax())) {
                    @this.DoubleIndirect(@inode.Block(13),
                        @sys.sub(n, @this.SIndMax()))
                } else {
                    @this.TripleIndirect(@inode.Block(14),
                        @sys.sub(n, @this.DIndMax()))
                }
            }
        }
    }
}

```

```

    }
}

function FindInodeNumber is integer (InodeID is integer)
    with groupdescriptor is ext2groupdescriptor at
        @sys.add(2048, @sys.mul(32, @sys.div(@sys.sub(InodeID, 1),
@superblock.InodesPerGroup()))))
    {
        @sys.add(
            @sys.mul(@groupdescriptor.InodeTable(), @this.BlockSize()),
            @sys.mul(@sys.mod(@sys.sub(InodeID, 1), @superblock.InodesPerGroup()), 128)
        )
    }

function RootDirectoryID is integer() {
    1
}

function GetRootDirectoryInodeLoc is integer() {
    @this.FindInodeNumber(2)
}

function FindInodeForFileID is integer (FileID is integer)
    with dirent is ext2dirent at FileID
    {
        if(@sys.equ(FileID, @this.RootDirectoryID())) {
            @this.GetRootDirectoryInodeLoc()
        } else {
            @this.FindInodeNumber(@dirent.Inode())
        }
    }
}

function DirectoryEntryLocStep is integer(inodeloc is integer, count is integer,
                                           block is integer, loc is integer)
    with inode is ext2inode at inodeloc,
        dirent is ext2dirent at
            @sys.add(@sys.mul(@this.TranslateBlock(inodeloc, block), @this.BlockSize()),
loc)
    {
        if(@sys.equ(block, @sys.div(@inode.Blocks(), 1))) {
            0
        } else {
            if(@sys.equ(count, 0)) {
                @sys.add(@sys.mul(@this.TranslateBlock(inodeloc, block), @this.BlockSize()), loc)
            } else {
                if(@sys.equ(@sys.add(loc, @dirent.RecordLength()), @this.BlockSize())) {
                    @this.DirectoryEntryLocStep(inodeloc, @sys.sub(count, 1), @sys.add(block, 1),
0)
                } else {
                    @this.DirectoryEntryLocStep(inodeloc, @sys.sub(count, 1), block,
@sys.add(loc, @dirent.RecordLength()))
                }
            }
        }
    }
}

function DirectoryEntryLoc is integer(DirectoryID is integer, n is integer)
    with dirent is ext2dirent at DirectoryID
    {
        @this.DirectoryEntryLocStep(@this.FindInodeForFileID(DirectoryID), n, 0, 0)
    }
}

function FilesInStep is integer(DirectoryID is integer, n is integer,
                                count is integer)
    with dirent is ext2dirent at @this.DirectoryEntryLoc(DirectoryID, n)
    {
        if(@sys.equ(@this.DirectoryEntryLoc(DirectoryID, n), 0)) {
            count
        } else {

```

```

        @this.FilesInStep(DirectoryID, @sys.add(n, 1),
        if(@dirent.IsUsed()) { @sys.add(count, 1) } else { count })
    }
}

function FilesIn is integer(DirectoryID is integer)
{
    @this.FilesInStep(DirectoryID, 0, 0)
}

function nthFileInStep is integer(DirectoryID is integer, n is integer,
                                count is integer)
with dirent is ext2dirent at @this.DirectoryEntryLoc(DirectoryID, n)
{
    if(@dirent.IsUsed()) {
        if(@sys.equ(count, 0)) {
            @this.DirectoryEntryLoc(DirectoryID, n)
        } else {
            @this.nthFileInStep(DirectoryID, @sys.add(n, 1), @sys.sub(count, 1))
        }
    } else {
        @this.nthFileInStep(DirectoryID, @sys.add(n, 1), count)
    }
}

function nthFileIn is integer(DirectoryID is integer, n is integer)
{
    @this.nthFileInStep(DirectoryID, 0, n)
}

function FileName is char(FileID is integer)
with dirent is ext2dirent at FileID
{
    @dirent.Name()
}

function FileSize is integer(FileID is integer)
with dirent is ext2dirent at FileID,
inode is ext2inode at @this.FindInodeNumber(@dirent.Inode())
{
    @inode.Size()
}

function FileIsDirectory is integer(FileID is integer)
with inode is ext2inode at @this.FindInodeForFileID(FileID)
{
    @inode.IsDirectory()
}

function FileIsRegular is integer(FileID is integer)
with inode is ext2inode at @this.FindInodeForFileID(FileID)
{
    @inode.IsRegularFile()
}

function FileHasEvenBlocks is integer(FileID is integer) {
    if(@sys.grt(@sys.mod(@this.FileSize(FileID), @this.BlockSize()), 0)) { 0 } else { 1 }
}

function BlocksInFile is integer(FileID is integer) {
    @sys.add(@sys.div(@this.FileSize(FileID), @this.BlockSize()),
    if(@this.FileHasEvenBlocks(FileID)) { 0 } else { 1 }
    )
}

function SizeOfBlock is integer(FileID is integer, n is integer) {
    if(@sys.and(@sys.equ(@sys.add(n, 1), @this.BlocksInFile(FileID)),
    @sys.not(@this.FileHasEvenBlocks(FileID)))) {
        @sys.mod(@this.FileSize(FileID), @this.BlockSize())
    }
}

```

```

    } else { @this.BlockSize() }
}

function FileBlock is char(FileID is integer, n is integer)
    with dirent is ext2dirent at FileID
{
    @sys.charAt(
        @sys.mul(@this.TranslateBlock(@this.FindInodeNumber(@dirent.Inode()), n),
            @this.BlockSize()), @this.SizeOfBlock(FileID, n))
}

function FileSlack is char(FileID is integer)
    with dirent is ext2dirent at FileID,
        inode is ext2inode at @this.FindInodeNumber(@dirent.Inode())
{
    @sys.charAt(
        @sys.add(
            @sys.mul(@this.TranslateBlock(@this.FindInodeNumber(@dirent.Inode()),
                @sys.sub(@this.BlocksInFile(FileID), 1)),
                @this.BlockSize()),
            @this.SizeOfBlock(FileID,
                @sys.sub(@this.BlocksInFile(FileID), 1))
        ),
        @sys.sub(@this.BlockSize(),
            @this.SizeOfBlock(FileID,
                @sys.sub(@this.BlocksInFile(FileID), 1)))
    )
}

function SupportsUGO is integer() { 1 }
function SupportsFSW is integer() { 0 }

function UstrRead is integer(FileID is integer)
    with inode is ext2inode at @this.FindInodeForFileID(FileID)
{ @inode.UserCanRead() }

function UstrWrit is integer(FileID is integer)
    with inode is ext2inode at @this.FindInodeForFileID(FileID)
{ @inode.UserCanWrite() }

function UstrExec is integer(FileID is integer)
    with inode is ext2inode at @this.FindInodeForFileID(FileID)
{ @inode.UserCanExec() }

function GrpRead is integer(FileID is integer)
    with inode is ext2inode at @this.FindInodeForFileID(FileID)
{ @inode.GroupCanRead() }

function GrpWrit is integer(FileID is integer)
    with inode is ext2inode at @this.FindInodeForFileID(FileID)
{ @inode.GroupCanWrite() }

function GrpExec is integer(FileID is integer)
    with inode is ext2inode at @this.FindInodeForFileID(FileID)
{ @inode.GroupCanExec() }

function OthRead is integer(FileID is integer)
    with inode is ext2inode at @this.FindInodeForFileID(FileID)
{ @inode.OtherCanRead() }

function OthWrit is integer(FileID is integer)
    with inode is ext2inode at @this.FindInodeForFileID(FileID)
{ @inode.OtherCanWrite() }

function OthExec is integer(FileID is integer)
    with inode is ext2inode at @this.FindInodeForFileID(FileID)
{ @inode.OtherCanExec() }

function Sticky is integer(FileID is integer)

```

```

        with inode is ext2inode at @this.FindInodeForFileID(FileID)
    { @inode.IsSticky() }

function SetUID is integer(FileID is integer)
    with inode is ext2inode at @this.FindInodeForFileID(FileID)
    { @inode.IsSetUID() }

function SetGID is integer(FileID is integer)
    with inode is ext2inode at @this.FindInodeForFileID(FileID)
    { @inode.IsSetGID() }

implement hadleyfs {
    BlockSize, RootDirectoryID, FilesIn, nthFileIn, FileIsDirectory,
    FileIsRegular, FileSize, FileName, FileBlock, SupportsUGO, SupportsFSW
}

implement hadleyugo {
    UsrRead, GrpRead, OthRead, UsrWrit, GrpWrit, OthWrit,
    UsrExec, GrpExec, OthExec, Sticky, SetUID, SetGID
}
}

-----
Source file: fat12.hsl
-----

spacetype fat12dirent {
    var FileNameFirstChar is integer littleend width 8 at 0.
    var FileName          is char[8] at 0.
    var FileExt           is char[3] at 8.
    var FileAttr          is public integer littleend width 8 at 11.
    var FileIsReadOnly    is public integer littleend width 1 at 11 minor 0.
    var FileIsHidden      is public integer littleend width 1 at 11 minor 1.
    var FileIsSystem      is public integer littleend width 1 at 11 minor 2.
    var FileIsVolumeLabel is public integer littleend width 1 at 11 minor 3.
    var FileIsDirectory   is public integer littleend width 1 at 11 minor 4.
    var FileIsArchive     is public integer littleend width 1 at 11 minor 5.
    var FirstBlock        is public integer littleend width 16 at 26.
    var FileSize          is public integer littleend width 32 at 28.

    # FAT12 filenames aren't null-terminated strings, which makes this a royal
    # pain. We have to figure out the length of the filename, figure out the
    # length of the extension, and concatenate them.

    # Here are the functions for the name.

    function FNLengthStep is integer(k is integer) {
        if(@sys.equ(k, 8)) { 8 }
        else {
            if(@sys.clipequ($FileName[k], " ")) { k }
            else { @this.FNLengthStep(@sys.add(k, 1)) }
        }
    }

    function FNLength is integer() { @this.FNLengthStep(0) }
    function FileNamePart is char() { $FileName<0, @this.FNLength(> }

    # And here are the functions for the extension.

    function FLengthStep is integer(k is integer) {
        if(@sys.equ(k, 3)) { 3 }
        else {
            if(@sys.clipequ($FileExt[k], " ")) { k }
            else { @this.FLengthStep(@sys.add(k, 1)) }
        }
    }

    function FLength is integer() { @this.FLengthStep(0) }
    function HasExt is integer() { @sys.not(@sys.equ(@this.FLength(), 0)) }

```

```

function FileExtPart is char() {
    if(@this.HasExt()) {
        @sys.clipcat(".", $FileExt<0, @this.FELength())>
    } else { "" }
}

function FileName is char() {
    @sys.clipcat(@this.FileNamePart(), @this.FileExtPart())
}

function IsFinalEnt is integer() { @sys.equ($FileNameFirstChar, 0) }
function IsDotEnt is integer() { @sys.equ($FileNameFirstChar, 46) }
function IsErased is integer() { @sys.equ($FileNameFirstChar, 229) }

# WRITING FUNCTIONS

function NullName is list() {
    @sys.listadd(
        @sys.newlist(@sys.makecharhint(" ", &!FileName)),
        @sys.makecharhint(" ", &!FileExt))
}

function SetNameParts is list(Name is char, Ext is char) {
    @sys.listaddend(@sys.listaddend(
        @this.NullName(),
        @sys.makecharhint(Name, &!FileName)),
        @sys.makecharhint(Ext, &!FileExt))
}

function SetName is list(Name is char) {
    if(@sys.clipclip(Name, ".") {
        @this.SetNameParts(@sys.subclip(Name, 0, @sys.clipclip(Name, ".")),
            @sys.subclip(Name, @sys.clipclip(Name, "."),
                @sys.sub(@sys.cliplen(Name), @sys.clipclip(Name, ".))))
    } else {
        @this.SetNameParts(Name, "")
    }
}

function SetSize is list(Size is integer) {
    @sys.newlist(@sys.makeinthint(Size, &!FileSize, 0, 32, LITTLEEND))
}

function SetFirstBlock is list(BlockNum is integer) {
    @sys.newlist(@sys.makeinthint(BlockNum, &!FirstBlock, 0, 16, LITTLEEND))
}

}

spacetype fat12 {
    use fat12dirent.

    var BytesPerSector          is integer littleend width 16 at 11.
    var SectorsPerBlock         is integer littleend width 8  at 13.
    var ReservedSectors         is integer littleend width 16 at 14.
    var NumberOfFats            is integer littleend width 8  at 16.
    var MaxRootDirEntries       is integer littleend width 16 at 17.
    var SectorsInPartition      is integer littleend width 16 at 19.
    var MediaDescriptor         is integer littleend width 8  at 21.
    var SectorsPerFAT           is integer littleend width 16 at 22.
    var SectorsPerTrack         is integer littleend width 16 at 24.
    var Heads                   is integer littleend width 16 at 26.
    var HiddenSectors           is integer littleend width 32 at 28.
    var SectorsInPartBig        is integer littleend width 32 at 32.
    var LogicalDriveNumber      is integer littleend width 16 at 36.
    var ExtendedSignature       is integer littleend width 8  at 38.
    var SerialNumber            is integer littleend width 32 at 39.
    var VolumeName              is char[11]                  at 43.

```

```

var BootCode          is char[448]          at 51.
var BootCodeMarker    is integer littleend width 16 at 499.

function RootDirectorySize is integer() {
    @sys.mul($MaxRootDirEntries, 32)
}

function RootDirectoryStart is integer() {
    @sys.mul(@sys.add($ReservedSectors,
        @sys.mul($NumberOfFats, $SectorsPerFAT)),
        $BytesPerSector)
}

function data is integer(k is integer) {
    @sys.add(k, @sys.add(@this.RootDirectoryStart(), @this.RootDirectorySize()))
}

function NonDataSectors is integer() {
    @sys.div(@this.data(0), $BytesPerSector)
}

function DataSectors is integer() {
    @sys.sub($SectorsInPartition, @this.NonDataSectors())
}

function DataBlocks is integer() {
    @sys.div(@this.DataSectors(), $SectorsPerBlock)
}

var FAT is integer[@sys.add(@this.DataBlocks(), 2)]
    littleend width 12 at
    @sys.mul($ReservedSectors, $BytesPerSector).

function BlockSize is integer() { @sys.mul($BytesPerSector,
    $SectorsPerBlock) }

function RootDirectoryID is integer() { 0 }

function FileAttr is integer(FileID is integer)
    with entry is fat12dirent at FileID
{ @entry.FileAttr() }

function FileName is char(FileID is integer)
    with entry is fat12dirent at FileID
{ @entry.FileName() }

function FileSize is integer(FileID is integer)
    with entry is fat12dirent at FileID
{ @entry.FileSize() }

function FileIsDirectory is integer(FileID is integer)
    with entry is fat12dirent at FileID
{ if(@sys.equ(FileID, @this.RootDirectoryID())) { 1 }
  else { @entry.FileIsDirectory() } }

function FileIsRegular is integer(FileID is integer)
    with entry is fat12dirent at FileID
{ @sys.and(@sys.not(@this.FileIsDirectory(FileID)),
    @sys.not(@entry.FileIsVolumeLabel())) }

function FileHasEvenBlocks is integer(FileID is integer) {
    if(@sys.grt(@sys.mod(@this.FileSize(FileID), @this.BlockSize()), 0)) { 0 } else { 1 }
}

function DataForBlock is char(k is integer, size is integer) {
    @sys.charAt(@this.data(@sys.mul(@sys.sub(k, 2), @this.BlockSize())), size)
}

function FirstBlock is integer(FileID is integer)

```



```

    with entry is fat12dirent at FileID
{
    @entry.FirstBlock()
}

function NextBlock is integer(k is integer) {
    if(@sys.and(@sys.lst($FAT[k], 4087), @sys.grt($FAT[k], 1))) { $FAT[k] } else { 0 }
}

function nthBlock is integer(FileID is integer, n is integer) {
    if(@sys.equ(n, 0)) { @this.FirstBlock(FileID) }
    else { @this.NextBlock(@this.nthBlock(FileID, @sys.sub(n, 1))) }
}

function BlocksInDir is integer(DirectoryID is integer, n is integer) {
    if(@sys.equ(@this.nthBlock(DirectoryID, n), 0)) { n }
    else { @this.BlocksInDir(DirectoryID, @sys.add(n, 1)) }
}

function BlocksInFile is integer(FileID is integer)
    with entry is fat12dirent at FileID
{
    if(@entry.FileIsDirectory()) {
        @this.BlocksInDir(FileID, 0)
    } else {
        @sys.add(@sys.div(@this.FileSize(FileID), @this.BlockSize()),
            if(@this.FileHasEvenBlocks(FileID)) { 0 } else { 1 }
        )
    }
}

function DirEntsPerBlock is integer() {
    @sys.div(@this.BlockSize(), 32)
}

function DirectoryEntryLoc is integer(DirectoryID is integer, n is integer) {
    if(@sys.equ(DirectoryID, 0)) {
        @sys.add(@this.RootDirectoryStart(), @sys.mul(n, 32))
    }
    else { @this.data(@sys.add(
        @sys.mul(@sys.sub(@this.nthBlock(DirectoryID, @sys.div(n, @this.DirEntsPerBlock())),
            @this.BlockSize()),
        @sys.mul(@sys.mod(n, @this.DirEntsPerBlock()), 32)
        )) }
}

function DirectoryEntryUsed is integer(DirectoryID is integer, n is integer)
    with entry is fat12dirent at @this.DirectoryEntryLoc(DirectoryID, n)
{
    if( @sys.or(@sys.equ(@entry.FileAttr(), 15),
        @sys.or(@entry.IsFinalEnt(),
            @entry.IsErased())) { 0 }
    else { 1 }
}

function DirectoryEntryFree is integer(DirectoryID is integer, n is integer)
    with entry is fat12dirent at @this.DirectoryEntryLoc(DirectoryID, n)
{
    if( @sys.or(@sys.equ(@entry.FileAttr(), 15),
        @sys.or(@entry.IsFinalEnt(),
            @sys.or(@entry.IsDotEnt(),
                @entry.IsErased())) { 0 }
    else { 1 }
}

function FilesInStep is integer (DirectoryID is integer, n is integer,
    count is integer)
    with entry is fat12dirent at @this.DirectoryEntryLoc(DirectoryID, n)

```

```

{
    if(@entry.IsFinalEnt()) { count }
    else { @this.FilesInStep(DirectoryID, @sys.add(n, 1),
        if(@this.DirectoryEntryUsed(DirectoryID, n)) { @sys.add(count, 1) }
        else {count})
    }
}

function FileStep is integer(DirectoryID is integer, n is integer, count is integer) {
    if(@this.DirectoryEntryUsed(DirectoryID, n)) {
        if (@sys.grt(count, 0)) {
            @this.FileStep(DirectoryID, @sys.add(n, 1), @sys.sub(count, 1))
        } else {
            @this.DirectoryEntryLoc(DirectoryID, n)
        }
    } else {
        @this.FileStep(DirectoryID, @sys.add(n, 1), count)
    }
}

function FileBlock is char(FileID is integer, n is integer) {
    if(@sys.and(@sys.equ(@sys.add(n, 1), @this.BlocksInFile(FileID)),
        @sys.not(@this.FileHasEvenBlocks(FileID))))
    { @this.DataForBlock(@this.nthBlock(FileID, n),
        @sys.mod(@this.FileSize(FileID), @this.BlockSize())) }
    else { @this.DataForBlock(@this.nthBlock(FileID, n), @this.BlockSize()) }
}

function FilesIn is integer(DirectoryID is integer) {
    @this.FilesInStep(DirectoryID, 0, 0)
}

function nthFileIn is integer(DirectoryID is integer, n is integer) {
    if(@sys.gre(n, @this.FilesIn(DirectoryID))) { -1 } else { @this.FileStep(DirectoryID, 0,
n) }
}

function SupportsUGO is integer() { 0 }
function SupportsFSW is integer() { 1 }

#####
# WRITING FUNCTIONS

function FirstFreeFAT is integer(step is integer) {
    # XXX: This will currently produce undefined behavior if the disk is full.
    if(@sys.equ($FAT[step], 0)) {
        step
    } else {
        @this.FirstFreeFAT(@sys.add(step, 1))
    }
}

function FirstFreeDirEnt is integer(DirectoryID is integer, step is integer)
with entry is fat12dirent at @this.DirectoryEntryLoc(DirectoryID, step)
{
    if(@entry.IsFinalEnt()) {
        @this.DirectoryEntryLoc(DirectoryID, step)
    } else {
        @this.FirstFreeDirEnt(DirectoryID, @sys.add(step, 1))
    }
}

function LastBlockOf is integer(FileID is integer)
with entry is fat12dirent at FileID
{
    if(@sys.equ(@entry.FirstBlock(), 0)) {
        -1
    } else {

```

```

        @this.nthBlock(FileID, @sys.sub(@this.BlocksInFile(FileID), 1))
    }
}

function UpdateFATs is list(ClusterNum is integer,
                           Value is integer,
                           Count is integer)
{
    if(@sys.equ(@sys.add(Count, 1), $NumberOfFats)) {
        @sys.listfrom(
            @sys.newlist(@sys.makeinthead(Value,
                                           &!FAT[ClusterNum],
                                           &.FAT[ClusterNum],
                                           12,
                                           LITTLEEND)),
            @sys.mul(@sys.mul(Count, $SectorsPerFAT), $BytesPerSector))
    } else {
        @sys.listcon(
            @sys.listfrom(
                @sys.newlist(@sys.makeinthead(Value,
                                               &!FAT[ClusterNum],
                                               &.FAT[ClusterNum],
                                               12,
                                               LITTLEEND)),
                @sys.mul(@sys.mul(Count, $SectorsPerFAT), $BytesPerSector)),
            @this.UpdateFATs(ClusterNum, Value, @sys.add(Count, 1)))
    }
}

# There are two places we might want this.  If the file doesn't have
# a last block, that means we want to write right into the directory
# entry.  On the other hand, if it has one, we need to traverse the FAT.

function IntForAppend is list(FileID is integer,
                              Block is char,
                              LastBlock is integer,
                              FirstFree is integer)
{
    with entry is fat12dirent at FileID
    {
        @sys.listcon(
            if(@sys.equ(LastBlock, -1)) {
                @sys.listfrom(@entry.SetFirstBlock(FirstFree), FileID)
            } else {
                @this.UpdateFATs(LastBlock, FirstFree, 0)
            },
            @this.UpdateFATs(FirstFree, 4095, 0))
    }
}

function AppendBlockPrime is list(FileID is integer,
                                  Block is char,
                                  LastBlock is integer,
                                  FirstFree is integer) {
    @sys.listadd(
        @this.IntForAppend(FileID, Block, LastBlock, FirstFree),
        @sys.makecharhint(Block,
                           @this.data(@sys.mul(@sys.sub(FirstFree, 2), @this.BlockSize()))))
}

function AppendBlock is list(FileID is integer, Block is char) {
    @this.AppendBlockPrime(FileID,
                           Block,
                           @this.LastBlockOf(FileID),
                           @this.FirstFreeFAT(0))
}

function ReplaceBlock is list(FileID is integer, n is integer, Block is char) {
    @sys.newlist(@sys.makecharhint(Block,
                                     @this.data(@sys.mul(@sys.sub(@this.nthBlock(FileID, n), 2), @this.BlockSize()))))
}

```

```

# 0: Entry
# 1: Directory
# 2: . entry
# 3: .. entry

function NewDirEnt is list(Location is integer, EntryType is integer) {
  @sys.newlist(@sys.makecharhint(
    if(@sys.equ(EntryType, 0)) {
      "\0          \x00\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0" 32
#      0 12345678901      2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
    } else { if(@sys.equ(EntryType, 1)) {
#      "\0          \x10\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0" 32
#      0 12345678901      2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
    } else { if(@sys.equ(EntryType, 2)) {
#      ".          \x10\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0" 32
#      012345678901      2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
    } else {
#      "..         \x10\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0" 32
#      012345678901      2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
      }}}},
    Location
  ))
}

function NewDirectoryBlock is list(Location is integer, n is integer) {
  @sys.listrepeat(@this.NewDirEnt(Location, 0),
    @this.DirEntsPerBlock(),
    32)
}

function ExpandDir is list(DirectoryID is integer, InBlock is integer) {
  @sys.listcon(
    @this.AppendBlockPrime(DirectoryID,
      "",
      @this.LastBlockOf(DirectoryID),
      InBlock),
    @this.NewDirectoryBlock(@this.data(@sys.mul(@sys.sub(InBlock, 2),
@this.BlockSize()), 0)
  )
}

function MustExpand is integer(DirectoryID is integer) {
  @sys.equ(@sys.mod(@this.FirstFreeDirEnt(DirectoryID, 0), @this.BlockSize()),
    @sys.sub(@this.BlockSize(), 32))
}

function SetName is list(FileID is integer, Name is char)
  with entry is fat12dirent at FileID
{
  @sys.listfrom(@entry.SetName(Name), FileID)
}

function SetSize is list(FileID is integer, Size is integer)
  with entry is fat12dirent at FileID
{
  @sys.listfrom(@entry.SetSize(Size), FileID)
}

function NewFile is list(DirectoryID is integer, Name is char) {
  @sys.listcon(@sys.listcon(
    @this.NewDirEnt(@this.FirstFreeDirEnt(DirectoryID, 0), 0),
    if(@this.MustExpand(DirectoryID)) {
      @this.ExpandDir(DirectoryID, @this.FirstFreeFAT(0))
    } else {
      @this.NewDirEnt(@sys.add(@this.FirstFreeDirEnt(DirectoryID, 0), 32), 0)
    }
  ),
  @this.SetName(@this.FirstFreeDirEnt(DirectoryID, 0), Name))
}

```

```

function NewDirectory is list(DirectoryID is integer, Name is char)
  with parent    is fat12dirent at DirectoryID,
  singledot     is fat12dirent
    at @this.data(@sys.mul(@sys.sub(@this.FirstFreeFAT(0), 2),
                           @this.BlockSize())),
  doubledot     is fat12dirent
    at @sys.add(@this.data(@sys.mul(@sys.sub(@this.FirstFreeFAT(0), 2),
                                       @this.BlockSize())), 32)

{
  @sys.listcon(@sys.listcon(@sys.listcon(@sys.listcon(@sys.listcon(@sys.listcon(@s
ys.listcon(
  @this.NewDirEnt(@this.FirstFreeDirEnt(DirectoryID, 0), 1),
  if(@this.MustExpand(DirectoryID)) {
    @this.ExpandDir(DirectoryID, @this.FirstFreeFAT(0))
  } else {
    @this.NewDirEnt(@sys.add(@this.FirstFreeDirEnt(DirectoryID, 0), 32), 0)
  })),
  @this.SetName(@this.FirstFreeDirEnt(DirectoryID, 0), Name)),
  @this.AppendBlockPrime(@this.FirstFreeDirEnt(DirectoryID, 0),
    "",
    -1,
    @this.FirstFreeFAT(0))),
  @this.NewDirectoryBlock(@this.data(@sys.mul(@sys.sub(@this.FirstFreeFAT(0), 2),
                                                  @this.BlockSize())), 0)),
  @this.NewDirEnt(@this.data(@sys.mul(@sys.sub(@this.FirstFreeFAT(0), 2),
                                                  @this.BlockSize())), 2)),
  @this.NewDirEnt(@sys.add(@this.data(@sys.mul(@sys.sub(@this.FirstFreeFAT(0), 2),
                                                  @this.BlockSize())), 32), 3)),
  @sys.listfrom(@singledot.SetFirstBlock(@this.data(@sys.mul(
    @sys.sub(@this.FirstFreeFAT(0), 2),
    @this.BlockSize()))),
    @this.data(@sys.mul(
      @sys.sub(@this.FirstFreeFAT(0), 2),
      @this.BlockSize()))),
  @sys.listfrom(@doubledot.SetFirstBlock(@parent.FirstBlock()),
    @sys.add(@this.data(@sys.mul(
      @sys.sub(@this.FirstFreeFAT(0), 2),
      @this.BlockSize())), 32)))
}

# IMPLEMENTATIONS

implement hadleyfs {
  BlockSize, RootDirectoryID, FilesIn, nthFileIn, FileIsDirectory,
  FileIsRegular, FileSize, FileName, FileBlock, SupportsUGO, SupportsFSW
}

implement hadleyfsw {
  NewFile, NewDirectory, SetName, SetSize, AppendBlock, ReplaceBlock
}
}

-----
Source file: hadleyfs.hsl
-----

template hadleyfs {
  function BlockSize is integer()
  function RootDirectoryID is integer()
  function FilesIn is integer(DirectoryID is integer)
  function nthFileIn is integer(DirectoryID is integer, FileNum is integer)

  function FileIsDirectory is integer(FileID is integer)
  function FileIsRegular is integer(FileID is integer)

  function FileSize is integer(FileID is integer)
  function FileName is char(FileID is integer)

```

```

    function FileBlock is char(FileID is integer, BlockNum is integer)

    function FSSupportsUGO is integer()
    function FSSupportsFSW is integer()
}

-----
Source file: hadleyfsw.hsl
-----

template hadleyfsw {
    function NewFile is list(DirectoryID is integer, Name is char)
    function NewDirectory is list(DirectoryID is integer, Name is char)

    function SetName is list(FileID is integer, Name is char)
    function SetSize is list(FileID is integer, Size is integer)

    function AppendBlock is list(FileID is integer, Block is char)
    function ReplaceBlock is list(FileID is integer, n is integer, Block is char)
}

-----
Source file: hadleyugo.hsl
-----

# The UNIX permission model.

template hadleyugo {
    function UsrRead is integer(FileID is integer)
    function GrpRead is integer(FileID is integer)
    function OthRead is integer(FileID is integer)
    function UsrWrit is integer(FileID is integer)
    function GrpWrit is integer(FileID is integer)
    function OthWrit is integer(FileID is integer)
    function UsrExec is integer(FileID is integer)
    function GrpExec is integer(FileID is integer)
    function OthExec is integer(FileID is integer)

    function Sticky is integer(FileID is integer)
    function SetUID is integer(FileID is integer)
    function SetGID is integer(FileID is integer)
}

```

## **APPENDIX B: HADLEY SYSTEM SOURCE CODE**

```
-----  
Source file: Makefile  
-----
```

```
default: all
```

```
all: core.a hslc fs.a tests  
tests: test-fat12temp test-ext2temp
```

```
clean:  
-rm ext2*.c fat12*.c hadleyfs.c hadleyfsw.c hadleyugo.c  
-rm ext2*.h fat12*.h hadleyfs.h hadleyfsw.h hadleyugo.h  
-rm *.o *.a hslc test-fat12temp test-ext2temp  
-rm hadley.output hadley.tab.c hadley.tab.h lex.yy.c
```

```
hslc: core.a hadley.tab.c hadley.tab.h lex.yy.c  
gcc -g -o hslc yacc-dummy.c hadley.tab.c lex.yy.c core.a -ll
```

```
lex.yy.c: hadley.l hadley.h  
flex hadley.l
```

```
hadley.tab.c hadley.tab.h: hadley.y hadley.h hadley-semantics.i  
bison -d -v hadley.y
```

```
core.a: hash.c core.c  
gcc -g -c core.c hash.c  
ar -r core.a core.o hash.o  
ranlib core.a
```

```
fs.a: ext2.o fat12.o hadleyfs.o hadleyugo.o hadleyfsw.o  
ar -r fs.a ext2*.o fat12*.o hadleyfs.o hadleyugo.o hadleyfsw.o  
ranlib fs.a
```

```
hadleyfsw.o: hadleyfsw.hsl hslc  
./hslc < hadleyfsw.hsl  
gcc -g -c hadleyfsw.c
```

```
hadleyfs.o: hadleyfs.hsl hslc  
./hslc < hadleyfs.hsl  
gcc -g -c hadleyfs.c
```

```
hadleyugo.o: hadleyugo.hsl hslc  
./hslc < hadleyugo.hsl  
gcc -g -c hadleyugo.c
```

```
ext2.o: ext2.hsl hadleyfs.o hadleyugo.o hadleyfsw.o hslc  
./hslc < ext2.hsl  
gcc -g -c ext2*.c
```

```
fat12.o: fat12.hsl hadleyfs.o hadleyugo.o hadleyfsw.o hslc  
./hslc < fat12.hsl  
gcc -g -c fat12*.c
```

```
test-fat12temp: core.a hslc fs.a test-fat12temp.c test-temp.c  
gcc -g -o test-fat12temp core.c hash.c test-fat12temp.c test-temp.c fs.a
```

```
test-ext2temp: core.a hslc fs.a test-ext2temp.c test-temp.c  
gcc -g -o test-ext2temp core.c hash.c test-ext2temp.c test-temp.c fs.a
```

```
-----  
Source file: hadley.l  
-----
```

```
%{  
#include "hadley.tab.h"  
#include <strings.h>  
%}
```



```

ws          [ \t\n]+
commentstring \#[^\n]*\n
qstring      \"[^\n]*[\"\\n]
string       [^$&!@[><}{](,.:\\|\"\\t\\n ]+
integer      [0-9]+

%%

{commentstring} ;

{qstring} { yylval.string = strdup(yytext+1);
            yyval.string[yytext-2] = '\\0';
            return QSTRING; }

spacetype { return SPACETYPE; }
template { return TEMPLATE; }
subspace { return SUBSPACE; }
var       { return VAR; }
function { return FUNCTION; }
if        { return IF; }
is        { return IS; }
width     { return WIDTH; }
else      { return ELSE; }
first     { return FIRST; }
max       { return MAX; }
so        { return SO; }
at        { return AT; }
bigend    { return BIGEND; }
littleend { return LITTLEEND; }
minor     { return MINOR; }
this      { return THIS; }
with      { return WITH; }
use       { return USE; }
public    { return PUBLIC; }
sc        { return STARTCOMMENT; }
ec        { return ENDCOMMENT; }
implement { return IMPLEMENT; }

[&$@\\][><}{](,.:|\"\\t\\n ] { return yytext[0]; }

{integer} { yylval.integer = atoi(yytext); return INTEGER; }
{string}  { yylval.string = strdup(yytext); return STRING; }

{ws} ;

-----
Source file: hadley.y
-----

%{

#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include "hash.h"

#include "hadley-semantic.i"

%}

%union {
    char *string;
    int integer;
}

/* Punctuation */

%token '{' '}' '(' ')' '[' ']' '<' '>' ',' ':' '.' '$' '!' '@' '&'

```

```

/* Literals */

%token <string> STRING QSTRING
%token <integer> INTEGER

/* types */

%type <string> spacetypeName functionparameterName variabletypeName
%type <string> variableName literalString functionName functiondeclName
%type <string> functionspaceName functionremoteName functionwithName username
%type <string> functionlocalName subspaceName subspacetypeName functionwithType
%type <string> templateName tfunctiondeclName tfunctionparameterName implementName

%type <integer> literalInteger

/* Reserved words */

%token SPACETYPE SUBSPACE VAR FUNCTION IF IS ELSE FIRST MAX SO AT WIDTH
%token BIGEND LITTLEEND MINOR THIS WITH USE PUBLIC
%token STARTCOMMENT ENDCOMMENT

%token TEMPLATE IMPLEMENT

%%

start:    module;
module:   rootdeclarations;

rootdeclarations:    rootdeclaration rootdeclarations
                    | rootdeclaration;

rootdeclaration:     spacetypeDeclaration
                    | templateDeclaration;

spacetypeDeclaration: SPACETYPE spacetypeName
                    { sdecl_start($2); }
                    '{' spacetypebody '}'
                    { sdecl_finish(); };

spacetypeName:       STRING;
spacetypebody:       usesp componentdeclarations;

usesp:               /* empty */
                    | uses;

uses:                use
                    | use uses;

use:                 USE username '.'
                    { usedecl($2) };

username:            STRING;

componentdeclarations: componentdeclaration componentdeclarations
                    | componentdeclaration;

componentdeclaration: functiondeclaration
                    | variableDeclaration
                    | subspaceDeclaration
                    | implementDeclaration;

/* Here we deal with functions. */

functiondeclaration: FUNCTION functiondeclName functiontypepart
                    { fdecl_header_start($2); }
                    '(' functionparametersp ')'
                    { fdecl_header_finish(); }
                    functionwithpartp
                    '{' functionbody '}'
                    { fdecl_finish(); };

functiondeclName:    STRING;

```

```

functiontypepart:      IS STRING
                        { fdecl_get_function_type($2); };
functionparametersp:  /* empty */
                        | { emit_crlf_both();
                           fdecl_between_parameters(); }
functionparameters:   functionparameters;
                        functionparameter ','
                        { fdecl_between_parameters(); }
functionparameter:    functionparameters;
                        | functionparameter
functionparametername: functionparametername
                        functionparametertype
                        { fdecl_output_parameter(); };
functionparametername: STRING
                        { fdecl_get_parameter_name($1); };
functionparametertype: IS variablename
                        { fdecl_get_parameter_type($2); };
functionwithpartp:    /* empty */
                        | { fdecl_with_init(); }
functionwiths:        WITH functionwiths;
functionwith:         functionwith ',' functionwiths
                        | functionwith;
functionwith:         functionwithname IS functionwithtype
                        { fdecl_with_start($1, $3); }
                        AT expression
                        { fdecl_with_finish(); };
functionwithname:     STRING;
functionwithtype:     STRING;
functionbody:         { fdecl_body_start(); }
                        expression
                        { fdecl_body_finish(); };

/* Here we deal with variables. */

variabledeclaration:  VAR variablename
                        { vdecl_start($2); }
                        variabletypepart
                        variableendianpart
                        variablewidthpart
                        variablelocationpart '.'
                        { vdecl_finish(); };
variablename:        STRING;
variablepublicpart:  /* empty */
                        | PUBLIC
                        { vdecl_makepublic(); };
variabletypepart:    IS variablepublicpart variablename
                        { vdecl_basetype($3); }
                        variablearraypart { };
variabletypename:    STRING;
variablearraypart:   /* empty */
                        { vdecl_nullarray(); }
                        | '['
                        { vdecl_array_start(); }
                        expression '['
                        { vdecl_array_finish(); };
variablewidthpart:   /* empty */
                        { vdecl_nullwidth(); }
                        | WIDTH
                        { vdecl_width_start(); }
                        expression
                        { vdecl_width_finish(); };
variablelocationpart: AT
                        { vdecl_loc_start(); }
                        expression
                        { vdecl_loc_finish(); }
                        variableminorpart;
variableminorpart:   /* empty */
                        { vdecl_nullminor(); }
                        | MINOR

```

```

        { vdecl_minor_start(); }
expression
        { vdecl_minor_finish(); };
variableendianpart: /* Empty */
        { vdecl_noend(); }
| BIGEND
        { vdecl_bigend(); }
| LITTLEEND
        { vdecl_littleend(); };

subspacedeclaration: SUBSPACE subspacename IS subspacetypename
        { ssdecl_start($2, $4); }
        AT expression '.'
        { ssdecl_finish() };
subspacename: STRING;
subspacetypename: STRING;

implementdeclaration: IMPLEMENT implementname
        { implem_start($2); }
        '{' implementfunctions '}'
        { implem_finish($2); };
implementname: STRING;
implementfunctions: implementfunction
| implementfunction ',' implementfunctions;
implementfunction: STRING
        { implem_add($1); };

/* Here we deal with expressions. */

expression: literal
| qualifiedreference
| conditional
| fparameter;

qualifiedreference: '$' variablereference
| '@' functioncall
| '&' '!' variablemajorlocation
| '&' '.' variableminorlocation;

literal: literalstring literalinteger
        { expr_literal_memory($1, $2); }
| literalstring
        { expr_literal_string($1); };
| literalinteger
        { expr_literal_integer($1); };
literalstring: QSTRING;
literalinteger: INTEGER;

variablereference: variablename
        { expr_vref_start($1); }
variablerefarraypart
        { expr_vref_finish(); };

variablemajorlocation: variablename
        { expr_vmaj_start($1); }
variablelocarraypart
        { expr_vmaj_finish(); };

variableminorlocation: variablename
        { expr_vmin_start($1); }
variablelocarraypart
        { expr_vmin_finish(); };

variablelocarraypart: /* empty */
        { expr_vref_noarray(); }
| '['
        { expr_vref_onearray(); }
expression ']' ;

```

```

variablerefarraypart:    /* empty */
                        { expr_vref_noarray(); }
                        | '['
                        { expr_vref_onearray(); }
                        expression ']'
                        | '<'
                        { expr_vref_multarray_start(); }
                        expression ','
                        { expr_vref_multarray_middle(); }
                        expression '>';

functioncall:            functionname '(' functionargumentspart ')'
                        { expr_fcall_finish(); };
functionname:            THIS '.' functionlocalname
                        { expr_fname_mine($3); }
                        | functionspacename '.' functionremotename
                        { expr_fname_qualified($1, $3); };
functionspacename:      STRING;
functionremotename:     STRING;
functionlocalname:      STRING;
functionargumentspart:  /* empty */
                        { expr_farg_between(); }
                        functionarguments;
functionarguments:       functionargument ','
                        { expr_farg_between(); }
                        functionarguments
                        | functionargument;
functionargument:        expression;

fparameter:             STRING
                        { expr_fparm($1); };

/* Left open for more conditionals */
conditional:             ifconditional;

ifconditional:           ifheader '{'
                        { expr_cond_if_start(); }
                        expression '}'
                        { expr_cond_if_middle(); }
                        elsepart
                        { expr_cond_if_finish(); };
ifheader:               IF '('
                        { expr_cond_if_header_start(); }
                        expression ')'
                        { expr_cond_if_header_finish(); };
elsepart:               /* empty */
                        { expr_cond_if_noelse(); }
                        | ELSE '{' expression '}'

/* Okay, done with spacetypes. Now for templates. */

templatedeclaration:     TEMPLATE templatename
                        { tdecl_start($2); }
                        '{' templatebody '}'
                        { tdecl_finish(); };
templatename:           STRING;
templatebody:           tfunctiondeclarations;

tfunctiondeclarations:   tfunctiondeclaration tfunctiondeclarations
                        | tfunctiondeclaration;

tfunctiondeclaration:    FUNCTION tfunctiondeclname tfunctiontypepart
                        { tfdecl_start($2); }
                        '(' tfunctionparametersp ')'
                        { tfdecl_finish(); };
tfunctiondeclname:      STRING;
tfunctiontypepart:      IS STRING
                        { tfdecl_get_function_type($2); };

```

```

tfunctionparametersp:    /* empty */
                        | { tfdecl_between_parameters(); }
                        tfunctionparameters;
tfunctionparameters:    tfunctionparameter ','
                        { tfdecl_between_parameters(); }
                        tfunctionparameters;
                        | tfunctionparameter
tfunctionparameter:    tfunctionparametername
                        tfunctionparametertype
                        { tfdecl_output_parameter(); };
tfunctionparametername: STRING
                        { tfdecl_get_parameter_name($1); };
tfunctionparametertype: IS variabletypename
                        { tfdecl_get_parameter_type($2); };

%%

-----
Source file: hadley-semantics.i
-----

char *stype = NULL;

char *vname = NULL;
char *vtype = NULL;
char *vwidth = NULL;
char *vloc = NULL;
char *vend = NULL;

char *fname = NULL;
char *ftype = NULL;
char *fptype = NULL;
char *fpname = NULL;

char *ssname = NULL;

FILE *coutfile = NULL;
FILE *houtfile = NULL;

hash withtypes;
hash subtypes;

int vpublic = 0;
int varray = 0;

void emit_crlf(void) {
    fprintf(coutfile, "\n");
}

void emit_crlf_both(void) {
    fprintf(coutfile, "\n");
    fprintf(houtfile, "\n");
}

/* This function creates a data grabber function.  vname, vtype, varray, vwidth
   and vloc are already full. */

void usedecl(char *s) {
    fprintf(coutfile, "#include \"%s.h\"\n", s);
}

void sdecl_start(char *s) {
    char coutfilename[256];
    char houtfilename[256];

    strcpy(coutfilename, s);
    strcat(coutfilename, ".c");
    strcpy(houtfilename, s);
    strcat(houtfilename, ".h");

```

```

    coutfile = fopen(coutfilename, "w");
    houtfile = fopen(houtfilename, "w");
    fprintf(coutfile, "#include \"hadley.h\"\n\n");
    fprintf(coutfile, "/* Hadley space %s */\n\n", s);
    fprintf(houtfile, "/* Hadley space %s prototypes */\n\n", s);
    stype = strdup(s);

    /* THIS CAN POTENTIALLY CREATE A MEMORY LEAK */
    hash_init(&subtypes);
    hash_init(&withtypes);
}

void sdecl_finish(void) {
    fprintf(coutfile, "/* End of Hadley space %s */\n\n", stype);
    fclose(coutfile);
    fclose(houtfile);
    free(stype);
}

void fdecl_header_start(char *s) {
    fprintf(coutfile, "%shsl_%s_function_%s(long long int coreContext",
            ftype, stype, s);
    fprintf(houtfile, "%shsl_%s_function_%s(long long int coreContext",
            ftype, stype, s);
    fname = strdup(s);
}

void fdecl_get_function_type(char *s) {
    ftype = strdup( !strcmp(s, "integer") ? "long long int " :
                   !strcmp(s, "char") ? "struct clip " :
                   !strcmp(s, "list") ? "struct cliplist * " : "");
    if(!(*ftype)) {
        fprintf(stderr, "ERROR: Invalid data type %s\n", s);
    }
}

void fdecl_between_parameters(void) {
    fprintf(coutfile, ", \n");
    fprintf(houtfile, ", \n");
}

void fdecl_output_parameter(void) {
    fprintf(coutfile, " %s%s", ftype, fname);
    fprintf(houtfile, " %s%s", ftype, fname);
    free(fname);
    free(ftype);
}

void fdecl_get_parameter_type(char *s) {
    ftype = strdup( !strcmp(s, "integer")
                   ? "struct clip "
                   : "long long int ");
}

void fdecl_get_parameter_name(char *s) {
    fname = strdup(s);
}

void fdecl_with_init(void) {
    hash_init(&withtypes);
}

/* THIS CURRENTLY CREATES A SMALL MEMORY LEAK */

void fdecl_with_start(char *name, char *type) {
    hash_insert(&withtypes, name, strdup(type));

    fprintf(coutfile, " long long int %s_loc = (", name);

```

```

}

void fdecl_with_finish(void) {
    fprintf(coutfile, ");\n");
}

void fdecl_header_finish(void) {
    fprintf(coutfile, ")\n{\n");
    fprintf(houtfile, ");\n");
}

void fdecl_body_start(void) {
    fprintf(coutfile, "    %s coreResult;\n", ftype);
    fprintf(coutfile, "    \n");
    fprintf(coutfile, "    if(coreContext) hsl_CORE_pushContext(coreContext);\n");
    fprintf(coutfile, "    coreResult = (");
}

void fdecl_body_finish(void) {
    fprintf(coutfile, ");\n");
    fprintf(coutfile, "    if(coreContext) hsl_CORE_popContext(coreContext);\n");
    fprintf(coutfile, "    return coreResult;\n");
    fprintf(coutfile, "}\n{\n");
}

void fdecl_finish(void) {
    free(fname);
    free(ftype);
}

void vdecl_start(char *s) {
    fprintf(coutfile, "/* Variable %s */\n{\n", s);
    vname = strdup(s);
}

void vdecl_makepublic(void) {
    vpublic = 1;
}

void vdecl_basetype(char *s) {
    vtype = strdup(s);
    fprintf(coutfile, "#define hsl_%s_var_%s_type \"%s\"\n{\n",
            stype, vname, vtype);
}

void vdecl_nullarray(void) {
    fprintf(coutfile,
            "long long int hsl_%s_get_%s_array()\n{\n    return (1);\n}\n{\n",
            stype, vname);
    varray = 0;
}

void vdecl_array_start(void) {
    fprintf(coutfile, "long long int hsl_%s_get_%s_array()\n{\n    return (",
            stype, vname);
    varray = 1;
}

void vdecl_array_finish(void) {
    fprintf(coutfile, ");\n}\n{\n");
}

void vdecl_nullwidth(void) {
    fprintf(coutfile,
            "long long int hsl_%s_get_%s_width()\n{\n    return (8);\n}\n{\n",
            stype, vname);
}

void vdecl_width_start(void) {

```



```

        fprintf(coutfile, "long long int hsl_%s_get_%s_width()\n{\n  return (",
                stype, vname);
    }

void vdecl_width_finish(void) {
    fprintf(coutfile, "};\n}\n\n");
}

void vdecl_loc_start(void) {
    fprintf(coutfile, "long long int hsl_%s_get_%s_location()\n{\n  return (",
                stype, vname);
}

void vdecl_loc_finish(void) {
    fprintf(coutfile, "};\n}\n\n");
}

void vdecl_nullminor(void) {
    fprintf(coutfile,
            "long long int hsl_%s_get_%s_minorloc()\n{\n  return (0);\n}\n\n",
            stype, vname);
}

void vdecl_minor_start(void) {
    fprintf(coutfile, "long long int hsl_%s_get_%s_minorloc()\n{\n  return (",
            stype, vname);
}

void vdecl_minor_finish(void) {
    fprintf(coutfile, "};\n}\n\n");
}

void vdecl_noend(void) {
    vend = strdup("none");
}

void vdecl_bigend(void) {
    vend = strdup("big");
}

void vdecl_littleend(void) {
    vend = strdup("little");
}

void vdecl_finish(void) {
    fprintf(coutfile, "#define hsl_%s_var_%s_end %d\n\n",
            stype, vname, !strcmp(vend, "big") ? 1 : 0);

    if(!strcmp(vtype, "integer")) {
        fprintf(coutfile, "long long int hsl_%s_get_%s_modified_loc_single(long long int
start)\n", stype, vname);
        fprintf(coutfile, "{\n");
        fprintf(coutfile, "    long long int l = hsl_%s_get_%s_location();\n", stype, vname);
        fprintf(coutfile, "    long long int m = hsl_%s_get_%s_minorloc();\n", stype, vname);
        fprintf(coutfile, "    long long int w = hsl_%s_get_%s_width();\n", stype, vname);
        fprintf(coutfile, "    \n");
        fprintf(coutfile, "    m += w * start;\n");
        fprintf(coutfile, "    l += m / 8;\n");
        fprintf(coutfile, "    return l;\n");
        fprintf(coutfile, "};\n");
        fprintf(coutfile, "\n");
        fprintf(coutfile, "long long int hsl_%s_get_%s_modified_minorloc_single(long long int
start)\n", stype, vname);
        fprintf(coutfile, "{\n");
        fprintf(coutfile, "    long long int l = hsl_%s_get_%s_location();\n", stype, vname);
        fprintf(coutfile, "    long long int m = hsl_%s_get_%s_minorloc();\n", stype, vname);
        fprintf(coutfile, "    long long int w = hsl_%s_get_%s_width();\n", stype, vname);
        fprintf(coutfile, "    \n");
        fprintf(coutfile, "    m += w * start;\n");
    }
}

```

```

        fprintf(coutfile, "    m %= 8;\n");
        fprintf(coutfile, "    return m;\n");
        fprintf(coutfile, "}\n");
        fprintf(coutfile, "\n");
        fprintf(coutfile, "long long int hsl_%s_get_%s_data_single(long long int start)\n",
stype, vname);
        fprintf(coutfile, "{\n");
        fprintf(coutfile, "    long long int l = hsl_%s_get_%s_location();\n", stype, vname);
        fprintf(coutfile, "    long long int m = hsl_%s_get_%s_minorloc();\n", stype, vname);
        fprintf(coutfile, "    long long int w = hsl_%s_get_%s_width();\n", stype, vname);
        fprintf(coutfile, "    int e = hsl_%s_var_%s_end;\n", stype, vname);
        fprintf(coutfile, "    long long int result;\n");
        fprintf(coutfile, "    \n");
        fprintf(coutfile, "    m += w * start;\n");
        fprintf(coutfile, "    l += m / 8;\n");
        fprintf(coutfile, "    m %= 8;\n");
        fprintf(coutfile, "    result = hsl_CORE_get_single_integer(l, m, w, e);\n");
        fprintf(coutfile, "    return result;\n");
        fprintf(coutfile, "}\n");
        fprintf(coutfile, "\n");
        fprintf(coutfile, "struct clip hsl_%s_get_%s_data_multiple(long long int start, long long
int n)\n", stype, vname);
        fprintf(coutfile, "{\n");
        fprintf(coutfile, "    long long int l = hsl_%s_get_%s_location();\n", stype, vname);
        fprintf(coutfile, "    long long int m = hsl_%s_get_%s_minorloc();\n", stype, vname);
        fprintf(coutfile, "    long long int w = hsl_%s_get_%s_width();\n", stype, vname);
        fprintf(coutfile, "    long long int major_n;\n");
        fprintf(coutfile, "    struct clip result;\n");
        fprintf(coutfile, "    \n");
        fprintf(coutfile, "    m += w * start;\n");
        fprintf(coutfile, "    l += m / 8;\n");
        fprintf(coutfile, "    m %= 8;\n");
        fprintf(coutfile, "    n *= w;\n");
        fprintf(coutfile, "    major_n = n / 8;\n");
        fprintf(coutfile, "    n %= 8;\n");
        fprintf(coutfile, "    result = hsl_CORE_get_complex_data(l, m, w, major_n, n);\n");
        fprintf(coutfile, "    return result;\n");
        fprintf(coutfile, "    }\n");
        fprintf(coutfile, "\n");
    } else {
        fprintf(coutfile, "long long int hsl_%s_get_%s_modified_loc_single(long long int
start)\n", stype, vname);
        fprintf(coutfile, "{\n");
        fprintf(coutfile, "    long long int l = hsl_%s_get_%s_location();\n", stype, vname);
        fprintf(coutfile, "    \n");
        fprintf(coutfile, "    return l + start;\n");
        fprintf(coutfile, "}\n");
        fprintf(coutfile, "\n");
        fprintf(coutfile, "long long int hsl_%s_get_%s_modified_minorloc_single(long long int
start)\n", stype, vname);
        fprintf(coutfile, "{\n");
        fprintf(coutfile, "    return 0;\n");
        fprintf(coutfile, "}\n");
        fprintf(coutfile, "\n");
        fprintf(coutfile, "struct clip hsl_%s_get_%s_data_single(long long int start)\n", stype,
vname);
        fprintf(coutfile, "{\n");
        fprintf(coutfile, "    long long int l = hsl_%s_get_%s_location();\n", stype, vname);
        fprintf(coutfile, "    struct clip result;\n");
        fprintf(coutfile, "    \n");
        fprintf(coutfile, "    result = hsl_CORE_get_char_data(l + start, 1);\n");
        fprintf(coutfile, "    return result;\n");
        fprintf(coutfile, "    }\n");
        fprintf(coutfile, "\n");
        fprintf(coutfile, "struct clip hsl_%s_get_%s_data_multiple(long long int start, long long
int n)\n", stype, vname);
        fprintf(coutfile, "{\n");
        fprintf(coutfile, "    long long int l = hsl_%s_get_%s_location();\n", stype, vname);
        fprintf(coutfile, "    struct clip result;\n");

```

```

        fprintf(coutfile, "  \n");
        fprintf(coutfile, "  result = hsl_CORE_get_char_data(l + start, n);\n");
        fprintf(coutfile, "  return result;\n");
        fprintf(coutfile, "}\n");
        fprintf(coutfile, "\n");
    }

    if(vpublic) {
        if(!strcmp(vtype, "integer")) {
            if(varray) {
                fprintf(houtfile, "long long int hsl_%s_function_%s(long long int coreContext,
long long int start);\n", stype, vname);
                fprintf(coutfile, "long long int hsl_%s_function_%s(long long int coreContext,
long long int start) {\n", stype, vname);
                fprintf(coutfile, "  long long int coreResult;\n\n");
                fprintf(coutfile, "  if(coreContext) hsl_CORE_pushContext(coreContext);\n");
                fprintf(coutfile, "  coreResult = hsl_%s_get_%s_data_single(start);\n", stype,
vname);
                fprintf(coutfile, "  if(coreContext) hsl_CORE_popContext(coreContext);\n");
                fprintf(coutfile, "  return coreResult;\n");
                fprintf(coutfile, "}\n\n");
            } else {
                fprintf(houtfile, "long long int hsl_%s_function_%s(long long int
coreContext);\n", stype, vname);
                fprintf(coutfile, "long long int hsl_%s_function_%s(long long int coreContext)
{\n", stype, vname);
                fprintf(coutfile, "  long long int coreResult;\n\n");
                fprintf(coutfile, "  if(coreContext) hsl_CORE_pushContext(coreContext);\n");
                fprintf(coutfile, "  coreResult = hsl_%s_get_%s_data_single(0);\n", stype, vname);
                fprintf(coutfile, "  if(coreContext) hsl_CORE_popContext(coreContext);\n");
                fprintf(coutfile, "  return coreResult;\n");
                fprintf(coutfile, "}\n\n");
            }
        } else {
            fprintf(stderr, "ERROR: string variable %s cannot be public\n", vname);
        }
    }
}

varray = 0;
vpublic = 0;
free(vname);
free(vtype);
free(vwidth);
free(vloc);
free(vend);
}

void ssdecl_start(char *name, char *type) {
    hash_insert(&subtypes, name, strdup(type));

    fprintf(coutfile, "long long int hsl_%s_get_%s_location()\n{\n  return(",
        stype, name);
}

void ssdecl_finish(void) {
    fprintf(coutfile, ");\n}\n");
}

void implem_start(char *s) {
    fprintf(coutfile, "#include \"%s.h\"\n\n", s);
    fprintf(houtfile, "void hsl_%s_register_%s(void);\n", stype, s);
    fprintf(coutfile, "void hsl_%s_register_%s() {\n", stype, s);
    fprintf(coutfile, "  hsl_%s_template t = {\n", s);
}

void implem_add(char *s) {
    fprintf(coutfile, "    hsl_%s_function_%s,\n", stype, s);
}

```

```

void implem_finish(char *s) {
    fprintf(coutfile, " }\n");
    fprintf(coutfile, " hsl_%s_template *p;\n\n", s);
    fprintf(coutfile, " p = malloc(sizeof(hsl_%s_template));\n", s);
    fprintf(coutfile, " memcpy(p, &t, sizeof(hsl_%s_template));\n", s);
    fprintf(coutfile, " register_hsl_%s_implementation(\"%s\", p);\n", s, stype);
    fprintf(coutfile, "}\n\n");
}

void expr_literal_string(char *s) {
    fprintf(coutfile, "(hsl_CORE_stringtoclip(\"%s\"))", s);
}

void expr_literal_memory(char *s, int i) {
    fprintf(coutfile, "(hsl_CORE_mementoclip(\"%s\", %d))", s, i);
}

void expr_literal_integer(int i) {
    fprintf(coutfile, "(%d)", i);
}

void expr_vref_start(char *s) {
    fprintf(coutfile, "hsl_%s_get_%s_data_", stype, s);
}

void expr_vref_finish(void) {
    fprintf(coutfile, "");
}

void expr_vref_noarray(void) {
    fprintf(coutfile, "single(0)");
}

void expr_vref_onearray(void) {
    fprintf(coutfile, "single(");
}

void expr_vref_multarray_start(void) {
    fprintf(coutfile, "multiple(");
}

void expr_vref_multarray_middle(void) {
    fprintf(coutfile, ", ");
}

void expr_vmaj_start(char *s) {
    fprintf(coutfile, "hsl_%s_get_%s_modified_loc_", stype, s);
}

void expr_vmaj_finish(void) {
    fprintf(coutfile, "");
}

void expr_vmin_start(char *s) {
    fprintf(coutfile, "hsl_%s_get_%s_modified_minorloc_", stype, s);
}

void expr_vmin_finish(void) {
    fprintf(coutfile, "");
}

void expr_fcall_finish(void) {
    fprintf(coutfile, "");
}

void expr_fname_builtin(char *s) {
    fprintf(coutfile, "hsl_builtin_%s(", s);
}

```

```

void expr_fname_mine(char *s) {
    fprintf(coutfile, "hsl_%s_function_%s(0", stype, s);
}

void expr_fname_qualified(char *t, char *s) {
    char *u;

    /* "sys" has magic foo. */

    if(strcmp(t, "sys")) {
        u = (char *) hash_lookup(&withtypes, t);
        if(u) {
            fprintf(coutfile, "hsl_%s_function_%s(%s_loc", u, s, t);
            return;
        }
        u = (char *) hash_lookup(&subtypes, t);
        if(u) {
            fprintf(coutfile, "hsl_%s_function_%s(hsl_%s_get_%s_location()",
                u, s, stype, t);
            return;
        }
        fprintf(stderr, "ERROR: %s is not a known location\n", t);
    } else {
        fprintf(coutfile, "hsl_%s_function_%s(0", t, s);
    }
}

void expr_farg_between(void) {
    fprintf(coutfile, ", ");
}

void expr_fparm(char *s) {
    fprintf(coutfile, "%s", s);
}

void expr_cond_if_start(void) {
    fprintf(coutfile, "\n( ");
}

void expr_cond_if_middle(void) {
    fprintf(coutfile, "\n) : (\n");
}

void expr_cond_if_finish(void) {
    fprintf(coutfile, "\n)");
}

void expr_cond_if_header_start(void) {
    fprintf(coutfile, "(( \n ");
}

void expr_cond_if_header_finish(void) {
    fprintf(coutfile, " \n) ?");
}

void expr_cond_if_noelse(void) {
    fprintf(coutfile, "0");
}

void expr_cond_or(void) {
    fprintf(coutfile, " || ");
}

void expr_cond_and(void) {
    fprintf(coutfile, " && ");
}

void tdecl_start(char *s) {
    char coutfilename[256];

```

```

char houtfilename[256];

strcpy(coutfilename, s);
strcat(coutfilename, ".c");
strcpy(houtfilename, s);
strcat(houtfilename, ".h");

coutfile = fopen(coutfilename, "w");
houtfile = fopen(houtfilename, "w");
fprintf(coutfile, "/* Hadley template %s code */\n\n", s);
fprintf(houtfile, "/* Hadley template %s typedefs */\n\n", s);
fprintf(houtfile, "typedef struct {\n ");

stype = strdup(s);

/* THIS CAN POTENTIALLY CREATE A MEMORY LEAK */
hash_init(&subtypes);
hash_init(&withtypes);
}

void tdecl_finish(void) {
    fprintf(houtfile, "} hsl_%s_template;\n\n", stype);
    fprintf(coutfile, "#include \"hadley.h\"\n");
    fprintf(coutfile, "#include \"%s.h\"\n", stype);
    fprintf(coutfile, "#include \"hash.h\"\n\n");
    fprintf(coutfile, "hash hsl_%s_template_instances;\n", stype);
    fprintf(coutfile, "int hsl_%s_template_initialized = 0;\n", stype);
    fprintf(coutfile, "void register_hsl_%s_implementation(char *name, hsl_%s_template\n*implementation) {\n", stype, stype);
    fprintf(coutfile, "    if(!hsl_%s_template_initialized) {\n", stype);
    fprintf(coutfile, "        hash_init(&hsl_%s_template_instances);\n", stype);
    fprintf(coutfile, "        hsl_%s_template_initialized = 1;\n", stype);
    fprintf(coutfile, "    }\n\n");
    fprintf(coutfile, "    hash_insert(&hsl_%s_template_instances, name, implementation);\n",
stype);
    fprintf(coutfile, "}\n\n");
    fprintf(coutfile, "hsl_%s_template *get_hsl_%s_implementation(char *name) {\n", stype,
stype);
    fprintf(coutfile, "    return (hsl_%s_template *) (hash_lookup(&hsl_%s_template_instances,\nname));\n", stype, stype);
    fprintf(coutfile, "}\n\n");

    fprintf(houtfile, "void register_hsl_%s_implementation(char *name, hsl_%s_template\n*implementation);\n", stype, stype);
    fprintf(houtfile, "hsl_%s_template *get_hsl_%s_implementation(char *name);\n", stype, stype);

    fprintf(coutfile, "/* End of Hadley template %s code */\n\n", stype);
    fclose(coutfile);
    fclose(houtfile);
    free(stype);
}

void tfdecl_start(char *s) {
    fprintf(houtfile, " %s(%s) (long long int coreContext", ftype, s);
    fname = strdup(s);
}

void tfdecl_get_function_type(char *s) {
    ftype = strdup( !strcmp(s, "integer") ? "long long int " :
                    !strcmp(s, "char") ? "struct clip " :
                    !strcmp(s, "list") ? "struct cliplist * " : "");
    if(!(*ftype)) {
        fprintf(stderr, "ERROR: Invalid data type %s\n", s);
    }
}

void tfdecl_between_parameters(void) {
    fprintf(houtfile, ", \n");
}

```

```

void tfdecl_output_parameter(void) {
    fprintf(houtfile, "    %s%s", fptype, fpname);
    free(fpname);
    free(fptype);
}

void tfdecl_get_parameter_type(char *s) {
    fptype = strdup(    strcmp(s, "integer")
        ? "struct clip "
        : "long long int ");
}

void tfdecl_get_parameter_name(char *s) {
    fpname = strdup(s);
}

void tfdecl_finish(void) {
    fprintf(houtfile, ");\n");
}

-----
Source file: hadley.h
-----

#ifdef __KERNEL__
#include <linux/slab.h>
#include <linux/mm.h>
#define malloc(a) kmalloc(a, SLAB_KERNEL)
#define free(a) kfree(a)
#else
#include <stdlib.h>
#endif

/* Built-in functions. */

#define LITTLEEND 0
#define BIGEND 0

#define hsl_sys_function_add(x, a, b)    (a + b)
#define hsl_sys_function_sub(x, a, b)    (a - b)
#define hsl_sys_function_mul(x, a, b)    (a * b)
#define hsl_sys_function_div(x, a, b)    (a / b)
#define hsl_sys_function_mod(x, a, b)    (a % b)
#define hsl_sys_function_equ(x, a, b)    (a == b)
#define hsl_sys_function_grt(x, a, b)    (a > b)
#define hsl_sys_function_lst(x, a, b)    (a < b)
#define hsl_sys_function_gre(x, a, b)    (a >= b)
#define hsl_sys_function_lse(x, a, b)    (a <= b)
#define hsl_sys_function_and(x, a, b)    (a && b)
#define hsl_sys_function_or(x, a, b)     (a || b)
#define hsl_sys_function_not(x, a)       (!a)
#define hsl_sys_function_intAt(x, l, m, w, e) \
    hsl_CORE_get_single_integer(l, m, w, e)
#define hsl_sys_function_charAt(x, l, n) \
    hsl_CORE_get_char_data(l, n)
#define hsl_sys_function_len(x, a)       (a.length)

long long int hsl_sys_function_2to(long long int x, long long int a);

/* Clip structure. */

struct clip {
    char *location;
    long long int value;
    long long int length;
    long long int outputloc;
    long long int outputminorloc;
    long long int endian;

```

```

    int isAllocated;
};

struct cliplist {
    void *c;
    struct cliplist *next;
};

struct clip hsl_CORE_stringtoclip(char *s);
struct clip hsl_CORE_memtoclip(void *s, long long int l);
void hsl_CORE_add_garbage(void *);
void hsl_CORE_garbagecollect(void);

/* Core function prototypes. */

char *hcl_get_context_current();

struct cliplist *hsl_sys_function_listrepeat(long long int context,
                                             struct cliplist *k,
                                             long long int times,
                                             long long int space);

struct cliplist *hsl_sys_function_listfrom(long long int context,
                                             struct cliplist *k,
                                             long long int l);

struct cliplist *hsl_sys_function_newlist(long long int context,
                                           struct clip s);

struct cliplist *hsl_sys_function_listadd(long long int context,
                                           struct cliplist *k,
                                           struct clip s);

// This version of listadd requires that s appear at the end of k. See
// the documentation for write_clips for why this is important.

struct cliplist *hsl_sys_function_listaddend(long long int context,
                                              struct cliplist *k,
                                              struct clip s);

struct cliplist *hsl_sys_function_listcon(long long int context,
                                           struct cliplist *k,
                                           struct cliplist *l);

struct clip hsl_sys_function_makecharhint(long long int context,
                                           struct clip c,
                                           long long int l);

struct clip hsl_sys_function_makecharhintfill(long long int context,
                                              struct clip c,
                                              long long int size,
                                              long long int l);

struct clip hsl_sys_function_makeinthint(long long int context,
                                          long long int value,
                                          long long int l,
                                          long long int m,
                                          long long int w,
                                          long long int e);

struct clip hsl_sys_function_clipcat(long long int context,
                                     struct clip s,
                                     struct clip t);

struct clip hsl_sys_function_subclip(long long int context,
                                     struct clip s,
                                     long long int start,
                                     long long int len);

```



```

long long int hsl_sys_function_clipclip(long long int context,
                                       struct clip s,
                                       struct clip t);

long long int hsl_sys_function_cliplen(long long int context,
                                       struct clip s);

long long int hsl_sys_function_clipecu(long long int context,
                                       struct clip s,
                                       struct clip t);

long long int hsl_CORE_get_single_integer(long long int location,
                                       long long int minor_loc,
                                       long long int width,
                                       int endian);

struct clip hsl_CORE_get_complex_data(long long int location,
                                       long long int minor_loc,
                                       long long int width,
                                       long long int bytes,
                                       long long int bits);

struct clip hsl_CORE_get_char_data(long long int location,
                                   long long int n);

// The Hadley implementation is *required* to write the clips in 1 such that
// clips appearing later in the list are written over earlier ones. This
// allows HSL developers to require ordering by using listaddend.

void hsl_CORE_write_clips(struct cliplist *l);
void hsl_CORE_write_single_integer(struct clip c);
void hsl_CORE_write_char(struct clip c);

void hsl_CORE_write_char_data(long long int l,
                              long long int n,
                              void *data);

void hsl_CORE_init(void);

char *hsl_CORE_cliptostring(struct clip c);

-----
Source file: hash.h
-----

#ifdef __KERNEL__
#include <linux/slab.h>
#include <linux/mm.h>
#define malloc(a) kmalloc(a, SLAB_KERNEL)
#define free(a) kfree(a)
#else
#include <stdlib.h>
#include <strings.h>
#endif

struct hash {
    char *key;
    void *data;
    struct hash *next;
};

typedef struct hash hash;

void hash_init(hash *);
void *hash_lookup(hash *, char *);
void hash_insert(hash *, char *, void *);
void hash_delete(hash *, char *);

-----

```

Source file: core.c

```
-----

#include "hadley.h"

#ifdef __KERNEL__
#include <linux/fs.h>
#include <linux/slab.h>
#include <linux/mm.h>
#include <linux/string.h>
#define malloc(a) kmalloc(a, SLAB_KERNEL)
#define free(a) kfree(a)
#else
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <string.h>
#endif

long long int contextMode = 0;
long long int thisContext = 0;

#ifdef __KERNEL__
void *contextLoc;
FILE *contextFile;
#endif

static int our_endian;
static struct cliplist *garbage = NULL;

static void swap_endian(unsigned long long int *u) {
    unsigned char *c, *d;

    c = (unsigned char *) u;
    d = c + sizeof(unsigned long long int) - 1;
    while(d > c) {
        char q;

        q = *d;
        *d = *c;
        *c = q;

        c++; d--;
    }
}

long long int hsl_sys_function_2to(long long int context,
                                   long long int to) {
    long long int k = 1;

    while(to) {
        to--;
        k *= 2;
    }

    return k;
}

long long int hsl_sys_function_clipequ(long long int context,
                                       struct clip s,
                                       struct clip t) {
    return ((s.length == t.length) &&
            !memcmp(s.location, t.location, (int) s.length));
}

struct cliplist *hsl_sys_function_listfrom(long long int context,
                                           struct cliplist *k,
                                           long long int l) {
    struct cliplist *ok = k;

```

```

    while(k) {
        // k is a list of output clips. For each entry in it,
        // increment that entry's outputloc by 1.
        ((struct clip *) k->c)->outputloc += 1;
        k = k->next;
    }
    return ok;
}

struct cliplist *hsl_sys_function_newlist(long long int context,
                                         struct clip s) {
    struct clip *t = malloc(sizeof(struct clip));
    struct cliplist *l = malloc(sizeof(struct cliplist));

    memcpy(t, &s, sizeof(struct clip));
    l->next = NULL;
    l->c = t;
    hsl_CORE_add_garbage(l);
    hsl_CORE_add_garbage(t);
    return l;
}

struct cliplist *hsl_sys_function_listadd(long long int context,
                                         struct cliplist *k,
                                         struct clip s) {
    struct clip *t = malloc(sizeof(struct clip));
    struct cliplist *l = malloc(sizeof(struct cliplist));

    memcpy(t, &s, sizeof(struct clip));
    l->next = k;
    l->c = t;
    hsl_CORE_add_garbage(l);
    hsl_CORE_add_garbage(t);
    return l;
}

struct cliplist *hsl_sys_function_listaddend(long long int context,
                                             struct cliplist *k,
                                             struct clip s) {
    struct clip *t = malloc(sizeof(struct clip));
    struct cliplist *l = malloc(sizeof(struct cliplist));
    struct cliplist *ok = k;

    memcpy(t, &s, sizeof(struct clip));
    l->next = NULL;
    l->c = t;
    while(k->next) k = k->next;
    k->next = l;
    hsl_CORE_add_garbage(l);
    hsl_CORE_add_garbage(t);
    return ok;
}

struct cliplist *hsl_sys_function_listcon(long long int context,
                                         struct cliplist *k,
                                         struct cliplist *l) {
    struct cliplist *ok = k;
    while(k->next) k = k->next;
    k->next = l;
    return ok;
}

struct cliplist *hsl_sys_function_listcopy(long long int context,
                                           struct cliplist *k) {
    struct cliplist *l = NULL;
    struct clip *c;

```

```

while(k) {
    c = malloc(sizeof(struct clip));
    memcpy(c, k->c, sizeof(struct clip));
    hsl_CORE_add_garbage(c);
    if(!l) {
        l = hsl_sys_function_newlist(context, *c);
    } else hsl_sys_function_listaddend(context, l, *c);
    k = k->next;
}
return l;
}

struct cliplist *hsl_sys_function_listrepeat(long long int context,
                                             struct cliplist *k,
                                             long long int times,
                                             long long int space) {

    int i, j;
    struct cliplist *l;

    j = 0;

    l = hsl_sys_function_listcopy(context, k);
    for(i = 1; i < times; i++) {
        j += space;
        l = hsl_sys_function_listcon(context, l,
                                     hsl_sys_function_listfrom(context,
                                     hsl_sys_function_listcopy(context, k),
                                     j)
                                     );
    }

    return l;
}

struct clip hsl_sys_function_makecharhint(long long int context,
                                           struct clip c,
                                           long long int l) {
    struct clip r = {c.location, 0, c.length, l, 0, 0, 0};
    return r;
}

struct clip hsl_sys_function_makecharhintfill(long long int context,
                                              struct clip c,
                                              long long int size,
                                              long long int l) {

    struct clip r;
    int i;

    r.location = malloc(size);
    r.length = size;
    r.outputloc = l;
    r.outputminorloc = 0;
    r.endian = 0;
    r.isAllocated = 1;

    for(i = 0; i < size / c.length; i++) {
        memcpy(r.location + c.length * i, c.location, c.length);
    }

    return r;
}

struct clip hsl_sys_function_makeinthint(long long int context,
                                         long long int value,
                                         long long int l,
                                         long long int m,
                                         long long int w,
                                         long long int e) {
    struct clip r = {NULL, value, w, l, m, e, 0};

```

```

    return r;
}

struct clip hsl_CORE_stringtoclip(char *s) {
    struct clip r;
    int l = strlen(s);

    r.location = malloc(l+1);
    strcpy(r.location, s);
    r.length = strlen(r.location);
    r.isAllocated = 1;
    r.value = r.outputloc = r.outputminorloc = r.endian = 0;
    hsl_CORE_add_garbage(r.location);

    return r;
}

struct clip hsl_CORE_memtoclip(void *s, long long int l) {
    struct clip r;

    r.location = malloc(l);
    memcpy(r.location, s, l);
    r.length = l;
    r.isAllocated = 1;
    r.value = r.outputloc = r.outputminorloc = r.endian = 0;
    hsl_CORE_add_garbage(r.location);

    return r;
}

struct clip hsl_sys_function_clipcat(long long int context,
                                     struct clip s,
                                     struct clip t) {
    struct clip r;

    r.length = s.length + t.length;
    r.location = malloc(r.length);
    r.value = r.outputloc = r.outputminorloc = r.endian = 0;

    memcpy(r.location, s.location, s.length);
    memcpy(r.location + s.length, t.location, t.length);

    hsl_CORE_add_garbage(r.location);
    r.isAllocated = 1;
    return r;
}

struct clip hsl_sys_function_subclip(long long int context,
                                     struct clip s,
                                     long long int start,
                                     long long int len) {
    struct clip r;

    r.length = len;
    r.location = malloc(r.length);
    r.value = r.outputloc = r.outputminorloc = r.endian = 0;

    memcpy(r.location, s.location + start, len);
    hsl_CORE_add_garbage(r.location);
    r.isAllocated = 1;
    return r;
}

void hsl_CORE_add_garbage(void *c) {
    struct cliplist *oldgarbage = garbage;

    garbage = malloc(sizeof(struct cliplist));
    garbage->c = c;
    garbage->next = oldgarbage;
}

```

```

}

void hsl_CORE_garbagecollect() {
    while(garbage) {
        struct cliplist *nextgarbage = garbage->next;
        free(garbage->c);
        free(garbage);
        garbage = nextgarbage;
    }
}

static unsigned char andmask[] =
/*      LIT      BIG */
{ 0xFF, 0xFF,
  0x01, 0x80,
  0x03, 0xC0,
  0x07, 0xE0,
  0x0F, 0xF0,
  0x1F, 0xF8,
  0x3F, 0xFC,
  0x7F, 0xFE};

/* These are the fundamental linking functions. */

#ifdef __KERNEL__

void hsl_CORE_set_context_memory_mode() {
    contextMode = 0;
}

void hsl_CORE_set_context_file_mode() {
    contextMode = 1;
}

void hsl_CORE_set_context_memory_location(void *p) {
    contextLoc = p;
}

void hsl_CORE_set_context_file(FILE *f) {
    contextFile = f;
}

#endif

#ifdef __KERNEL__
extern struct super_block *HSL_super;
#endif

struct clip hsl_CORE_get_char_data(long long int l, long long int n) {
    char *c;
    struct clip q;

#ifdef __KERNEL__
    long long int blocks, bs, bufpos, i, firstblock;
    struct buffer_head *bh;

    l += thisContext;
    bs = HSL_super->s_blocksize;
    if(bs == 0) bs = 512; // XXX: *HACK*
    //      printk(KERN_ALERT "Attempting to read %d bytes from %d at blocksize %d.\n",
    //              (int) n, (int) l, (int) bs);
    blocks = n / bs;
    if(n % bs) blocks++;
    if(((l) % bs) + (n % bs) > bs) blocks++;

    firstblock = l / bs;
    bufpos = 0;
    c = (char *) malloc(blocks * bs);

```

```

//      printk(KERN_ALERT "Attempting to read %d blocks, starting at the %dth.\n",
//      (int) blocks, (int) firstblock);
for(i = 0; i < blocks; i++) {
    //      printk(KERN_ALERT " .");
    bh = bread(HSL_super->s_dev, firstblock + i, bs);
    memcpy(c + bufpos, bh->b_data, bs);
    brelse(bh);
    bufpos += bs;
}
hsl_CORE_add_garbage(c);
c += (1 % bs);
//      printk(KERN_ALERT "\nDone.\n");

#else
if(contextMode == 0) {
    c = contextLoc + thisContext + 1;
}

if(contextMode == 1) {
    c = malloc(n);
    hsl_CORE_add_garbage(c);
    fseek(contextFile, thisContext + 1, SEEK_SET);
    fread(c, n, 1, contextFile);
}
#endif

if(n == 512) {
    printf("          Read sector %#d.\n", (int) (1 / 512));
}

q.location = c;
q.length = n;
q.value = q.outputloc = q.outputminorloc = q.endian = 0;
q.isAllocated = 0;
return q;
}

void hsl_CORE_write_char_data(long long int l, long long int n, void *data) {
#ifdef __KERNEL__
    if(contextMode == 0) {
        char *c = contextLoc;
        c += l;
        memcpy(c, data, n);
    }

    if(contextMode == 1) {
        fseek(contextFile, l, SEEK_SET);
        fwrite(data, n, 1, contextFile);
        fflush(contextFile);
    }
#endif
}

struct clip hsl_CORE_get_complex_data(long long int location,
                                     long long int minor_loc,
                                     long long int width,
                                     long long int bytes,
                                     long long int bits) {
    struct clip dummy = {NULL, 0, 0, 0, 0, 0, 0};

    return dummy;
}

char *hsl_CORE_cliptostring(struct clip c) {
    char *c2;

    c2 = malloc(c.length + 1);
    memcpy(c2, c.location, c.length);
    c2[c.length] = '\0';
}

```

```

    return c2;
}

long long int hsl_sys_function_clipclip(long long int context,
                                       struct clip s,
                                       struct clip t) {

    char *ss;
    char *st;
    char *res;
    long long int i;

    ss = hsl_CORE_cliptostr(s);
    st = hsl_CORE_cliptostr(t);
    res = strstr(ss, st);

    i = res ? (long long int) (res - ss) + 1 : 0;

    free(ss);
    free(st);

    return i;
}

long long int hsl_sys_function_clipplen(long long int context, struct clip s) {
    return s.length;
}

#ifdef __KERNEL__

static void outbuf(unsigned char *c, int n) {
    int i;

    for(i = 0; i < n; i++) {
        printf("%02x ", c[i]);
    }
    printf("\n");
}

#endif

long long int hsl_CORE_get_single_integer(long long int l,
                                          long long int m,
                                          long long int w,
                                          int e) {

    struct clip source;
    unsigned char *scratchbuf;
    unsigned long long int result;

    int max_bytewidth, bytewidth, uneven, mrev, unevenrev;

    {
        unsigned char mbuf[32];
        mbuf[0] = '\0';
        scratchbuf = mbuf + 1;

        uneven = w % 8;
        unevenrev = 8 - uneven;
        mrev = 8 - m;
        max_bytewidth = (w + m) / 8 + (((w + m) % 8) ? 1 : 0);
        bytewidth = w / 8 + ((w % 8) ? 1 : 0);

        source = hsl_CORE_get_char_data(l, max_bytewidth);
        memcpy(scratchbuf, source.location, max_bytewidth);
        scratchbuf[max_bytewidth] = '\0';

        /* Minor point exists */

        if(m) {

```



```

    unsigned char *c, *d;
    int i;

    c = scratchbuf;
    d = c + 1;

    if(e) { /* Big Endian */
        for(i = 0; i < max_bytewidth; i++) {
            *c = (*c << m) | (*d >> mrev);
            c++; d++;
        }
    } else { /* Little Endian */
        for(i = 0; i < max_bytewidth; i++) {
            *c = (*c >> m) | (*d << mrev);
            c++; d++;
        }
    }

    scratchbuf[bytewidth-1] &= andmask[uneven * 2 + e];

    if(uneven && e) {
        unsigned char *c, *d;
        int i;

        c = scratchbuf + bytewidth - 1;
        d = c - 1;

        for(i = bytewidth - 1; i >= 0; i--) {
            *c = (*c >> unevenrev) | (*d << uneven);
            c--; d--;
        }
    }

    /* Okay. We now have a valid bytewidth-byte integer in the first
       bytewidth bytes of scratchbuf. */

    result = 0;
    if(e) {
        unsigned char *c = (unsigned char *) &result;

        c = c + sizeof(unsigned long long int) - bytewidth;
        memcpy(c, scratchbuf, bytewidth);
    } else memcpy(&result, scratchbuf, bytewidth);
}

/* result now holds a valid integer of the native endian for what we read!
   Of course, that may not be our endian. */

if(e != our_endian) swap_endian(&result);

#ifdef __KERNEL__
    // printk("Returning %d.\n", (int) result);
#endif
return result;
}

void hsl_CORE_write_clips(struct cliplist *l) {
    while(l) {
        struct clip *c = (struct clip *) l->c;
        if(c->location) hsl_CORE_write_char(*c);
        else hsl_CORE_write_single_integer(*c);
        l = l->next;
    }
}

void hsl_CORE_write_single_integer(struct clip c) {
    unsigned long long int value, l, m, w, e;
    int max_bytewidth, bytewidth, uneven, mrev, unevenrev;

```

```

unsigned char mbuf[32];
unsigned char *scratchbuf;
int i;

for(mrev = 0; mrev < 32; mrev++) mbuf[mrev] = '\0';
scratchbuf = mbuf + 1;

l = c.outputloc;
w = c.length;
m = c.outputminorloc;
e = c.endian;
value = c.value;

uneven = w % 8;
unevenrev = 8 - uneven;
mrev = 8 - m;
max_bytewidth = (w + m) / 8 + (((w + m) % 8) ? 1 : 0);
bytewidth = w / 8 + ((w % 8) ? 1 : 0);

/* Uneven, unevenrev, mrev, max_bytewidth and bytewidth are now all
   correct. */

/* We begin with a valid integer. First, we'll make sure it's the right
   endian for what we're writing. */

if(e != our_endian) swap_endian(&value);

/* Let's get a bytewidth-byte integer into the first bytewidth
   bytes of scratchbuf. */

if(e) {
    unsigned char *c = (unsigned char *) &value;

    c = c + sizeof(unsigned long long int) - bytewidth;
    memcpy(scratchbuf, c, bytewidth);
} else memcpy(scratchbuf, &value, bytewidth);

/* Got that. Now, if we don't have an even bytewidth integer and we're
   in big-endian mode, we need to pack everything to the left. */

if(uneven && e) {
    unsigned char *c, *d;
    int i;

    d = scratchbuf;
    c = d + 1;

    for(i = bytewidth; i > 0; i--) {
        *d = (*d << unevenrev) | (*c >> uneven);
        c++; d++;
    }
}

/* Now if the minor point exists, we get to go back to the right, according
   to endian rules. */

if(m) {
    unsigned char *c, *d;
    int i;

    d = scratchbuf + max_bytewidth - 1;
    c = d - 1;

    if(e) { /* Big Endian */
        for(i = 0; i < max_bytewidth; i++) {
            *d = (*d >> m) | (*c << mrev);
            c--; d--;
        }
    } else { /* Little Endian */

```

```

        for(i = 0; i < max_bytewidth; i++) {
            *d = (*d << m) | (*c >> mrev);
            c--; d--;
        }
    }
}

/* Our integer is ready, but we may need to mask it with source data. */

if(m || uneven) {
    struct clip source;
    unsigned char newbuf[32];
    int i;
    int k = (m + uneven) % 8;

    source = hsl_CORE_get_char_data(l, max_bytewidth);
    memcpy(newbuf, source.location, max_bytewidth);

    if(max_bytewidth == 1) {
        newbuf[0] &= (andmask[m*2+(1-e)] | andmask[(mrev-w)*2+(1-e)]);
    } else {
        if(!m) {
            newbuf[0] = '\0';
        } else {
            newbuf[0] &= andmask[m*2+e];
        }

        for(i = 1; i < max_bytewidth - 1; i++) {
            newbuf[i] = '\0';
        }

        if(!k) {
            newbuf[max_bytewidth - 1] = '\0';
        } else {
            newbuf[max_bytewidth - 1] &= andmask[(mrev-w)*2+(1-e)];
        }
    }

    for(i = 0; i < max_bytewidth; i++) {
        newbuf[i] |= scratchbuf[i];
        scratchbuf[i] = newbuf[i];
    }
}

hsl_CORE_write_char_data(l, max_bytewidth, scratchbuf);
}

void hsl_CORE_write_char(struct clip c) {
    hsl_CORE_write_char_data(c.outputloc, c.length, c.location);
}

void hsl_CORE_init(void) {
    /* Check our endianness. */

    unsigned short int endian_check = 0x00FF;
    unsigned char *c = (unsigned char *) &endian_check;

    our_endian = (*c == 0xFF) ? 0 : 1;
}

void hsl_CORE_pushContext(long long int context) {
    thisContext += context;
}

void hsl_CORE_popContext(long long int context) {
    thisContext -= context;
}

-----

```

Source file: hash.c

```
-----
#include "hash.h"

/* A quick, dumb string-key hashing data structure. */

void hash_init(hash *h) {
    h->key = malloc(1);
    h->key[0] = '\0';
    h->data = NULL;
    h->next = NULL;
}

void *hash_lookup(hash *h, char *s) {
    while(h) {
        if(!strcmp(h->key, s)) return h->data;
        h = h->next;
    }
    return NULL;
}

void hash_insert(hash *h, char *s, void *d) {
    int l = strlen(s);
    while(h->next) h = h->next;

    h->next = malloc(sizeof(hash));
    h = h->next;
    h->key = malloc(l + 1);
    strcpy(h->key, s);
    h->data = d;
    h->next = NULL;
}

void hash_delete(hash *h, char *s) {
    hash *oh;

    while(h && strcmp(s, h->key)) {
        oh = h;
        h = h->next;
    }

    if(h) {
        oh->next = h->next;
        free(h->key);
        free(h);
    }
}
-----
```

Source file: linux-ext2.c

```
-----
#include <linux/fs.h>
#include <linux/module.h>
#include <linux/init.h>

#include "hadley.h"
#include "hadleyfs.h"
#include "hadleyugo.h"

hsl_hadleyfs_template *fs;

struct super_block *hadley_read_super(struct super_block *super,
                                     void *opt, int silent);
void hadley_read_inode(struct inode *in);
struct dentry *hadley_lookup_inode(struct inode *in, struct dentry *de);
ssize_t hadley_read(struct file *f, char *c, size_t s, loff_t *l);
int hadley_readdir(struct file *f, void *v, filldir_t filldir);
-----
```

```

struct super_operations hadley_sops = {
    hadley_read_inode,
    NULL,
    NULL,
    NULL, // hadley_write_inode,
    NULL,
    NULL, // hadley_delete_inode,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL, // hadley_stat,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL
};

struct inode_operations hadley_null_iops = {
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL
};

struct inode_operations hadley_dir_iops = {
    NULL, // hadley_create_inode,
    hadley_lookup_inode,
    NULL,
    NULL,
    NULL,
    NULL, // hadley_mkdir,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL
};

struct file_operations hadley_dir_fops = {
    NULL,
    NULL,
    generic_read_dir,
    NULL,
    hadley_readdir,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL
};

```

```

    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL
};

struct file_operations hadley_file_fops = {
    NULL,
    NULL,
    hadley_read,
    NULL, // hadley_write,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL
};

struct file_operations hadley_null_fops = {
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL
};

struct super_block *HSL_super;

struct super_block *hadley_read_super(struct super_block *super,
                                     void *opt, int silent) {
    long long int blocksize, rootid;
    struct inode *root_inode;

    // printk(KERN_ALERT "Attempting to read the superblock.\n");

    // MOD_INC_USE_COUNT;

    // lock_super(super);
    HSL_super = super;
    set_blocksize(super->s_dev, 512);
    super->s_blocksize = 512;
    // printk(KERN_ALERT "Getting block size.\n");

```

```

blocksize = fs->BlockSize(0);
// printk(KERN_ALERT "Getting root directory ID.\n");
rootid = fs->RootDirectoryID(0);
// printk(KERN_ALERT "Block size %d, rootdir %d.\n", (int) blocksize,
//      (int) rootid);

// printk(KERN_ALERT "Setting superblock block size.\n");
super->s_blocksize = blocksize;
set_blocksize(super->s_dev, blocksize);
super->s_op = &hadley_sops;
/* Get the root directory inode */

// printk(KERN_ALERT "Filling the root inode.\n");
root_inode = iget(super, rootid);

// printk(KERN_ALERT "Inserting the root inode.\n");
// insert_inode_hash(root_inode);
super->s_root = d_alloc_root(root_inode);

// unlock_super(super);
// printk(KERN_ALERT "Garbage collecting.\n");
hsl_CORE_garbagecollect();
// printk(KERN_ALERT "Superblock done.\n");
return super;
}

void hadley_read_inode(struct inode *in) {
    long long int fileid = in->i_ino;

    // printk(KERN_ALERT "Attempting to fill inode %d.\n", (int) fileid);

    in->i_uid = in->i_gid = 0;
    in->i_nlink = 128;
    in->i_atime = in->i_ctime = in->i_mtime = 0;
    in->i_mode = S_IRWXU;
    if(fs->FileIsDirectory(0, fileid)) {
        in->i_mode |= S_IFDIR;
        in->i_op = &hadley_dir_iops;
        in->i_fop = &hadley_dir_fops;
        // printk(KERN_ALERT "Calling FilesIn...\n");
        in->i_size = fs->FilesIn(0, fileid);
    } else if(fs->FileIsRegular(0, fileid)) {
        in->i_op = &hadley_null_iops;
        in->i_fop = &hadley_file_fops;
        in->i_size = fs->FileSize(0, fileid);
    } else {
        // special file.
        in->i_op = &hadley_null_iops;
        in->i_fop = &hadley_null_fops;
        in->i_size = 0;
    }

    // printk(KERN_ALERT "Done filling.\n");
}

void hadley_write_inode(struct inode *in, int i) {
    return;
}

void hadley_delete_inode(struct inode *in) {
    return;
}

int hadley_stat(struct super_block *super, struct statfs *stats) {
    return -1;
}

int hadley_create_inode(struct inode *in, struct dentry *de, int i) {
    return -1;
}

```

```

}

struct dentry *hadley_lookup_inode(struct inode *in, struct dentry *de) {
    struct inode *result = NULL;
    int i;
    long long int fileid;

    for(i = 0; i < fs->FilesIn(0, in->i_ino); i++) {
        fileid = fs->nthFileIn(0, in->i_ino, i);
        if(!strcmp(de->d_name.name, hsl_CORE_cliptostring(fs->FileName(0, fileid))))
        {
            result = iget(in->i_sb, fileid);
        }
    }

    d_add(de, result);
    hsl_CORE_garbagecollect();
    return NULL;
}

int hadley_mkdir(struct inode *in, struct dentry *de, int i) {
    return -1;
}

ssize_t hadley_read(struct file *f, char *c, size_t s, loff_t *l) {
    long long int fileid = f->f_dentry->d_inode->i_ino;
    long long int bs = fs->BlockSize(0);
    long long int filesize = fs->FileSize(0, fileid);
    char *buf;
    int blocks;
    long long int bufpos = 0;
    int i;
    long long int firstblock;

    if(s + *l >= filesize) s -= (s + *l - filesize);
    if(s <= 0) return 0;

    blocks = s / bs;
    if(s % bs) blocks++;
    if((*l % bs) + (s % bs) > bs) blocks++;
    bufpos = 0;
    buf = (char *) malloc(blocks * bs);
    firstblock = (*l) / bs;

    for(i = 0; i < blocks; i++) {
        memcpy(buf + bufpos, (fs->FileBlock(0, fileid, firstblock + i)).location, bs);
        hsl_CORE_garbagecollect();
        bufpos += bs;
    }

    memcpy(c, buf + (*l % bs), s);

    free(buf);
    hsl_CORE_garbagecollect();
    *l += s;
    return s;
}

ssize_t hadley_write(struct file *f, const char *c, size_t s, loff_t *l) {
    return -1;
}

int hadley_readdir(struct file *f, void *v, filldir_t filldir) {
    /* We flatly cheat here with f_pos. */
    long long int rootid = fs->RootDirectoryID(0);
    long long int directoryid = f->f_dentry->d_inode->i_ino;
    long long int pos = f->f_pos;
    long long int filecount;
    long long int fileid;

```



```

char *filename;

// printk("Attempting to read directory.\n");
filecount = fs->FilesIn(0, directoryid);

/* Try without this.
if(directoryid == rootid) {
    switch(pos) {
        case 0:
            (filp->fpos)++;
            filldir(de, ".", 1, pos, rootid);
            return 0;
        case 1:
            (filp->fpos)++;
            filldir(de, "..", 2, pos, 0);
            return 0;
        default:
            pos -= 2;
            filecount += 2;
    }
} */

if(pos < filecount) {
    fileid = fs->nthFileIn(0, directoryid, pos);
    filename = hsl_CORE_cliptostring(fs->FileName(0, fileid));
    filldir(v, filename, strlen(filename), pos, fileid, DT_UNKNOWN);
    // break;
    // filldir(v, filename, strlen(filename), pos, fileid,
    // fs->FileIsDirectory(0, fileid) ? DT_DIR : DT_REG);
    pos++;
    f->f_pos++;
}

hsl_CORE_garbagecollect();
return 0;
}

DECLARE_FSTYPE_DEV(hadley_fs_type, "HadleyFS", hadley_read_super);

static int __init init_hadleyfs_fs(void) {
    // printk(KERN_ALERT "Hadley is initializing.\n");
    hsl_ext2_register_hadleyfs();
    hsl_fat12_register_hadleyfs();
    // printk(KERN_ALERT "Hadley filesystem types registered.\n");
    fs = get_hsl_hadleyfs_implementation("ext2");
    // printk(KERN_ALERT "Got implementation.\n");
    return register_filesystem(&hadley_fs_type);
    // printk(KERN_ALERT "Registered with the Linux kernel.\n");
}

static void __exit exit_hadleyfs_fs(void) {
    unregister_filesystem(&hadley_fs_type);
}

module_init(init_hadleyfs_fs)
module_exit(exit_hadleyfs_fs)

-----
Source file: linux-fat12.c
-----

#include <linux/fs.h>
#include <linux/module.h>
#include <linux/init.h>

#include "hadley.h"
#include "hadleyfs.h"
#include "hadleyugo.h"

```





```

// MOD_INC_USE_COUNT;

// lock_super(super);
HSL_super = super;
set_blocksize(super->s_dev, 512);
super->s_blocksize = 512;
// printk(KERN_ALERT "Getting block size.\n");
blocksize = fs->BlockSize(0);
// printk(KERN_ALERT "Getting root directory ID.\n");
rootid = fs->RootDirectoryID(0);
// printk(KERN_ALERT "Block size %d, rootdir %d.\n", (int) blocksize,
//      (int) rootid);

// printk(KERN_ALERT "Setting superblock block size.\n");
super->s_blocksize = blocksize;
set_blocksize(super->s_dev, blocksize);
super->s_op = &hadley_sops;
/* Get the root directory inode */

// printk(KERN_ALERT "Filling the root inode.\n");
root_inode = iget(super, rootid);

// printk(KERN_ALERT "Inserting the root inode.\n");
// insert_inode_hash(root_inode);
super->s_root = d_alloc_root(root_inode);

// unlock_super(super);
// printk(KERN_ALERT "Garbage collecting.\n");
hsl_CORE_garbagecollect();
// printk(KERN_ALERT "Superblock done.\n");
return super;
}

void hadley_read_inode(struct inode *in) {
    long long int fileid = in->i_ino;

    // printk(KERN_ALERT "Attempting to fill inode %d.\n", (int) fileid);

    in->i_uid = in->i_gid = 0;
    in->i_nlink = 128;
    in->i_atime = in->i_ctime = in->i_mtime = 0;
    in->i_mode = S_IRWXU;
    if(fs->FileIsDirectory(0, fileid)) {
        in->i_mode |= S_IFDIR;
        in->i_op = &hadley_dir_iops;
        in->i_fop = &hadley_dir_fops;
        // printk(KERN_ALERT "Calling FilesIn...\n");
        in->i_size = fs->FilesIn(0, fileid);
    } else if(fs->FileIsRegular(0, fileid)) {
        in->i_op = &hadley_null_iops;
        in->i_fop = &hadley_file_fops;
        in->i_size = fs->FileSize(0, fileid);
    } else {
        // special file.
        in->i_op = &hadley_null_iops;
        in->i_fop = &hadley_null_fops;
        in->i_size = 0;
    }

    // printk(KERN_ALERT "Done filling.\n");
}

void hadley_write_inode(struct inode *in, int i) {
    return;
}

void hadley_delete_inode(struct inode *in) {
    return;
}

```

```

}

int hadley_stat(struct super_block *super, struct statfs *stats) {
    return -1;
}

int hadley_create_inode(struct inode *in, struct dentry *de, int i) {
    return -1;
}

struct dentry *hadley_lookup_inode(struct inode *in, struct dentry *de) {
    struct inode *result = NULL;
    int i;
    long long int fileid;

    for(i = 0; i < fs->FilesIn(0, in->i_ino); i++) {
        fileid = fs->nthFileIn(0, in->i_ino, i);
        if(!strcmp(de->d_name.name, hsl_CORE_cliptostring(fs->FileName(0, fileid))))
        {
            result = iget(in->i_sb, fileid);
        }
    }

    d_add(de, result);
    hsl_CORE_garbagecollect();
    return NULL;
}

int hadley_mkdir(struct inode *in, struct dentry *de, int i) {
    return -1;
}

ssize_t hadley_read(struct file *f, char *c, size_t s, loff_t *l) {
    long long int fileid = f->f_dentry->d_inode->i_ino;
    long long int bs = fs->BlockSize(0);
    long long int filesize = fs->FileSize(0, fileid);
    char *buf;
    int blocks;
    long long int bufpos = 0;
    int i;
    long long int firstblock;

    if(s + *l >= filesize) s -= (s + *l - filesize);
    if(s <= 0) return 0;

    blocks = s / bs;
    if(s % bs) blocks++;
    if((*l % bs) + (s % bs) > bs) blocks++;
    bufpos = 0;
    buf = (char *) malloc(blocks * bs);
    firstblock = (*l) / bs;

    for(i = 0; i < blocks; i++) {
        memcpy(buf + bufpos, (fs->FileBlock(0, fileid, firstblock + i)).location, bs);
        hsl_CORE_garbagecollect();
        bufpos += bs;
    }

    memcpy(c, buf + (*l % bs), s);

    free(buf);
    hsl_CORE_garbagecollect();
    *l += s;
    return s;
}

ssize_t hadley_write(struct file *f, const char *c, size_t s, loff_t *l) {
    return -1;
}

```

```

int hadley_readdir(struct file *f, void *v, filldir_t filldir) {
    /* We flatly cheat here with f_pos. */
    long long int rootid = fs->RootDirectoryID(0);
    long long int directoryid = f->f_dentry->d_inode->i_ino;
    long long int pos = f->f_pos;
    long long int filecount;
    long long int fileid;
    char *filename;

    // printk("Attempting to read directory.\n");
    filecount = fs->FilesIn(0, directoryid);

    /* Try without this.
    if(directoryid == rootid) {
        switch(pos) {
            case 0:
                (filp->fpos)++;
                filldir(de, ".", 1, pos, rootid);
                return 0;
            case 1:
                (filp->fpos)++;
                filldir(de, "..", 2, pos, 0);
                return 0;
            default:
                pos -= 2;
                filecount += 2;
        }
    } */

    if(pos < filecount) {
        fileid = fs->nthFileIn(0, directoryid, pos);
        filename = hsl_CORE_cliptostring(fs->FileName(0, fileid));
        filldir(v, filename, strlen(filename), pos, fileid, DT_UNKNOWN);
        // break;
        // filldir(v, filename, strlen(filename), pos, fileid,
        // fs->FileIsDirectory(0, fileid) ? DT_DIR : DT_REG);
        pos++;
        f->f_pos++;
    }

    hsl_CORE_garbagecollect();
    return 0;
}

DECLARE_FSTYPE_DEV(hadley_fs_type, "HadleyFS", hadley_read_super);

static int __init init_hadleyfs_fs(void) {
    // printk(KERN_ALERT "Hadley is initializing.\n");
    hsl_ext2_register_hadleyfs();
    hsl_fat12_register_hadleyfs();
    // printk(KERN_ALERT "Hadley filesystem types registered.\n");
    fs = get_hsl_hadleyfs_implementation("fat12");
    // printk(KERN_ALERT "Got implementation.\n");
    return register_filesystem(&hadley_fs_type);
    // printk(KERN_ALERT "Registered with the Linux kernel.\n");
}

static void __exit exit_hadleyfs_fs(void) {
    unregister_filesystem(&hadley_fs_type);
}

module_init(init_hadleyfs_fs)
module_exit(exit_hadleyfs_fs)

```

```

-----
Source file: maintest.c
-----

```

```

#include "hadley.h"
#include "test.h"
/* Hadley System 0.1 */

/* This is a test version of the HCL. All it does is loads a single file,
sets the context to be the buffer for that file and runs some test
functions. */

char mainbuffer[] = "ABCDEFGHJKLMNOPQRSTUVWXYZ";
char intbuffer[] = {0xAB, 0x0C, 0x00, 0xDE, 0xF0, 0x00};

char *context;

char *hcl_get_context_current() { return context; }

int main(void) {
    int i;

    hsl_CORE_init();
    context = mainbuffer;

    printf("Character test. Should be EFG.\n");
    printf("HSL -> %s\n", hsl_CORE_cliptostring(hsl_test_function_barfun(0)));
    printf("\n");

    context = intbuffer;
    printf("Integer tests.\n");
    printf("Should be AB0C.\n");
    printf("HSL -> %x\n", (int) hsl_test_function_foolfun(0));
    printf("Should be CAB.\n");
    printf("HSL -> %x\n", (int) hsl_test_function_foo2fun(0));
    printf("Should be B0C0.\n");
    printf("HSL -> %x\n", (int) hsl_test_function_foo3fun(0));
    printf("What the heck should this be, anyway?\n");
    printf("HSL -> %x\n", (int) hsl_test_function_foo4fun(0));
    printf("Little endian tests. Should be CAB, 000, 0DE, 00F.\n");
    for(i = 0; i < 4; i++) {
        printf("HSL -> %x\n", (int) hsl_test_function_fooarray1fun(0, i));
    }
    printf("Big endian tests. Should be AB0, C00, DEF, 000.\n");
    for(i = 0; i < 4; i++) {
        printf("HSL -> %x\n", (int) hsl_test_function_fooarray2fun(0, i));
    }
    printf("Static subspace test. Should be 00DE.\n");
    printf("HSL -> %x\n", (int) hsl_test_function_sstest(0));
    printf("Dynamic subspace test. Should be DEF0.\n");
    printf("HSL -> %x\n", (int) hsl_test_function_withtest(0));
    return 0;
}

```

-----  
Source file: test-ext2temp.c  
-----

```

/* Hadley System 0.1 */

/* This is a test version of the HCL. All it does is loads a single file,
sets the context to be the buffer for that file and runs some test
functions. */

#define CONTEXT_SIZE 7065600
#define CONTEXT_FNAME "ext2disk.in"
#define FSTYPE "ext2"

int main(void) {
    testbyteplate("ext2", "ext2disk.in", 7065600);
    return 0;
}

```

```

-----
Source file: test-fat12temp.c
-----

/* Hadley System 0.1 */

/* This is a test version of the HCL. All it does is loads a single file,
   sets the context to be the buffer for that file and runs some test
   functions. */

int main(void) {
    testbytemplate("fat12", "fat12disk.in", 1500000);
    return 0;
}

-----
Source file: test-temp.c
-----

/* Hadley System 0.1 */

/* This is a test version of the HCL. All it does is loads a single file,
   sets the context to be the buffer for that file and runs some test
   functions. */

#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>

#include "hadley.h"
#include "hadleyfs.h"
#include "hadleyugo.h"
#include "hadleyfsw.h"
#include "fat12.h"
#include "ext2.h"

char *context;

hsl_hadleyfs_template *fs = NULL;
hsl_hadleyugo_template *ugo = NULL;
hsl_hadleyfsw_template *fsw = NULL;

#define EXTRACT_MAX_SIZE 1024000

void indent(int level) {
    int i;

    for(i = 0; i < level; i++) {
        printf(" ");
    }
}

long long int getFileIDfor(long long int directoryid, char *name) {
    long long int nfiles = fs->FilesIn(0, directoryid);
    long long int k = 0;
    int i;
    long long int fileid;

    for(i = 0; i < nfiles; i++) {
        fileid = fs->nthFileIn(0, directoryid, i);
        if(!strcmp(hsl_CORE_cliptostrng(fs->FileName(0, fileid)), name)) k = fileid;
    }

    return k;
}

int doDirectory(long long int rootid, int level, int chop) {
    int i;

```



```

long long int nfiles;
static char dirstack[4096];

if(!level) strcpy(dirstack, "extract-temp/");

nfiles = fs->FilesIn(0, rootid);
indent(level);
printf(" There are %d files in this directory.\n", (int) nfiles);

for(i = 0; i < nfiles; i++) {
    char name[4096];
    char extractname[4096];
    long long int ID, size, isDirectory, isRegular;

    ID = fs->nthFileIn(0, rootid, i);
    size = fs->FileSize(0, ID);
    isDirectory = fs->FileIsDirectory(0, ID);
    isRegular = fs->FileIsRegular(0, ID);
    strcpy(name, hsl_CORE_cliptostring(fs->FileName(0, ID)));

    indent(level);
    printf(" File %d ID %d name %s size %d %s ",
        i, (int) ID, name, (int) size,
        (isDirectory) ? "dir" :
            (isRegular) ? "file" :
                "special");

    if(ugo) {
        printf("%c%c%c%c%c%c%c%c%c %c%c%c",
            ugo->UsrRead(0, ID) ? 'r' : '-',
            ugo->UsrWrit(0, ID) ? 'w' : '-',
            ugo->UsrExec(0, ID) ? 'x' : '-',
            ugo->GrpRead(0, ID) ? 'r' : '-',
            ugo->GrpWrit(0, ID) ? 'w' : '-',
            ugo->GrpExec(0, ID) ? 'x' : '-',
            ugo->OthRead(0, ID) ? 'r' : '-',
            ugo->OthWrit(0, ID) ? 'w' : '-',
            ugo->OthExec(0, ID) ? 'x' : '-',
            ugo->Sticky(0, ID) ? 's' : '-',
            ugo->SetUID(0, ID) ? 'u' : '-',
            ugo->SetGID(0, ID) ? 'g' : '-');
    }
    printf("\n");
    if(isDirectory && strcmp(name, ".") && strcmp(name, "..")) {
        strcat(dirstack, name);
        strcat(dirstack, "/");
        mkdir(dirstack, S_IRWXU|S_IRWXG|S_IRWXO);
        doDirectory(ID, level + 1, strlen(name) + 1);
    } else if(isRegular && size < EXTRACT_MAX_SIZE) {
        long long int block = 0;
        long long int j = size;
        FILE *ofp;

        strcpy(extractname, dirstack);
        strcat(extractname, name);
        ofp = fopen(extractname, "w");
        while(j > 0) {
            struct clip c;

            c = fs->FileBlock(0, ID, block);
            fwrite(c.location, c.length, 1, ofp);
            j -= c.length;
            block++;
        }
        fclose(ofp);
    }
    hsl_CORE_garbagecollect();
}
dirstack[strlen(dirstack) - chop] = '\\0';
}

```

```

void testbytemplate(char *FSTYPE, char *CONTEXT_FNAME, int CONTEXT_SIZE) {
    FILE *ifp;
    long long int rootid;
    size_t fsize;

    hsl_CORE_init();

#ifdef 0
    ifp = fopen(CONTEXT_FNAME, "rb");
    fseek(ifp, 0, SEEK_END);
    fsize = ftell(ifp);
    rewind(ifp);
    printf("Reading from image file %s, size is %d.\n", CONTEXT_FNAME, fsize);
    context = malloc(fsize);
    fread(context, fsize, (size_t) 1, ifp);
    fclose(ifp);

    hsl_CORE_set_context_memory_mode();
    hsl_CORE_set_context_memory_location(context);
#endif

    ifp = fopen(CONTEXT_FNAME, "r+b");
    if(!ifp) { printf("WARNING: Could not open file %s\n", CONTEXT_FNAME); }
    fseek(ifp, 0, SEEK_END);
    fsize = ftell(ifp);
    rewind(ifp);
    printf("Reading from image file %s, size is %d.\n", CONTEXT_FNAME, fsize);

    hsl_CORE_set_context_file_mode();
    hsl_CORE_set_context_file(ifp);

    printf("File context set.\n");

    hsl_fat12_register_hadleyfs();
    hsl_fat12_register_hadleyfsw();
    hsl_ext2_register_hadleyfs();
    hsl_ext2_register_hadleyugo();

    printf("Templates registered.\n");

    printf("Getting FS implementation...\n");
    fs = get_hsl_hadleyfs_implementation(FSTYPE);
    if(fs->FSSupportsUGO(0)) {
        printf("Getting UGO implementation...\n");
        ugo = get_hsl_hadleyugo_implementation(FSTYPE);
    }
    if(fs->FSSupportsFSW(0)) {
        printf("Getting FSW implementation...\n");
        fsw = get_hsl_hadleyfsw_implementation(FSTYPE);
    }
    printf("Got implementations.\n");

    rootid = fs->RootDirectoryID(0);
    printf("Root directory ID is %d.\n", (int) rootid);

    if(0) {
// if(fsw) {
        struct cliplist *l;
        long long int fileid;
        char block[16384];
        int i;

        for(i = 0; i < 16384; i++) block[i] = '.';
        strcpy(block, "Feh.");

        printf("Writing test.\n");
        printf("  Creating file NEWFTEST.HSL.\n");
    }
}

```

```

    l = fsw->NewFile(0, rootid, hsl_CORE_stringtoclip("NEWFTEST.HSL"));
    hsl_CORE_write_clips(1);
    hsl_CORE_garbagecollect();

    fileid = getFileIDfor(rootid, "NEWFTEST.HSL");
    printf("  File ID is %d.\n", (int) fileid);
    printf("  Writing into file.\n");

    l = fsw->AppendBlock(0, fileid, hsl_CORE_memtoclip(block, fs->BlockSize(0)));
    hsl_CORE_write_clips(1);
    hsl_CORE_garbagecollect();
    l = fsw->SetSize(0, fileid, fs->BlockSize(0));
    hsl_CORE_write_clips(1);
    hsl_CORE_garbagecollect();
    l = fsw->AppendBlock(0, fileid, hsl_CORE_memtoclip(block, 5));
    hsl_CORE_write_clips(1);
    hsl_CORE_garbagecollect();
    l = fsw->SetSize(0, fileid, fs->BlockSize(0) + 5);
    hsl_CORE_write_clips(1);
    hsl_CORE_garbagecollect();

    printf("  Creating directory NEWDTEST.\n");

    l = fsw->NewDirectory(0, rootid, hsl_CORE_stringtoclip("NEWDTEST"));
    hsl_CORE_write_clips(1);
    hsl_CORE_garbagecollect();

    printf("Done.\n");

    fileid = getFileIDfor(rootid, "NEWDTEST");
    printf("  File id is %d.\n", (int) fileid);

    printf("  Creating file SUBFILE.\n");

    l = fsw->NewFile(0, fileid, hsl_CORE_stringtoclip("SUBFILE"));
    hsl_CORE_write_clips(1);
    hsl_CORE_garbagecollect();

    fileid = getFileIDfor(fileid, "SUBFILE");
    printf("  File ID is %d.\n", (int) fileid);
    printf("  Writing into file.\n");

    l = fsw->AppendBlock(0, fileid, hsl_CORE_memtoclip(block, fs->BlockSize(0)));
    hsl_CORE_write_clips(1);
    hsl_CORE_garbagecollect();
    l = fsw->SetSize(0, fileid, fs->BlockSize(0));
    hsl_CORE_write_clips(1);
    hsl_CORE_garbagecollect();
    l = fsw->AppendBlock(0, fileid, hsl_CORE_memtoclip(block, 5));
    hsl_CORE_write_clips(1);
    hsl_CORE_garbagecollect();
    l = fsw->SetSize(0, fileid, fs->BlockSize(0) + 5);
    hsl_CORE_write_clips(1);
    hsl_CORE_garbagecollect();
}

printf("Reading test.\n");
mkdir("extract-temp", S_IRWXU|S_IRWXG|S_IRWXO);
doDirectory(rootid, 0, 0);
fclose(ifp);
}

```

```

-----
Source file: yacc-dummy.c
-----

```

```
#include <stdio.h>
```

```
extern FILE *yyin;

void yyerror (const char *error) { perror(error); }

int main(void) {
    yyin = stdin;
    return yyparse();
}
```

## **APPENDIX C: HSL GRAMMAR (YACC FORMAT)**

```

start:                module;
module:               rootdeclarations;

rootdeclarations:     rootdeclaration rootdeclarations
                      | rootdeclaration;

rootdeclaration:      spacetypedeclaration
                      | templatedeclaration;

spacetypedeclaration: SPACETYPE spacetypename
                      '{' spacetypebody '}'

spacetypename:        STRING;
spacetypebody:        usesp componentdeclarations;

usesp:                /* empty */
                      | uses;

uses:                 use
                      | use uses;
use:                  USE username '.';
username:              STRING;

componentdeclarations: componentdeclaration componentdeclarations
                      | componentdeclaration;

componentdeclaration: functiondeclaration
                      | variabledeclaration
                      | subspacedeclaration
                      | implementdeclaration;

functiondeclaration:  FUNCTION functiondeclname functiontypepart
                      '(' functionparametersp ')'
                      functionwithpartp
                      '{' functionbody '}';
functiondeclname:     STRING;
functiontypepart:     IS STRING;
functionparametersp:  /* empty */
                      | functionparameters;
functionparameters:   functionparameter ','
                      | functionparameters;
functionparameter:    functionparameter
                      | functionparametername
                      | functionparametertype;
functionparametername: STRING;
functionparametertype: IS variabletypename;
functionwithpartp:    /* empty */
                      | WITH functionwiths;
functionwiths:        functionwith ',' functionwiths
                      | functionwith;
functionwith:          functionwithname IS functionwithtype
                      AT expression;
functionwithname:      STRING;
functionwithtype:      STRING;
functionbody:          expression;

```

```

/* Here we deal with variables. */

variabledeclaration:      VAR variablename
                           variabletypepart
                           variableendianpart
                           variablewidthpart
                           variablelocationpart '.';
variablename:            STRING;
variablepublicpart:      /* empty */
                           | PUBLIC;
variabletypepart:        IS variablepublicpart variabletypename
                           variablearraypart;
variabletypename:        STRING;
variablearraypart:       /* empty */
                           | '[' expression ']';
variablewidthpart:       /* empty */
                           | WIDTH expression;
variablelocationpart:    AT expression variableminorpart;
variableminorpart:       /* empty */
                           | MINOR expression;
variableendianpart:      /* Empty */
                           | BIGEND
                           | LITTLEEND;

subspacedeclaration:     SUBSPACE subspacename IS subspacetype
                           AT expression '.';
subspacename:            STRING;
subspacetype:            STRING;

implementdeclaration:     IMPLEMENT implementname
                           '{' implementfunctions '}';
implementname:           STRING;
implementfunctions:       implementfunction
                           | implementfunction ',' implementfunctions;
implementfunction:       STRING;

/* Here we deal with expressions. */

expression:              literal
                           | qualifiedreference
                           | conditional
                           | fparameter;

qualifiedreference:       '$' variablereference
                           | '@' functioncall
                           | '&' '!' variablemajorlocation
                           | '&' '.' variableminorlocation;

literal:                 literalstring literalinteger
                           | literalstring
                           | literalinteger;
literalstring:           QSTRING;
literalinteger:          INTEGER;

variablereference:       variablename

```

```

                                variablerefarraypart;

variablemajorlocation:    variablename
                                variablelocarraypart;

variableminorlocation:    variablename
                                variablelocarraypart;

variablelocarraypart:    /* empty */
                        | '[' expression ']';

variablerefarraypart:    /* empty */
                        | '[' expression ']'
                        | '<' expression ',' expression '>';

functioncall:            functionname '(' functionargumentspart ')';
functionname:            THIS '.' functionlocalname;
                        | functionspacename '.' functionremotename;
functionspacename:        STRING;
functionremotename:        STRING;
functionlocalname:        STRING;
functionargumentspart:    /* empty */
                        | functionarguments;
functionarguments:        functionargument ','
                        | functionarguments
                        | functionargument;
functionargument:        expression;

fparameter:            STRING;

conditional:            ifconditional;

ifconditional:            ifheader '{' expression '}' elsepart;
ifheader:                IF '(' expression ')';
elsepart:                /* empty */
                        | ELSE '{' expression '}';

templatedeclaration:    TEMPLATE templatename
                        | '{' templatebody '}';
templatename:            STRING;
templatebody:            tfunctiondeclarations;

tfunctiondeclarations:    tfunctiondeclaration tfunctiondeclarations
                        | tfunctiondeclaration;

tfunctiondeclaration:    FUNCTION tfunctiondeclname tfunctiontypepart
                        | '(' tfunctionparametersp ')';
tfunctiondeclname:        STRING;
tfunctiontypepart:        IS STRING;
tfunctionparametersp:    /* empty */
                        | tfunctionparameters;
tfunctionparameters:        tfunctionparameter ','
                        | tfunctionparameters
                        | tfunctionparameter

```



```
tfunctionparameter:      tfunctionparametername  
                          tfunctionparametertype;  
tfunctionparametername:  STRING;  
tfunctionparametertype:  IS variabletypename;
```

## LIST OF REFERENCES

- Aho, Alfred and Ullman, Jeffrey. *Foundations of Computer Science*. (1992) W.H. Freeman, New York. ISBN 0-7167-8233-2.
- Anton, Marius. *Romanian Mini Linux*. (2003) Available at URL:  
<http://www.geocities.com/rominlinux/>
- Barjaktarovic, Milica; Chin, Shiu-Kai and Jabbour, Kamal. “Formal specification and verification of the kernel functional unit of the OSI session layer protocol and service using CCS”. *Proceedings of the 1996 ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, New York.
- Broy, Manfred and Ștefănescu, Gheorghe. “The algebra of stream processing functions”. *Theoretical Computer Science* 258 (1-2) pp. 99-129 (2001). Elsevier, London.
- Card, Rémy; Ts'o, Theodore; and Tweedie, Stephen. “Design and Implementation of the Second Extended Filesystem.” *First Dutch International Symposium on Linux*, Amsterdam.  
Available at URL: <http://web.mit.edu/tytso/www/linux/ext2intro.html>.
- Carrier, Brian. “Defining Digital Forensic Examination and Analysis Tools Using Abstraction Layers”. *International Journal of Digital Evidence* 1 (4) (2002). Available at URL:  
<http://www.ijde.org/>. Syracuse University, Utica.
- Chou, Andy; Yang, Junfeng; Chelf, Benjamin; Hallem, Seth and Engler, Dawson. “An Empirical Study of Operating Systems Errors.” *Proceedings of the 18th Symposium on Operating Systems Principles (SOSP)* (2001). ACM, New York.
- Ciancarini, Paolo; Fogli, Daniela and Gaspari, Mauro. “A declarative coordination language”. *Computer Languages* 26 (2-4), pp. 125-163 (2000). Elsevier, London.

Cisco Systems. *Cisco Certified Network Associate Examination*. Document #640-607 (Retired).

Cisco Systems, San Jose.

*Daubert v. Merrill Dow Pharmaceuticals* (509 U.S. 579, 1993)

Defense Advanced Research Projects Agency Internet Program. *RFC 793: Transmission*

*Control Protocol* (1981). Available at URL: <http://www.rfc-editor.org/rfc/rfc793.txt>.

DriveSavers Data Recovery. <http://www.drivesavers.com/index.html>

Gerber, Matthew and Leeson, John. "Shrinking the Ocean: Formalizing I/O Methods in Modern

Operating Systems." *International Journal of Digital Evidence* 1 (2) (2002). Available

at URL: <http://www.ijde.org/>. Syracuse University, Utica.

Gerber, Matthew and Leeson, John. "Formalization of Computer Input and Output: The Hadley

Model." *Digital Investigation* 1 (3) (2004), pp214-224. Elsevier, London.

Gordon, Andrew. "An Operational Semantics for I/O in a Lazy Functional Language."

*Conference on Functional Programming Languages and Computer Architecture*, pp.

136-145 (1993). ACM, New York.

Gordon, Andrew. *Functional Programming and Input/Output*. Doctoral dissertation. Cambridge

University Press, 1994.

Guidance Software. *Encase* (software package). Information available at URL:

<http://www.guidancesoftware.com/>.

Hall, Jim. *FreeDOS Beta 9* (operating system). Available at URL: <http://www.freedos.org/>.

Heisel, Maritta. "Specification of the Unix file system: A comparative case study". *Algebraic*

*Methodology and Technology Lecture Notes in Computer Science* 936, pp. 475-488

(1995). Springer, New York.

- Heydon, Allen and Tygar, J.D.. “Specifying and Checking Unix Security Constraints”.  
*Computing Systems* 7 (1), pp. 91-112 (1994). MIT Press, Cambridge.
- Hill, Mark; Condon, Anne; Plakal, Manoj and Sorin, Daniel. “A System-Level Specification Framework for I/O Architectures”. *Annual ACM Symposium on Parallel Algorithms and Architecture* pp. 138-147 (1999). ACM, New York.
- International Committee on Information Technology Standards Technical Committee T13: AT Attachment. Information available at URL: <http://www.t13.org>.
- International Committee on Information Technology Standards Technical Committee T10: Lower-Level Interfaces. Information available at URL: <http://www.t10.org>.
- International Organization for Standardization. “Information Technology—Open Systems Interconnection—Basic Reference Model: The Basic Model”. Publication ISO/IEC 7498-1:1994.
- International Organization for Standardization. “Information Technology—Open Systems Interconnection—Basic Reference Model: The Basic Model”. Publication ISO/IEC 7498-1:1994.
- Kjoernes, Thomas. “File Allocation Table: How It Seems To Work”. 2000. Available at URL: <http://home.no.net/tkos/info/fat.html>.
- Kumho Tire Company v. Patrick Carmichael* (526 U.S., 1999)
- Landin, P.J. “A Correspondence Between ALGOL-60 and Church’s Lambda-Notation: Part I”.  
*Communications of the ACM* 8 (2) (1965). ACM, New York.
- Mason, Luke. “Access Corruption: Top Ten Prevention Strategies”. *TechRepublic* (April 2002). CNet Networks, San Francisco.

Microsoft. Windows XP Resource Kit: FAT File System. Available at URL:

<http://www.microsoft.com/>.

New Technologies Inc. *Safeback* (software package). Information available at URL:

<http://www.forensics-intl.com/safeback.html>.

Peterson, John and Chitil, Olaf. *Haskell: A Purely Functional Language*. Available at URL:

<http://www.haskell.org/>.

Poirer, Dave. *The Second Extended File System: Internal Layout*. Available at URL:

<http://ftp.gnu.org/savannah/files/ext2-doc/ext2.rtf>.

Shimizu, Kanna and Dill, David. "Using Formal Specifications for Functional Validation of Hardware Designs". *IEEE Design and Test of Computers* 19(4), pp. 96-106 (2002).

IEEE, Piscataway.

Smotherman, Mark. "A Sequencing-Based Taxonomy of I/O Systems and Review of Historical Machines". *Computer Architecture News* 17 (5), pp. 10-15 (1989). ACM, New York.

Software in the Public Interest, Inc. *Debian GNU/Linux* (operating system). Available at URL:

<http://www.debian.org/>.

Torvalds, Linus et al. *Linux* (operating system kernel). Available at URL:

<http://www.kernel.org/>.

Tweedie, Stephen. "Journaling the Linux ext2fs Filesystem". *LinuxExpo* 1998. Available at

URL: <ftp://ftp.uk.linux.org:/pub/linux/sct/fs/jfs/journal-design.ps.gz>.

United States Department of the Treasury. *iLook* (software package). Information available at

URL: <http://ilook-forensics.org/>.

User-Mode Linux. Available at URL: <http://user-mode-linux.sourceforge.net/>

Wadler, Philip. "Comprehending Monads." *Proceedings of the ACM Conference on Lisp and Functional Programming*. Nice, France (June 1990). Available at URL:  
<http://citeseer.nj.nec.com/wadler92comprehending.html>. ACM, New York.