UNIVERSITY OF CALIFORNIA
Santa Barbara

The Image Processing Workbench

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Geography

by

James Edward Frew, Jr.

Committee in charge:

Professor Jeff Dozier, Chairman
Professor David Simonett
Professor Raymond C. Smith
Professor Richard Kemmerer
Professor Frank Davis

July 1990

31 July 1990

ACKNOWLEDGEMENTS

My foremost debt is to Jeff Dozier, my committee chairman, advisor, colleague, and friend, without whose encouragement I would never have entered graduate school, much less (finally!) finished a Ph.D. I am also indebted to Dave Simonett for building a world-class geography department and acting as intellectual godfather to a generation of graduate students, and in particular for bailing me out of almost every hole, both intellectual and bureaucratic, that I dug myself into. Ray Smith has been a role model both as a scientist and as an administrator; his direction of the Computer Systems Laboratory has made it a model environment for both employees and researchers (or those who happen to be both.) Dick Kemmerer and Frank Davis have provided valuable insights and criticisms from their perspectives of computer science and terrestrial ecology.

Outside of my committee, I have benefited considerably from the advice and ideas of John Bruno, Paul Eggert, John Melack, and Alan Strahler.

IPW could never have evolved as it has without incorporating the experience of several users and programmers. Bob Crippen, Danny Marks, and Zhengming Wan all wrote substantial software for IPW's predecessor QDIPS (the Quick and Dirty Image Processing System), while Mike Frampton and Barbara Marks used QDIPS to complete their master's theses; QDIPS could not have evolved into IPW without their suggestions and encouragement.

The current version of IPW owes a great deal to the programming efforts and overall insight of Ralph Dubayah, Jud Harward, Shusun Li, Judy Paddon, and Jiancheng Shi, and to the suggestions of IPW users Jerome Collin, Bert Davis, Kelly Elder, Earl Hajic, Anne Nolin, and Curtis Woodcock. Judy Paddon deserves a special mention for her efforts in turning an early version of IPW into a useful instructional tool (not to mention sharing the Incredible Shrinking Office with me). Curtis Woodcock of Boston University deserves everything he's gotten for fearlessly promoting IPW as his department's principal image processing system.

A small army of administrative and support personnel kept things running smoothly when my research spilled over into time I should have been spending on other things. In the Geography Department, Teresa Everett has been looking out for me ever since she got here; I certainly owe her more than the $10 bet she won. In the College of Engineering, I must thank Paul Hyder, Alex Kreis, Tom Marazita, Dave Probert, Stu Swartz, Alan Stebbens, and especially Teddi Potter for helping make my life much easier than it had any right to be. Here in the Computer Systems Laboratory, special thanks go to Cindy Fulce, Charleen Johnson, Kae Lecce, and especially Sarah Bretz, who compiled the bibliography and has assisted with almost everything. I would also like to thank Karen Baker, Dave Menzies, and Kirk Waters for putting up with all the flakey computers I was neglecting.

Finally, I must acknowledge the encouragement and support of my colleagues Jon Forrest, Rick Kattelmann, Ralph Milliff, Mark Williams, Shiren Yang, Mike Zika, and especially Kelly Elder, all of whom have from time to time kept the load from getting too heavy, or too light.

Over the years, this work has received funding from the National Aeronautics and Space Administration, the National Oceanic and Atmospheric Administration, the California Space Institute, the Hughes Santa Barbara Research Center, and the Regents of the University of California. The generous support of these institutions is gratefully acknowledged.

This work is dedicated to my parents, James Frew and Rose Marie Frew, who helped me start it, and to my wife, Shannon, who helped me finish it.

VITA

04 November 1954 — born, Long Beach, CA.
1977 — B.A., Geography, University of California, Santa Barbara.
1980 — M.A., Geography, UCSB.
1979 - 1981 — Lecturer, Geography and Computer Science, UCSB.
1981 - 1988 — Programmer, Computer Systems Laboratory, UCSB.
1988 - present — Manager, Computer Systems Laboratory, UCSB.

PUBLICATIONS

Dozier, J., and J. Frew, ‘‘Atmospheric corrections to satellite radiometric data over rugged terrain,’’ *Remote Sensing of Environment*, vol. 11, pp. 191-205, 1981.

Marks, D., J. Dozier, and J. Frew, ‘‘Automated basin delineation from digital elevation data,’’ *Geo-Processing*, vol. 2, pp. 299-311, 1984.

Frew, J., and J. Dozier, ‘‘The QDIPS image processing system,’’ CSL Technical Report, Computer Systems Laboratory, University of California, Santa Barbara, CA, 1984.

Frew, J., ‘‘Registering Thematic Mapper Imagery to Digital Elevation Models,’’ *Proceedings of the Tenth International Symposium on Machine Processing of Remotely Sensed Data*, June 12-14, 1984, Purdue University, West Lafayette, IN, pp. 432-435.

Frew, J., and J. Dozier, ‘‘The Image Processing Workbench - Portable Software for Remote Sensing Instruction and Research,’’ *Proceedings of IGARSS ’86 Symposium*, Zürich, 8-11 Sept. 1986, ESA SP-254, pp. 271-276.

Dozier, J., and Frew, J., ‘‘Rapid Calculation of Terrain Parameters for Radiation Modeling from Digital Elevation Data,’’ *Proceedings of IGARSS ’89 Symposium*, Vancouver, Canada, 10-14 July 1989, pp. 1769-1774.

ABSTRACT

The Image Processing Workbench

by

James Edward Frew, Jr.

This dissertation presents the design and implementation of the Image Processing Workbench (IPW), a UNIX-based image processing software system. IPW is a development environment for algorithms and applications using image data from primarily remote sensing sources.

IPW is simple, extensible, and portable. A well-defined set of basic image processing operations are provided which may be combined to form complex processing sequences. The underlying capabilities of UNIX (sequential interprocess communication, programmable command language, regular command syntax) are exploited rather than duplicated. No particular display hardware is required; instead, several display types are supported as ‘‘sinks’’ for processing pipelines.

IPW has a single, portable image data format that accommodates both integer and floating-point data representations, and an unlimited amount of ancillary information. Images may have an arbitrary number of channels or bands. Point, vector, and polygon data in textual form may be easily inserted into or extracted from image data.

Function libraries and program generation tools are provided for programmers wishing to extend IPW. An emphasis on portability has allowed the migration of IPW source code and image data across a variety of heterogeneous computing environments.

This dissertation examines IPW from the successive points of view of a user, a programmer, and a system maintainer. Complete documentation for all IPW programs and library functions is included, as is an appendix containing the IPW C programming standards.

Table of Contents

CHAPTER 1: INTRODUCTION

Although remote sensing technology has existed for over a century [Simonett 1983], the availability of remotely-sensed image data in digital form is a relatively recent phenomenon, since which time the technologies of remote sensing and digital image processing have been closely linked [Castleman 1979] . The two technologies are individually mature; remotely-sensed image data are commercially available (e.g., [SPOT 1989]), as are image processing systems at all levels of integration (e.g., [Bracken 1983].) However, the use of digital image processing techniques for the acquisition, restitution, and analysis of remotely-sensed data remains an active field of research. There are several reasons for this.

First, the sheer volume of remotely-sensed image data available for processing is staggering: A single Landsat Thematic Mapper (TM) multispectral image occupies over 200 MBytes[1] in digital form [EOSAT 1985], and it is estimated that the digital data acquired during a single year's operation of the proposed EOS platforms will exceed 1 PByte[2] [Chase 1986]. Dealing with data volumes of this magnitude presses current hardware and software environments to (and beyond) the limits of their capabilities.

Second, many of the techniques needed to process and analyze remotely sensed digital imagery have yet to be developed. For example, although multispectral digital imagery has been available for almost two decades, it is only recently that data have been available with fine enough spectral resolution to permit analysis by classical spectroscopic techniques (e.g., [Vane 1988].) Similarly, the increasing spatial resolution of remotely-sensed imagery has led to new questions about scale effects in physical models [Dubayah 1990].

Finally, all phases of digital image processing for remote sensing require both a volume and a variety of ancillary data that are unusual in other image processing applications. For example, the extraction of Earth surface exitance from TM image data requires at least [Dozier 1984]:

- end-to-end characterization of the sensor system geometry, so image pixels may be associated with earth surface locations;
- end-to-end characterization of the sensor system radiometry, so image pixels may be converted to the actual radiance values measured at the satellite;
- coregistered surface parameters (elevation, slope, aspect, etc.);
- atmospheric optical parameters;
- solar geometry.

To be usable for remote sensing applications, an image processing system must maintain these kinds of ancillary data.

1.1. NEED FOR SOMETHING LIKE IPW

The **Image Processing Workbench** (IPW) is a digital image processing software system, designed to address the following general applications of digital image processing to remote sensing:

---

[1] 1 MByte ("megabyte") = 2^{20} (≈ 1 million) bytes

[2] 1 PByte ("petabyte") = 2^{50} (≈ 1 quadrillion) bytes

- algorithm development
- application development
- instruction

The areas addressed by this list reflect the software's grounding in an academic environment. First and foremost, we require a digital image processing environment that is usable for algorithm and applications development by researchers. However, we must not ignore the fact that many research problems involve processing image data in quantities large enough to be indistinguishable from ‘‘operational’’ applications. The research environment must therefore support rapid migration of applications to a production environment.

The distinction between **algorithms** and **applications** is an important one. Algorithms are usually lower-level than applications. Algorithms are often problem-independent, while applications tend to be problem-specific. Algorithms are most often developed by programmers, while applications are most often developed by scientists. Support for both development styles requires a highly flexible system.

Finally, IPW must support instruction, since teaching is a primary mission of the academic environment. It is especially important that the teaching and research systems be as similar as possible, both to minimize the effort involved in maintaining duplicate systems, and to facilitate migration of students from classroom to real-world research problems.

1.2. OVERVIEW OF DISSERTATION

The structure of this dissertation is as follows:

Chapter 2 enumerates the overall design principles by which IPW was developed. Chapter 3 presents a brief functional specification of a IPW.

The next 5 chapters explore IPW at increasing levels of detail, from the successive perspectives of a user, an application programmer, and a system administrator.

Chapter 4 describes the user-level interface to IPW, while Chapter 5 is a complete, albeit terse, reference for all IPW features (commands, files, etc.) encountered by a non-programming user of the system. Together, Chapters 4 and 5 may be considered a ‘‘user's manual’’ for IPW.

Chapter 6 describes the programmer-level interface to IPW, while Chapter 7 is a complete, albeit terse, reference for all IPW features (functions, commands, file formats, etc.) encountered by a programmer attempting to modify or extend IPW. Together, Chapters 6 and 7 may be considered a ‘‘programmer's manual’’ for IPW.

Chapter 8 describes how IPW is installed on a new host system, including issues involved in porting the software to a previously unsupported host configuration. This chapter also discusses how to maintain an existing IPW installation. Chapter 8 may be considered a ‘‘system administrator's manual’’ for IPW.

The final 3 chapters place IPW in a broader perspective. Chapter 9 criticizes the shortcomings of IPW as it is currently implemented, while Chapter 10 summarizes the major contributions to remote sensing image processing embodied in IPW. Chapter 11 suggests some possible directions for the future development of IPW.

An appendix contains the IPW C language coding standard.

It is my intention that this dissertation should provide a definitive background and reference for users of the IPW software, as well as a case study for those interested in the overall problem of image processing software design. The contents have thus been organized to allow an easy separation of tutorial, reference, and analytical material.

CHAPTER 2: DESIGN GOALS

Six broad design goals have guided the implementation of IPW:

- **simplicity**: IPW should be easy to understand and use effectively.
- **extensibility**: IPW should be easy to adapt to applications unforeseen in its original design.
- **portability**: IPW programs and data should be usable in any standard UNIX and C environments.
- **uniqueness**: IPW should complement, rather than duplicate, existing capabilities of the host environments.
- **leveraging UNIX**: IPW should take advantage of the particular strengths of UNIX (pipelines, hierarchical file system, programmable command language, etc.)
- **an open system**: The IPW source code should be freely available.

2.1. SIMPLICITY

Complexity is probably the single most important enemy of software quality. [Meyer 1988]

Simplicity promotes comprehensibility: simple software is easier to understand. In turn, software that is understood is more likely to be trusted and used. Simplicity also promotes maintainability: simple software is easier to debug and modify.

Of course, lack of complexity should not imply lack of necessary functionality. Rather, the temptation to add functionality should always be balanced against the need to keep the entire system comprehensible.

2.1.1. Comprehensible

IPW may be understood at many levels. The casual user may only be concerned with the functioning of a few common commands; e.g., just enough to display an image on a particular output device. A more serious user may need to understand how multiple IPW commands may be combined at the command-script level. An application programmer will need to understand the structure of an IPW command at the source level, and the facilities provided by the IPW function libraries. Finally, the maintainer of IPW may ultimately need to understand the structure and function of any IPW module.

Complexity at any of these levels hinders understanding. Without simplicity as an overall design goal, it is easy to select complex designs which offer more features or greater efficiency. It is easy to forget that these purported gains may be useless if the code embodying them is misused or avoided because its behavior is not fully understood.

2.1.2. Maintainable

Another important consequence of simplicity is ease of maintenance. The less complicated a piece of software is, the easier it is to track down bugs in it. This applies at all levels — command scripts are easier to maintain if the commands they invoke have a limited, well-defined set of options, just as C code is easier to maintain if the algorithms employed are the most straightforward possible.

2.2. EXTENSIBILITY

A key design goal of the IPW implementation is extensibility. It is expected that IPW will never be ‘‘finished’’, but will constantly be modified as new applications arise. It is essential that the design of IPW allow for this evolutionary growth.

2.2.1. No arbitrary limits of magnitude

To the maximum extent possible, there are no limits of magnitude incorporated into IPW. One aspect of this is avoiding static limits where dynamic limits may be used instead, such as in the allocation of internal data structures. Another is to minimize chances of resource exhaustion, such as not reading an entire image into memory if it can be processed in smaller pieces.

Such limits as exist are imposed by the underlying C and UNIX environments. For example, various image dimensions are manipulated as long or unsigned long integers. The number of simultaneously open files in a process is limited by the operating system. Every effort is made to ensure that limits like these are the only ones imposed by IPW.

2.2.2. Design for growth

Like most scientific software, IPW will probably be used long beyond its originally envisioned life span. It is therefore important that the software be designed for growth over time, both in terms of migrating to new platforms, and adding functionality. To this end, almost all of the underlying structures in IPW are open-ended. For example, new types of image header data may be easily added, and are harmlessly ignored by older programs that don’t recognize them. The pipe-filter paradigm, described below, allows new programs to seamlessly inter-operate with existing ones.

2.2.3. New data sources

IPW operates on generic multiband images. Nothing about the software is restricted to or specialized for a particular data source (sensor, data system, etc.). IPW’s internal data format is extremely simple, and it has proven easy to write ingest programs to convert new types of data to the IPW format. This is important, since once an ingest program has been written, the rest of IPW is available to the new data source.

2.3. PORTABILITY

Scientific computing is in a continual state of flux, especially in an academic environment, where new hardware is often available at deep discounts. To take maximum advantage of the available computing power, it is essential that commonly-used software be easily ported to new environments. Moreover, the data must also be portable, especially in a network of heterogeneous machines. Finally, investigators using the software and data must be able to move between these environments without retraining.

2.3.1. Code portability

IPW has been implemented from the beginning with portability in mind. This has been achieved both by a comprehensive coding standard (see Appendix A), and by careful specification of a generic target C and UNIX environment.

2.3.2. Data portability

The ability to run IPW on heterogeneous hardware would be seriously compromised if IPW image data could not be moved from one machine to another of different architecture. IPW handles this by using only ASCII or unsigned binary integer data, and by transparently adjusting the byte order of multi-byte integers. All ancillary data in IPW images are stored in ASCII, and all pixel data are stored as (possibly multi-byte) unsigned integers. All IPW images contain a datum identifying the byte ordering of the machine on which they were created, and the IPW I/O routines transparently adjust the byte order if it differs from that of the machine on which the image is being read.

2.3.3. Investigator portability

The user, programmer, and maintainer interfaces to IPW are identical across all IPW hosts. This frees investigators from having to shift gears when they use IPW on a variety of hardware platforms. Such **investigator portability,** often overlooked in favor of exploiting unique features of particular environments, is essential if IPW is to be usable in a heterogeneous computing environment.

2.4. UNIQUENESS

IPW attempts to fill a unique niche in the set of tools available to a remote sensing investigator. This is not to say that IPW itself is unique in the sense that it is the only available image processing software, but rather that IPW does not duplicate any functionality already found in the UNIX environment, and works well with other tools in that environment.

2.4.1. Avoid reinventing the wheel

The UNIX environment already provides considerable functionality that can be directly utilized by IPW (see §2.5). IPW does not attempt to duplicate this functionality. Unlike many image processing software systems, IPW does not contain a command interpreter, a file system, or any general text-manipulation tools, all of which are already provided by UNIX. While these UNIX facilities may not be perfect, they are certainly adequate, and moreover are uniform across all UNIX implementations. As much as possible, IPW attempts to solve only image processing problems, leaving general command and data management problems to the host system.

2.4.2. Software tools philosophy

A **software tool** may be defined as follows:

> it uses the machine; it solves a general problem, not a special case; and it's so easy to use that people will use it instead of building their own. [Kernighan 1976]

IPW programs are designed as tools, and the IPW function libraries are designed to promote tool construction. Large software systems, and especially image processing systems, have traditionally grown by accretion, with a new function or program added for each new operation to be performed. In IPW, the emphasis is on generic operations (§4.2) out of which more specialized operations may constructed. This both helps to keep the overall size of IPW manageable, and to guarantee that any new additions are of maximum utility.

2.4.3. Data import and export

In order to avoid reinventing the wheel, and to function as tools, IPW programs must easily communicate with non-IPW software. Therefore, a fundamental design decision has been to facilitate the importation of data into IPW, and the exportation of IPW datasets to non-IPW software. It is not uncommon for IPW commands to be implemented as scripts which freely intermingle IPW programs and generic UNIX commands.

2.5. UNIX

The UNIX computing environment provides an abundance of support facilities that IPW takes advantage of. In keeping with the principle of uniqueness, every attempt is made to exploit an existing feature of UNIX before adding a new feature to IPW.

2.5.1. Pipes and filters

One of the principal contributions of UNIX has been the paradigm of pipes and filters [Ritchie 1974]: constraining programs to a single primary input and a single output stream, and providing a command-level operator to connect the output of one command to the primary input of another. This leads to a data-driven model of computing that adapts naturally to image processing [Hunt 1982]. To the extent possible, all IPW programs are written as filters, and combining IPW (and non-IPW) programs into pipelines is fundamental to IPW application-building.

2.5.2. Sequential processing

The pipe-filter model has an important side effect in that it mandates serial processing: a program reading from a pipeline cannot rewind or randomly access its input stream; similarly, data written to a pipe cannot be read back. In some respects this is a salubrious constraint, since implementations of serial processing code tend to be simple and compact; however, some operations (e.g. requantization) require repeated passes over data, which in turn require large memories or the use of scratch files if the filter paradigm is to be maintained.

2.5.3. Shell

The UNIX shell is actually a high-level programming language, whose operands are individual programs (or scripts of programs). IPW takes advantage of this by making liberal use of shell scripts to construct application programs. Experience in shell programming is immediately transferable to IPW application development.

2.5.4. Other UNIX commands (awk, sed, etc.)

Almost all UNIX implementations contain powerful standard utilities that IPW takes advantage of. The `awk` text-processing language and `sed` stream editor are used as prototyping tools for, and often in the final versions of, several applications, since much IPW processing involves manipulating data that are stored in ASCII (e.g. image headers) or may be easily converted between ASCII and binary (e.g. lookup tables). IPW source code is maintained with the `make` recompilation manager and RCS revision control software. In general, any existing UNIX utility that is either widely available, or freely distributable with IPW, may be an integral part of the overall IPW environment.

2.5.5. File system

UNIX's hierarchical file system provides a ready-made image data cataloging system. While a relational model might be more appropriate for an image catalog, the hierarchical model is eminently usable, and is provided on all UNIX implementations.

2.5.6. Command syntax

IPW commands use a UNIX standard command-line syntax [Hemenway 1984] and thus resemble most other UNIX commands. While it may be argued that this syntax is not terribly user-friendly, it is consistent, and once mastered, is portable to all UNIX environments. Here IPW piggybacks on the user's existing UNIX experience, rather than constructing an arguably better but inconsistent interface.

2.5.7. C

The C language is supported in all UNIX environments and is thus the natural implementation language for IPW. C has often been called a "portable assembly language," in that it allows most of the low-level bit and byte manipulations formerly possible only in assembly languages. Such operations are a cornerstone of image processing, another reason why C is a suitable language for IPW. Finally, with suitable discipline, it is possible to write C code that is both readable and efficient, both of which are important for extensible image processing code.

2.6. AN OPEN SYSTEM

IPW is freely distributable. Much of the source code is in the public domain; the remainder is copyrighted but is freely distributable. This is obviously important to academic users with their chronic shortage of funds. However, the main reason IPW is freely available is to maximize its utility as a tool. IPW users may become IPW programmers if they wish, since they will always have access to the source code. Bugs are more likely to be uncovered (and fixed) if the person encountering the bug can debug the program at the source level. Most important of all, IPW is more likely to be extended if its source code is available for study and emulation.

2.7. COMMENTARY

The design principles enumerated in this chapter distinguish IPW from other image processing software environments in several ways. Of these, the most important is the extent to which IPW exploits the standard facilities of UNIX. Most other image processing software environments (e.g., LAS [Wharton 1987] and VICAR [LaVoie 1987]) duplicate such key components of UNIX as the command language or the file system, and thus bring needless complexity to a UNIX host.

Few existing systems have been designed from the outset for portability of both software and data; most are inextricably tied to proprietary operating systems, display hardware, or data representations (e.g. binary floating-point). These systems bind their users to specific hardware and software environments, whereas IPW users may never notice the difference between the variety of UNIX environments in which IPW is implemented.

IPW's emphasis on constructing operations out of pipelines of carefully-specified generic software tools is shared by a few other software systems (e.g., HIPS [Landy 1984] and the IM Raster Toolkit [Paeth 1986a]), but these systems have other shortcomings with respect to the functional specifications outlined in the next chapter, largely due to their having been originally designed for engineering and computer graphic applications, rather than for remote sensing and Earth science.

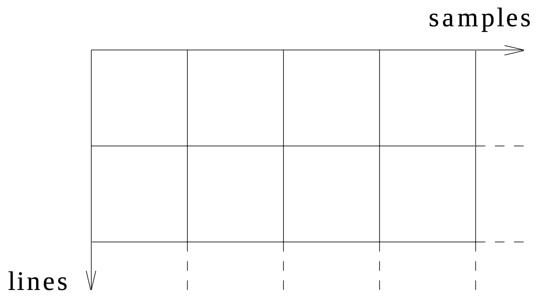
CHAPTER 3: BRIEF FUNCTIONAL SPECIFICATION

This chapter contains a brief functional description of IPW. The system may be characterized from three distinct perspectives:

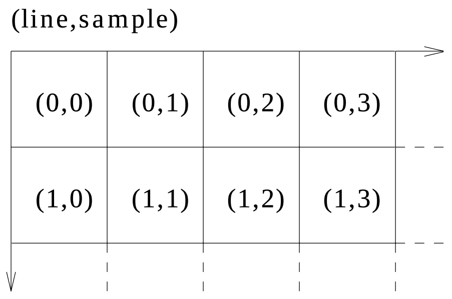
- **image data**: the images and other data objects manipulated by IPW.
- **user environment**: the programs, scripts, and execution environment with which an end-user of IPW interacts.
- **programmer environment**: the tools, function libraries, and source code used by programmers to maintain and extend IPW.

3.1. IMAGES

The fundamental data object manipulated by IPW is an **image**. An image is a rectangular data structure composed of equally-sized elements or **samples**, arranged in **lines** that each contain an equal number of samples.

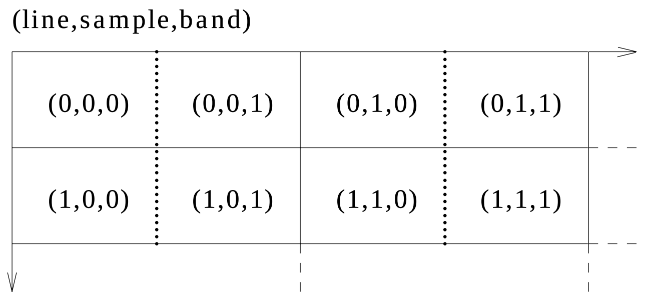


Of the four possible corners of an image, we arbitrarily define the upper left corner as the image **origin**. We also arbitrarily define lines as running horizontally, thus establishing an **intrinsic coordinate system** for the image. Any particular sample in the image may be located by the ordered pair of the number of the line containing the sample (where line 0 is the first line in the image), and the number of the sample within the line (where sample 0 is the first sample in the line):



IPW is oriented towards the types of applications wherein a sample corresponds to an Earth surface location. In these applications, a sample will typically assume multiple coregistered values, each representing a measurement, derivation, or simulation of some physical parameter. Thus, it is essential that IPW support multivariate samples.

In IPW, a sample is a vector of one or more **bands** (e.g., colors, sensor channels, etc.). A **pixel** is the value of a single band within a sample; i.e., it is the datum indexed by a particular (line, sample, band) triple.



Note that the subscript order (line, sample, band) describes a particular **interleaving** strategy; i.e., the order in which lines, samples, and bands appear in the image. IPW images have **BIP** (band interleaved by pixel) interleaving [Simonett 1978], wherein all of the pixels of a given sample are physically as well as logically contiguous. This interleaving is the most appropriate for multiband processing operations.

Like the number of lines or number of samples per line, the number of bands per sample is constant for a given image: an image may not have ‘‘ragged’’ boundaries in any of its 3 dimensions.

Images have two modes in which they may exist. The **external** representation of an image is as either a unique **file** in the UNIX file system, or the contents of a unique interprocess communication channel (e.g., a UNIX pipe). The **internal** representation of an image is as a set of C data structures containing pixel and ancillary data.

3.1.1. Headers

Each IPW image has associated with it several **headers** containing ancillary (i.e., non-pixel) data. Each header specifies a number of related attributes (geographic location, sensor parameters, pixel quantization parameters, etc.) There are no inherent limits to either the number of headers in an image, or the number or complexity of attributes in a particular header.

The only required header (the **basic image header**), specifies the dimensions of the image raster and the storage configuration of each band’s pixels. Another way of stating this is: the only required ancillary information is that which is sufficient to allow the location and extraction of the stored bit pattern for any single pixel.

IPW image header data is stored externally as printable ASCII characters preceding the pixel data; thus, it is highly portable across differing hardware architectures. The use of printable ASCII also means that header data may be examined by humans without the use of special tools, and in non-IPW environments.

Efficiency is achieved by imposing an easily-parsed structure on the header data. At the lowest level, header records are lines of text; i.e., sequences of ASCII characters terminated by a newline character (ASCII NL). This text model pervades the UNIX system and is accommodated by most UNIX tools[3]. Each header text line may be further classified as either a **preamble**, which introduces a new header, or a **datum**, which specifies a single (possibly multi-valued) attribute in a *keyword*=*value*... format.

To facilitate direct viewing of image header data by UNIX text pagination utilities (e.g., `more`), the last header text line (before the beginning of the pixel data) always

[3] See the description of the `edhdr` command in §5.2.

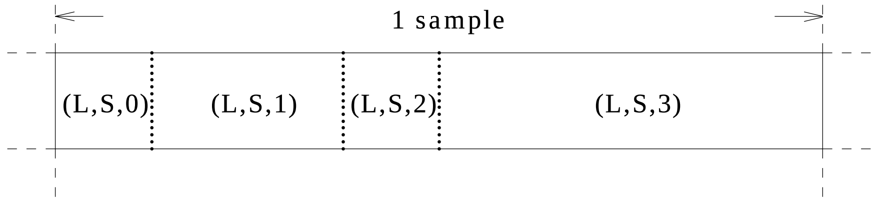
contains a formfeed character (ASCII NP) immediately preceding the terminating newline[4]. This causes the pagination utilities to pause, so they can be terminated before attempting to write image data to a terminal screen. Also, filters such as `sed` or `awk` can use the formfeed to recognize the end of the headers, and thus avoid processing non-ASCII image pixel data.

3.1.2. Pixels

IPW image pixel values are represented externally as unsigned integers. This is the most portable possible binary representation, requiring at most a change of byte order to move between different machine architectures. It is also the traditional pixel representation for digital image data, which simplifies the importation of existing image data into IPW.

Each band of an IPW image has two pixel size parameters associated with it: the number of **bytes** per pixel, and the number of **bits** within those bytes that contain significant data. As a compromise between portability and flexibility, a pixel must occupy either 1, 2, or 4 bytes, but may utilize an arbitrary number of the low-order bits of those bytes[5].

Unlike many other image processing systems, IPW allows different bands in the same image to have different size pixels. This is essential for arbitrarily combining bands of differing precision into a single multiband image. For example, a 4-band image might contain a mix of 1-, 2-, and 4-byte pixels in each sample:



3.1.3. Quantization

Pixel values in remotely sensed images are **quantized** approximations to measurements of "real-world" phenomena, such as radiance, brightness temperature, or surface roughness. These phenomena are typically most conveniently represented as floating-point (i.e., real), rather than integer, quantities. However, for the reasons discussed in the previous section, it is undesirable to use floating-point as an external representation format.

IPW solves this representation problem by storing, in an optional image header, the parameters governing the quantization of pixel values from floating-point to integer[6]. These parameters are **break points** defining a monotonic (hence invertible), piecewise linear transformation. The presence of this optional header will cause IPW components that require floating-point values to automatically perform the required transform to retrieve input pixel values, and to automatically apply the inverse

[4] This trick is borrowed from the IM Raster Toolkit software [Paeth 1986a].

[5] Although an attempt has been made to keep IPW independent of the size of a byte, it has only been used on machines with 8-bit bytes.

[6] This concept derives from the "binary fraction" format [Frew 1984, Paeth 1986a].

transform to quantize output values. This quantization scheme yields the best of both worlds: pixels are stored externally in a portable and easily manipulated format, while an internal representation is available which recovers all of the information inherent in the original data.

For example, a quantization parameter header for an image with a single band of 8-bit pixels might contain the following break points:

```
0 0.0
255 1.0
```

indicating that the range of unsigned integer pixel values from 0 to 255 should be linearly mapped into the range of floating-point values from 0 to 1. The IPW component accessing these pixels will see only values between 0 and 1, and will be unaware of the underlying representation.

3.2. USER ENVIRONMENT

The IPW user environment is intended to be a seamless extension of the UNIX command environment:

- IPW images are ordinary UNIX files.
- The IPW ‘‘command interpreter’’ is the UNIX shell. IPW commands behave as ordinary UNIX commands.
- Scripts (command files) of IPW commands may be created, using the programming facilities of the UNIX shell.
- All components of IPW, including source code, are normally accessible to any IPW user.

These features combine to minimize the marginal cost of learning and using IPW for researchers already familiar with the UNIX system.

3.2.1. Data files

An IPW user perceives IPW as a collection of UNIX commands operating on UNIX files. Although these image files have a definite internal structure imposed by IPW, they appear to UNIX as simply an unstructured sequence of bytes. As such, they can be processed by any normal UNIX file handling utility, such as:

- **compress**, to reduce the size of the stored image without loss of information;
- **tar**, to combine multiple images into a single portable file suitable for distribution;
- **awk** and **sed**, to process image headers.

All of these examples, that a monolithic image processing system would have to provide for itself, are available to IPW ‘‘free’’ as part of the UNIX environment.

The hierarchical UNIX file system provides a useful means of organizing image data, without having to build cataloging or database functionality into IPW itself. The ability to gather related image files into arbitrarily nested directories, while preserving the atomicity of image-as-file, is a powerful data management tool.

3.2.2. Primitives

Just as the basic data units in IPW are image files, the basic functional units of IPW are the **primitives**, the low-level commands embodying fundamental image processing operations.

All IPW commands are executable UNIX files, either compiled programs or command scripts. The directories containing these commands are incorporated into the user's shell command search path[7], so the IPW commands appear to be ‘‘part of the system’’, coequal with the standard UNIX utilities.

The syntax of all IPW commands adheres to standard UNIX usage [Hemenway 1984]:

```
command [options] [operands]
```

All options are single characters preceded by a hyphen. If an option accepts multiple arguments, then the arguments are separated by commas. Operands, if present, always represent input file names. If a command expects an input file and none is specified, then the standard input is assumed. The standard input may also be explicitly specified by a single hyphen in any context where an input file name is expected.

The IPW primitives are all **filters** [Kernighan 1976]; that is, they read input data from the **standard input** and write output data to the **standard output**, both of which are predefined UNIX I/O channels, normally connected to the user's terminal, that may be redirected from or to files by shell operators. Primitives that accept more than one input always provide, via options, for any one of the inputs to come from the standard input. Primitives never produce more than one output.

Besides conceptual simplicity, the filter paradigm has the important consequence of allowing multiple IPW primitives to be sequentially connected via **pipes**, a UNIX operating system construct that links the standard output of one command to the standard input of another. This is the perhaps the single feature of UNIX most effectively exploited by IPW, for it allows IPW primitives to behave as operators in a simple yet powerful image processing ‘‘language.’’

3.2.3. Scripts

The programming capabilities of the UNIX shell allow IPW primitives to be combined in arbitrarily elaborate **scripts** (i.e., executable command files). Scripts are an essential part of IPW. They frequently allow applications to be developed entirely from existing IPW primitives and UNIX commands.

As a simple example, given a primitive `zoom` which magnifies or minifies an image by replicating or skipping an integer number of lines or samples, we can construct a script `ratzoom` which zooms an image by a rationally-expressed non-integer amount:

```
zoom -s numerator | zoom -s -denominator |
        zoom -l numerator | zoom -l -denominator
```

where `-s` and `-l` indicate the number of samples or lines to replicate (positive count) or skip (negative count).

---
[7] in the PATH environment variable

Scripts are interpreted directly by the shell, so there is no compile phase in the script development cycle. This makes scripts an excellent tool for rapidly prototyping a potential application, even if the application is to be eventually coded as a primitive for efficiency's sake.

3.3. PROGRAMMER ENVIRONMENT

The distinction between an IPW user and an IPW programmer is blurred somewhat by the programmability of the UNIX shell. Most IPW users are also ‘‘programmers’’ in that they usually wind up creating command scripts tailored to their particular applications. We therefore use the term **programmer** to identify someone who develops new IPW primitives by programming in the C language[8]. IPW provides the following support for C programmers:

- an extensive function library;
- program development tools;
- program and function templates;
- on-line access to the entire corpus of IPW source code.

3.3.1. Libraries

IPW provides a library of C-callable functions, which may be grouped according to the following capabilities:

- program command line interpretation;
- image header manipulation;
- image pixel manipulation;
- miscellaneous utility functions.

The syntax of an IPW primitive's command line is enforced by a family of functions that handle option and option-argument retrieval; the IPW programmer need only specify, in a single data structure, the options and option-arguments that the program expects.

Image header manipulation is performed by two layers of functions. At the higher level, a set of input, output, and constructor functions are provided for each type of header recognized by IPW. At the lower level, a set of header-independent I/O functions are provided to facilitate the creation of new header types.

Image pixel manipulation is also performed by two layers of functions. The higher level presents pixels as floating point quantities, while the lower level presents them as unsigned integers. Programs may use whichever layer is most appropriate to the particular operation being performed.

Finally, utility functions are provided for common operations, such as allocation of IPW data structures.

[8] While other languages (e.g., Fortran) may certainly be used to write IPW programs, the C language is preferable because of its ubiquity in the UNIX environment.

3.3.2. Tools

IPW provides several tools to assist in the development of IPW programs. The most important of these is `ipwmake`, a front-end for the UNIX `make` utility [Feldman 1979] that generates customized `makefiles` for IPW programs and libraries. The `makefiles` are created by combining generic rules, site-specific rules set up by the IPW administrator, and an absolute minimum of program-specific information.

IPW also provides `lint` libraries corresponding to all IPW functions, allowing the UNIX `lint` utility [Johnson 1978] to guarantee that a program is using the IPW functions correctly. `lint` may be accessed by `ipwmake`, or by the `ipwlint` tool.

3.3.3. Templates

IPW programs and functions often have similar structures, since they must often accomplish similar tasks. A `main` function will always specify its command line and then deal with the actual arguments provided. A function that reads an image header will always call the routines that parse *keyword*=*value* statements.

To simplify the creation of new IPW source code, elaborate **templates** are provided. These templates are the source code for skeleton programs and functions, and illustrate a number of IPW coding practices:

- a recommended coding style;
- a recommended strategy for partitioning a primitive into modules;
- usage of some of the more complicated IPW functions.
- format of special comments that may be automatically converted to separate documentation.

3.3.4. Access to source code

Perhaps the most important programming aid provided by IPW is the accessibility of the source code. The source code is the ultimate reference for the behavior of any IPW component, and provides usage guidelines beyond those available from the template files. The source code is also the principal low-level documentation for IPW, since each source file contains special **header comments** which are extracted and reformatted by the `ipwman` command. This tends to ease the burden on the programmer of producing written documentation, since a usable form of documentation may be maintained in the code itself.

3.4. COMMENTARY

Two of the functional aspects of IPW are, so far as we know, unique. The first of these is the exclusive use of BIP interleaving for multiband images. This interleaving style is particularly suited to images with a very large number of bands, such as are obtained from imaging spectrometers. Few other image processing systems support BIP interleaving, and those that do must usually support other interleaving strategies as well (e.g., [LaVoie 1987].) IPW opts for the simplicity of a single format, at the possible cost of a slight increase in processing overhead for images with relatively few bands.

The other functionally unique aspect of IPW is its representation of floating-point pixel values as piecewise-linearly-quantized binary integers. Most other image processing systems either do not support floating-point pixel values at all, or do so using the host system's native (and nonportable) binary floating-point format. The IM Raster Toolkit [Paeth 1986a] is a notable exception in that it allows integer pixel values in the range $0..2^{nbits-1}$ to be optionally mapped into the floating-point range 0..1; however, this range may not be changed. In IPW, any monotonic mapping between integer and floating-point values is allowed, which allows enormous flexibility in specifying the range and precision of floating-point pixel values, while maintaining an efficient and portable storage format.

CHAPTER 4: THE USER LEVEL

This chapter contains a description of the user-level interface to IPW. We begin with a description of the IPW commands that function as "sources" and "sinks"; i.e., commands that prepare existing digital image datasets for processing by IPW, and commands (e.g., image display) that deal with the results of IPW processing operations. We then describe the basic IPW image processing commands; i.e., the primitives that embody generic image processing operations. Finally, examples of complex processing sequences are given, including some that would typically be embodied in scripts and thus appear to be "commands" themselves.

The two most fundamental IPW commands should be mentioned here. They are `ipw`, which displays a list of all available IPW commands, and `ipwman`, which displays the documentation for a particular IPW component (command, function, or file format). Entering the command `ipw` is a simple way to determine if your UNIX account is properly set up for IPW; if this command doesn't work, see §5.1. For information about any command listed in the output of `ipw`, type `ipwman` *command*.

The remainder of this chapter contains examples of the use of most IPW commands. Reference documentation for these commands is found in Chapter 5.

4.1. DATA MANAGEMENT

There are several IPW commands that do not perform image processing *per se*, but instead perform vital data management functions. These include:

- **ingestion**: converting existing digital images to a format that IPW understands;
- **ancillary data**: creating and modifying optional image headers;
- **format conversion**: creating multiband images, and incorporating non-image spatial data;
- **exporting:** converting IPW images to formats suitable for display or export to other processing environments.

Since these components are typically the first to be utilized by a new user of IPW, they will be described here, before the actual image processing commands.

4.1.1. Importing image data

Importing image data into IPW is a three-step process:

- express the basic geometry of the image as an IPW basic image header;
- express any ancillary data as an appropriate combination of IPW optional headers;
- convert the image pixel data to a format understood by IPW.

(Creation of optional headers is discussed in §4.1.2.)

An IPW basic image header (BIH) contains the following **per-image** (i.e., same for all bands) information:

- number of lines in the image
- number of samples per line
- number of bands per sample

The BIH also contains the following **per-band** information:

- number of bytes per pixel
- number of bits per pixel
- annotative text (optional)

An IPW basic image header is created by the `mkbih` command. By default, `mkbih` assumes a single-band image with 1-byte, 8-bit pixels, so a minimal BIH may be created by the command:

```
mkbih -l #lines -s #samples
```

As specified, this command will write a BIH to the standard output, and then attempt to copy the appropriate number of image data bytes from the standard input to the standard output. So, if a data file `raw_image` exists that contains nothing but the pixels for a 512 line by 512 sample image, then the corresponding IPW image would be created by:

```
mkbih -l 512 -s 512 <raw_image >ipw_image
```

Often it is inconvenient to supply the raw image data as the standard input for `mkbih`. For example, if an extremely large image is being read from tape, then forcing `mkbih` to copy it can be rather inefficient. Instead, the `-f` option can be used to tell `mkbih` to write a stand-alone header, to which the image data are later appended; e.g.:

```
mkbih -l 6000 -s 7000 -f >ipw_image
dd bs=7000 <tape >>ipw_image
```

What if an image to be imported into IPW contains its own header data? There are several ways to deal with this. If a fixed number of bytes of non-pixel data precede the image data, then they may be skipped using the `bs=` and `skip=` options with the UNIX `dd` command, and the result may either be piped to `mkbih`, or appended to an existing header as in the previous example. Similarly, non-pixel data following the last of the image data (i.e., "trailers", as opposed to headers) may be ignored by using the `bs=` and `count=` options with `dd`.

What about multiband images? If the raw image is already band-interleaved-by-pixel, then supplying the appropriate options to `mkbih` is sufficient. Unfortunately, the IPW BIP format is not common. Far more common are the band-interleaved-by-line (BIL) and band-sequential (BSQ) formats. In both of these cases, the strategy is to use IPW tools to break these images up into multiple single-band images.

A raw image in BSQ format is logically indexed as:

```
[band][line][sample]
```

which is another way of saying: all pixels for a given band are physically contiguous. The simplest way to deal with a BSQ image is:

- Append the raw data to a BIH which denotes 1 band, *#samples* samples per line, and *#lines*#bands* lines.
- Use the IPW command `window` (see §4.2.4.1) to extract each band from this image.

This example illustrates the ingestion of an `xim`[9] color image, with 512 lines, 512

[9] `xim` is a freely-distributable image display package for the X Window System.

samples per line, and 3 8-bit channels (note that the `dd` command is instructed to skip the 1024-byte `xim` header):

```
dd bs=1024 skip=1 <xim_file | mkbih -l 1536 -s 512 >xim.bsq
window -n 512,512           <xim.bsq >xim.0
window -n 512,512 -b  512,0 <xim.bsq >xim.1
window -n 512,512 -b 1024,0 <xim.bsq >xim.2
```

A raw image in BIL format is logically indexed as:

```
[line][band][sample]
```

which is another way of saying: all pixels for a given line are physically contiguous. A similar technique to that used for BSQ images may be employed; in this case, the IPW BIL image will be declared with 1 band, *#samples*#bands* samples per line, and *#lines* lines.

This example illustrates the ingestion of an AVIRIS[10] image, with 256 lines, 640 samples per line, and 192 bands (each band has 10-bit pixels):

```
mkbih -l 256 -s 122880 -i 10 -f >aviris.bil
dd bs=blocksize skip=header_blocks <tape >>aviris.bil
window -n 256,640           <aviris.bil >aviris.0
window -n 256,640 -b 0,640  <aviris.bil >aviris.1
window -n 256,640 -b 0,1280 <aviris.bil >aviris.2
...
```

Multiband IPW images may be assembled from multiple single-band images with the `mux` command, described in §4.1.3.

Finally, it should be mentioned that textual image data (i.e., pixel values expressed as printable text) can be converted to an IPW image with the support command `atob`. This command converts decimal integers expressed as printable text (e.g., ASCII) to binary integers. For example, a file of ten thousand integers with values between 0 and 255 generated by, say, a statistical package, could be converted to a 100 by 100 IPW image as follows:

```
atob -1 <text | mkbih -l 100 -s 100 >image
```

where the `-1` option indicates that `atob` should output 1-byte unsigned integers.

4.1.2. Optional headers

The BIH is the only IPW header which must always be present. All other headers are optional (although some particular commands may require them). The BIH is also the only header which contains per-image information; for all other headers, a separate instance of the header must be present for each band to which the header applies.

Analogous to `mkbih`, there are IPW commands provided for each optional header type, that will create one or more headers of that type and insert them into an IPW image. These commands take an existing IPW image as input and write it, with the newly created headers, to the standard output. The newly-created headers replace any existing headers of the same type in the specified bands.

[10] **A**irborne **V**isible and **I**nfra**R**ed **I**maging **S**pectrometer

The optional header creation commands all have names of the pattern mk*hdr*h, where *hdr* is the IPW name of the optional header. Some currently supported optional headers are:

name	**description**
geo	geographic location
lq	linear quantization parameters
sat	satellite parameters
sun	solar position
win	window coordinates

All of these commands accept a -b *band,...* option, specifying which bands the header is to be applied to (the default is all bands), and a -f option, which causes all input headers, and the newly-created headers, to be output, without any image data[11]; this allows the creation of a standalone BIH followed by optional headers, for use in an ingest scenario such as was described in the previous section.

As an example, the following adds a sat header to the Landsat Thematic Mapper data stored in image:

```
mksath -p Landsat-5 -s TM -l 045-034 -d 1988,2,15 image
```

This specifies a satellite header indicating the Landsat 5 platform, the Thematic Mapper sensor, a World Reference location of path 45, row 34, and an acquisition date of 15 February 1988. A copy of image with the new satellite header is written to the standard output.

4.1.3. Multiband images

The concept of a multiband image is central to IPW, and it is often necessary to merge or separate image data according to bands. Since the IPW BIP interleaving strategy is a form of ‘‘multiplexing’’, the IPW commands that manage the assembly or decomposition of multiband images are called mux and demux.

mux reads multiple input images and writes a single multiband output image, which has a number of bands equal to the sum of the number of input bands. The input images may each have an arbitrary number of bands, but they must all have the same number of lines and number of samples per line. As a simple example, the command:

```
mux red green blue >color
```

combines the images red, green, and blue into a single output image color.

The converse of mux is demux, which reads a multiband input image and writes an output image containing only selected input bands. To continue with the above example, the command:

```
demux -b 2 color >blue
```

would extract band 2 (0-relative) from color and write only that band into the output image blue.

[11] Note that this is not precisely the same behavior as the -f option to mkbih, since all of the optional header creation commands still require at least a BIH as input.

`mux` and `demux` are frequent sources and sinks for processing pipelines. For example, `mux` is often needed to assemble a multiband image for input to an IPW primitive that does some form of multivariate processing. `demux` is often needed to select a single channel for display. Since all IPW image headers are bound to a particular band, `mux` and `demux` can easily preserve any ancillary information.

4.1.4. Masks

Many IPW primitives accept an optional **mask** as input, either as the highest-numbered input image band, or as a separate, single-band image. In either case, processing of input pixels by the primitive is restricted to those samples which correspond to non-zero pixels in the mask. The principal use of masks is to define non-rectangular regions of interest.

For example, suppose that `dem` is a digital elevation model that includes a drainage basin for which elevation statistics are desired, and `basin` is an image (registered to `dem`) that contains nonzero values in every pixel located in the drainage basin. Then:

```
hist -m basin dem
```

would write a histogram of the elevations in the drainage basin to the standard output.

4.1.5. Point and vector data

While IPW is entirely a raster-oriented system, there are provisions for incorporating point and vector data into IPW images. Point data, expressed as printable text, may be inserted into an image with the `edimg` command. Vector data, also expressed as printable text, may be digitized by the `poly` command into points suitable for input to `edimg`.

A point datum is a single line of text with the format:

line sample value

where *line* and *sample* are intrinsic image coordinates, and *value* is the value of the pixel(s) at (*line, sample*). The `edimg` command reads two inputs, an image and a file of point data, and writes an output image that is a copy of the input image, except that each pixel with an associated point datum has that datum's value (i.e., the point data `replace` the input image data, where their coordinates coincide). For example:

```
edimg -i image
100 200 255
```

would write a copy of `image` to the standard output, but with the pixel(s) at line 100, sample 200 set to value 255. A typical use of `edimg` would be to superimpose field measurements on an image of the field study area.

Vectors represented by their endpoints may likewise be superimposed on images by pre-processing the endpoints with the `poly` command. `poly` reads point data (**without** *value*s) and writes the points and the digitized coordinates of the lines connecting them. If the points are the vertices of a closed polygon (i.e., the first vertex is also the last), then `poly` may also optionally **fill** the polygon; i.e., write the coordinates of all points interior to the polygon, in addition to the polygon's outline. In either case, the output of `poly` is fed as input to `edimg`.

For example, if the file `counties` contains the endpoints of vectors delineating county boundaries, then the following command sequence:

```
poly counties | edimg -i dem
```

will scribe those boundaries onto a digital elevation model `dem` and write the result to the standard output.

4.1.6. Exporting image data

Exporting image data from IPW involves converting the image data from the standard IPW format to a format expected by a particular display device or non-IPW software. IPW provides specific conversion programs for a few common formats, plus a generic procedure for converting image data into a portable format. In almost all cases, the exported image must be single band, so `demux` is a necessary tool when preparing images for export.

The IPW command `sunras` reads a single-band IPW image with either 1 or 8 bits per pixel, and writes a Sun "rasterfile" [Sun 1988], suitable for further processing by Sun software, or for display by the `rastool` command.

The `pspic` command reads a single-band IPW image with no more than 1 byte per pixel, and writes a PostScript [Adobe 1985] image, suitable for display on any PostScript output device (window system, laser printer, etc.)[12].

The `ipw2xim` command reads a single-band IPW image with no more than 1 byte per pixel, and writes an `xim`-format image. The output image may be displayed on any device running an X window system server [Scheifler 1986], by the `xxim` client program.

In general, most non-IPW image processing software can handle single-band IPW image data (perhaps with bytes swapped), but cannot interpret IPW headers. Therefore, the headers should be stripped from the exported image with the `rmhdr` command. This command reads an IPW image and writes only the image data (therefore, its output is **not** a valid IPW image). If the exported image is being written to tape, it is customary to make the block size equal to the number of bytes per image line, as a convenience for some non-UNIX systems where the physical attributes of a file affect its logical structure. For example, the following command sequence writes a 3-band IPW image (640 samples per line, 1 band, 1 byte per pixel) to tape as a sequence of 3 headerless files:

```
demux -b 0 image | rmhdr | dd obs=640 >tape
demux -b 1 image | rmhdr | dd obs=640 >tape
demux -b 2 image | rmhdr | dd obs=640 >tape
```

where *tape* is a "no-rewind-on-close" tape device.

For situations where image data must be exported in a textual form (e.g., for input to a statistical package), the command:

```
primg -a -i image
```

may be used to print the value of every pixel in `image` on the standard output as

[12] For example, an 8-bit PostScript image is automatically halftoned when displayed on a monochrome PostScript laser printer.

readable text, one sample per line, with as many values per line as there are bands in the input image.

4.2. GENERIC IMAGE PROCESSING OPERATIONS

The IPW image processing primitives are designed around the notion of **generic** image processing operations. Broad classes of image processing operations have been identified, and IPW primitives have been built to provide the most general-purpose coverage of these operations.

Four classes of image-based operations may be identified [Castleman 1979]:

- **univariate point operations**, whose contexts are single pixels;
- **multivariate point operations**, whose contexts are two or more pixels within a sample;
- **neighborhood operations**, whose contexts are pixels within contiguous samples;
- **geometric operations**, which copy pixel values from one sample to another.

4.2.1. Univariate point operations

In univariate point operations, pixel values in the output image are solely a function of the corresponding pixel values in the input image:

$$OUT_{line,sample,band} = f\,(IN_{line,sample,band})$$

The function f may be definable *a priori* (e.g., logarithm, exponential, scaling by a constant, etc.); or it may involve computing image statistics (e.g., normalization using image mean and standard deviation); or it may be hardware-dependent (e.g., pseudocolor mapping of a single-channel image).

Since only a single input pixel is required to generate an output pixel, univariate point operations do not require random access to the input image. This meshes well with the IPW's model of image processing operations communicating by pipes.

4.2.1.1. Lookup tables

In IPW, the generic univariate point operation is implemented by the `lutx` primitive. `lutx` accepts two inputs, an image and a **lookup table**. The pixels of the input image are mapped through the lookup table, and the mapped values are written to the standard output.

The lookup table input to `lutx` is actually an IPW image, with the following special properties:

- exactly 1 line;
- at least as many bands as the input image;
- 2^{nbits} samples per line, where *nbits* is the maximum number of bits per pixel in the input image.

An input image pixel value is used as a sample coordinate in the lookup table, and the value of the corresponding pixel for the same band becomes the output value:

$$OUT_{line,sample,band} = LUT_{0,IN_{line,sample,band},band}$$

Since lookup tables are images, they have (at least) a basic image header, followed by binary integer pixel data. Therefore, the simplest way to create a lookup table is to create a basic image header and append binary data to it. For example, assume the file `lut_text` contains 256 printable integers, one per line, with values between 0 and 255. Then the command:

```
atob -1 lut_text | mkbih -l 1 -s 256
```

will write a single-band, 256 element lookup table to the standard output.

Creating lookup tables from text files is common enough that IPW provides tools to help simplify the procedure. The command

mklut -i *inbits* -o *outbits* -k *bkgd*

creates a single-band lookup table capable of mapping *inbits*-bits input pixels into *outbits*-bits output pixels. All output values are initialized to *bkgd* (which of course must be between 0 and $2^{outbits}-1$). `mklut` then reads pairs of text integers, one per line, from the standard input:

in out
in out
...

and sets each lookup table sample *in* to the value *out*. For example, the following command sequence:

```
mklut -i 8 -o 8 -k 0
100 255
```

creates a lookup table that will map input pixels with value 100 into value 255, and map all other input pixel values to 0.

The command `interp` is useful in conjunction with `mklut` when a range of input values is to be linearly mapped into a range of output values. `interp`, like `mklut`, read pairs of text integers from the standard input. `interp`'s standard output is the input pairs, plus any intervening input values and the corresponding, linearly interpolated output values. For example, the command sequence:

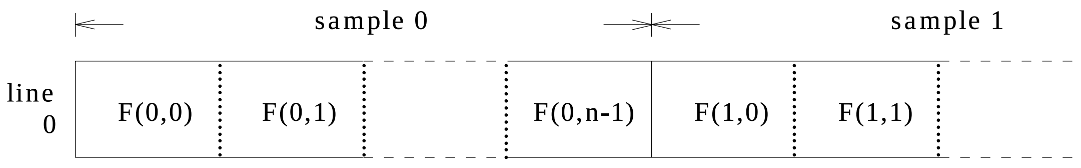
```
interp | mklut -i 12 -o 8
0 0
4095 255
```

creates a lookup table capable of converting 12-bit input pixels to 8-bit output pixels, using linear scaling.

4.2.1.2. Histograms

There is a subset of univariate point operations, which might be called ‘‘univariate summary operations’’, in which the output is not an image, but a **summary** of some univariate property of the input image. The most important of these summary operations is the computation of a **histogram** of the image's pixel values.

An IPW histogram is represented externally as an IPW image, with the same properties as a lookup table. For a given band, an input image pixel value is used as a sample coordinate in the histogram, and the value of that histogram pixel is the frequency of the input pixel value (i.e., the number of times the pixel value occurred in the input image):



where $F(i,j)$ is the frequency of pixel value i in input band j.

Some IPW commands use the histogram as an intermediate step in a complex calculation. For example, a common image enhancement processing sequence is:

```
hist image | histeq | lutx -i image
```

`hist` calculates the histogram of *image*; `histeq` converts the histogram to a lookup table; and `lutx` applies the lookup table to *image*. The overall purpose of this pipeline is to create an output image with a flat or "equalized" histogram [Pratt 1978]; i.e., an output image in which every possible pixel value has an equal probability of occurring.

Often there is considerable information for an analyst in a graphical display of the image histogram. The IPW command `grhist` converts an IPW histogram to a form suitable for display by the UNIX `plot` command:

```
hist image | grhist | plot
```

The histogram will be displayed as bar graph with labeled axes. The format of the display may be controlled by options to `grhist`, which accepts the same arguments as the UNIX `graph` command[13].

4.2.2. Multivariate point operations

Multivariate point operations combine multiple coregistered input pixels into a single output pixel. There are two ways to perform these operations. One is to combine multiple input images:

$$OUT_{line,sample,band} = IN^0{}_{line,sample,band} \text{ op } \cdots \text{ op } IN^{N-1}{}_{line,sample,band}$$

The other strategy, and the one adopted by IPW, is to combine the bands of a single input image:

$$OUT_{line,sample,0} = IN_{line,sample,0} \text{ op } \cdots \text{ op } IN_{line,sample,N-1}$$

The band-combination method allows all input for a multivariate point operation to be provided on a single input stream, and thus allows more flexibility when incorporating a multivariate point operation into a pipeline.

Note that multivariate point operations usually cannot be implemented as lookup tables, since the cardinality of the table would increase exponentially with the number of input pixels required to generate an output pixel.

While multivariate point operations do not require random access to the input image, they may require an entire input sample to generate an output pixel[14].

---

[13] `grhist` is just a shell script "wrapper" that invokes the `graph` command.

[14] This can be a non-trivial amount of data for imaging spectrometers with several hundred bands.

4.2.2.1. Algebraic operations

The most important multivariate point operations are those that yield algebraic combinations of their input values. IPW provides three primitives implementing these combinations.

- `bitcom` performs bitwise logical operations on pixels;
- `lincom` computes a weighted sum of pixels;
- `mult` computes a pixel product.

Each pixel output from `bitcom` is a bitwise logical combination (AND, inclusive-OR, exclusive-OR) of the corresponding input pixels. A typical application would be to ‘‘black out’’ the portions of an image outside a region of interest, by AND-ing the image with a mask of the region:

```
mux image region | bitcom -a -m
```

Here the `-m` option indicates that the final input band is a mask, as described in §4.1.4.

Each pixel output from `lincom` is a linear combination of the corresponding input pixels.

$$OUT_{line,sample,0} = \sum_{i=0}^{N-1} C_i IN_{line,sample,i}$$

where C_i is the user-specified weight for band i. The default weights are $\frac{1}{N}$, so just:

```
lincom
```

by itself outputs an average of all input bands.

Negative weights cause subtraction instead of addition. For example:

```
lincom -c 1,-1
```

subtracts the second input band from the first.

Three bands representing red, green, and blue may be combined into a single ‘‘black-and-white’’ image by:

```
lincom -c 0.299,0.587,0.114
```

which weights each color according to its contribution to NTSC luminance [Pratt 1978].

Each pixel output from `mult` is the product of the corresponding input pixels. Optionally, pixels in selected input bands may be replaced by their reciprocals before the multiplication is performed; this allows `mult` to do division as well as multiplication. For example:

```
mux num denom | mult -r 1
```

computes the ratio of the single-band images `num` and `denom`.

4.2.2.2. Statistics

The primitive `mstats` computes the basic multivariate statistics of a multichannel input image, and writes the statistics as text on the standard output. The values computed include the mean and variance of each input band, and the covariance of each pair of input bands.

The output from `mstats` is useful in a variety of contexts. For example, it is fairly straightforward to compute the eigenvectors of the variance-covariance matrix [Press 1988], which can then be used as `lincom` coefficients to produce principal component images [Green 1976].

`mstats` accepts an optional second input image which indicates **classes** for which separate sets of statistics should be computed. The value of each pixel in the class image is the class to which the corresponding input sample should be assigned. A separate set of output statistics will be produced for each distinct class. (If a class image is not supplied, then all samples in the input image are assumed to be in class 0.) For example, the classes could be associated with the training sites for a Bayesian classifier [Richards 1986], which would be driven by the statistics output from `mstats`.

4.2.2.3. Specialized operations

The primitives described so far are sufficient to implement most multivariate point operations, but not always with acceptable efficiency. For example, a problem that frequently arises in surface climate studies is the computation of the cosine of the local solar zenith angle, which is used to weight calculations of the solar beam irradiance. Given the slope S and azimuth A of the local gradient (see §4.2.3.2), the cosine of the solar zenith angle θ_s is given by [Sellers 1965]:

$$\cos\theta_s = \cos\theta_0 \cos S + \sin\theta_0 \sin S \cos(\phi_0 - A)$$

where θ_0 and ϕ_0 are the solar zenith and azimuth relative to a horizontal surface.

This calculation could be performed by a sequence of `lincom`s, `mult`s, and `lutx`s (to evaluate the trigonometric functions), but it would be slow, and would involve the creation of several temporary files. Since this calculation is performed often enough for its speed to be a consideration, it has been embodied in its own primitive `shade`, which accepts a 2-band gradient image as its input, and writes an image of $\cos\theta_s$ values as its output.

4.2.3. Neighborhood operations

Neighborhood operations combine multiple contiguous input pixels into a single output pixel:

$$OUT_{line,sample,band} = F(IN_{L_1,S_1,band}, \ldots, IN_{L_U,S_V,band})$$

where $L_1 \leq line \leq L_U$ and $S_1 \leq sample \leq S_V$ (i.e., the input neighborhood is U lines by V samples). The neighborhood is also called a window or a **kernel**.

Whereas point operations can be implemented using simple sequential image access, neighborhood operations must maintain more context in the input image. Given the external geometry of an IPW image, the minimum necessary context is almost always U lines of the input image, over which the U by V kernel is passed. This still allows reasonable throughput in a pipeline, since the neighborhoods may be processed in output pixel order, and the output pixels can be written as soon as they are calculated.

4.2.3.1. Convolution

Convolution, or spatial filtering, is a neighborhood operation that computes an output pixel value from a weighted sum of the input neighborhood pixels:

$$OUT_{line,sample,band} = \sum_{i=line-\frac{U}{2}}^{line+\frac{U}{2}} \sum_{j=sample-\frac{V}{2}}^{sample+\frac{V}{2}} W_{i,j} IN_{i,j,B}$$

Convolution is implemented by the IPW primitive `convolve`, which reads an input image and a text file of weights, and writes a filtered output image. The weights are specified as a kernel size, followed by the weights for each pixel in the kernel. For example:

```
convolve -i image
3 3
1 1 1
1 1 1
1 1 1
```

writes an image each of whose pixels is the sum of the neighboring 9 pixels in the input image. The general format of the weights file is:

$$\begin{array}{lll} U \;\; V & & \\ W_{0,0} & \cdots & W_{0,V-1} \\ \cdots & \cdots & \cdots \\ W_{U-1,0} & \cdots & W_{U-1,V-1} \end{array}$$

Some other useful kernels are Sobel operators, for directional edge detection: [Duda 1973]:

```
3 3                   3 3
-1 0 -1               -1 -2 -1
-2 0 -2                0  0  0
-1 0 -1                1  2  1
```

and the digital Laplacian, for edge enhancement:

```
3 3
 0 -1  0
-1  5 -1
 0 -1  0
```

(This kernel actually represents an equal weighting of the original image and the Laplacian, which tends to produce a visually pleasing ‘‘sharpening’’ of the image.)

4.2.3.2. Gradient

A special neighborhood operator that is used frequently enough to warrant its own primitive is the **gradient**. The gradient of a surface is a vector quantity. Since the gradient is used most often in calculations involving digital elevation models (DEMs), we use the terms **slope** (S) for the magnitude of the gradient, and **azimuth** (A) for its direction. These components are defined as [Dozier 1989]:

$$\tan S = \left[(\partial Z/\partial x)^2 + (\partial Z/\partial y)^2 \right]^{1/2}$$

$$\tan A = \frac{-\partial Z / \partial y}{-\partial Z / \partial x}$$

Z is elevation. The partial derivatives are computed as:

$$\frac{\partial Z}{\partial x} = \frac{Z_{line,\,sample+1} - Z_{line,\,sample-1}}{2\Delta h}$$

$$\frac{\partial Z}{\partial y} = \frac{Z_{line+1,\,sample} - Z_{line-1,\,sample}}{2\Delta h}$$

where Δh is the line and sample spacing.

As with `shade`, a combination of `convolve`s, `lutx`s, and `lincom`s could perform the gradient calculation, but with an unacceptable overhead. Hence the primitive `gradient` implements these calculations directly. `gradient` reads a DEM image and write a 2-band image (slope and azimuth). The integrated implementation allows for such optimizations as storing slopes as quantized sines; this increases the precision of representation of shallow slopes, which are most common in natural terrain.

4.2.4. Geometric operations

Geometric operations change the locations of samples in the output image, with respect to the input image:

$$OUT_{i,j,band} = IN_{u,v,band}$$

Geometric operations are defined by the **mapping function** that matches output to input image coordinates:

$$u = F_U(i,j)$$
$$v = F_V(i,j)$$

There are two broad classes of geometric operations, based on the relative complexity of the mapping functions. The most complicated operations are those in which the functions F:

- are independent;
- may yield non-integral values for i and/or j.

This class of operations is often called **warping**. Implementing these operations requires:

- exhaustive evaluation of F;
- **resampling** of IN, to interpolate pixel values at non-integral coordinates.

These function were implemented in IPW's predecessor QDIPS [Frew 1984], but have not yet been reimplemented in IPW.

The simpler class of geometric operations are implemented by a variety of IPW primitives, which in combination can produce many of the effects that would normally require warping, at a considerable savings in complexity and execution time.

4.2.4.1. Subscene extraction

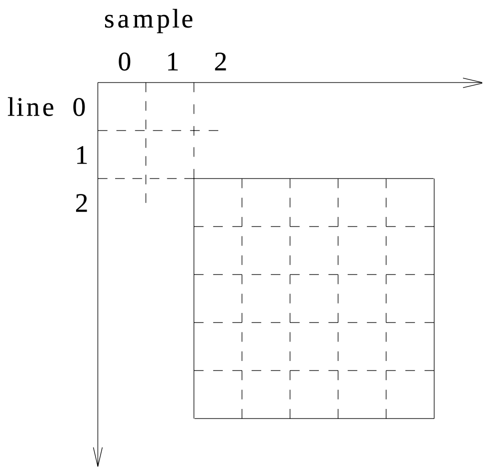
The simplest geometric operation involves creating an output image that is a rectangular subset of an input image. This is accomplished by the `window` primitive. `window` accepts a variety of specifications of the subscene to be extracted:

- beginning pixel
- end pixel
- center pixel
- number of pixels

For example:

```
window -b 2,2 -n 5,5
```

extracts a 5 line by 5 sample subscene from the input image, beginning at line 2, sample 2.



`window` also allows the subscene to be specified in extrinsic coordinates, if the input image has a `geo` or `win` header.

4.2.4.2. Reflection

Another useful geometric operation involves reflecting an image about either its horizontal or vertical axes; i.e., reversing its line and/or sample ordering. This is implemented by the IPW `flip` primitive. For example:

```
flip -s
```

produces a mirror of the input image, by reversing the order of sample in each line, while:

```
flip -l -s
```

also reverses the order of the lines in the image, effecting a 180 degree rotation of the input image.

4.2.4.3. Transposition

Transposing an image involves exchanging the order of its lines and sample:

$$OUT_{i,j,band} = IN_{j,i,band}$$

In IPW this is implemented by the `transpose` command. This implementation is quite fast: the output image is kept in memory, and each line of the input image is scattered into the output image as it is read.

`transpose` is seldom used as an end in itself, but it is an indispensable component of many processing pipelines. For example, the combination of `flip` and `transpose` may be used to effect right-angle rotations. To rotate an image 90° clockwise:

```
flip -l | transpose
```

To rotate an image 90° counterclockwise:

```
flip -s | transpose
```

4.2.4.4. Zooming

Although magnifying or minifying an image by arbitrary factors necessitates resampling, a reasonable approximation may be achieved by replicating or skipping image lines and/or samples. This ‘‘integral zooming’’ is implemented by the IPW `zoom` primitive.

A simple example would be:

```
zoom -s 2 -l 2
```

which replicates each input sample and line, effecting a 2x zoom. (The line and sample factors need not be identical.)

Zooming by non-integer factors can be simulated, if the factor is rationally expressible. In this case, a pipeline of two `zoom`s is used: the first magnifies by the numerator of the zoom factor, and the second minifies by the denominator of the zoom factor. For example, suppose an image must be shrunk in the sample dimension only by a factor of 3/4, for display on a monitor with a 4:3 aspect ratio:

```
zoom -s 3 | zoom -s -4
```

(A negative zoom factor indicates minification.)

4.2.4.5. Shearing

The primitive `skew` shears an image horizontally; i.e., it horizontally displaces the origins of the input image lines. The amount of shearing is specified as the angle from the left edge of the output image to the left edge of the input image. For example:

```
skew -a 30
```

shears the input image 30 degrees clockwise (negative angles are counterclockwise).

`skew`, like `transpose`, is most useful as component in a pipeline. For example, combinations of `skew`, `flip`, and `transpose` may be used to produce an output image whose lines are oriented at an arbitrary angle with respect to the input image. This allows scanline-type algorithms to be applied in an arbitrary direction across an image.

4.3. EXAMPLE APPLICATIONS

We close this chapter with some illustrations of how the IPW primitives may be combined to produce complex processing sequences. It should be remembered that any processing sequence may be preserved in a shell script and may thus also appear to be a ‘‘primitive’’ operation to an IPW user.

In [Paeth 1986b] a scheme is outlined whereby an image may be rotated through an arbitrary angle using sequences of shearings, reflections, and transpositions. The following is an example of this method implemented as a pipeline of IPW primitives.

```
skew -h -a {θ/2} |
        flip -l |
        transpose |
        skew -h -a {tan⁻¹sinθ} |
        flip -s |
        transpose |
        skew -h -a {θ/2}
```

The output of this pipeline is the input image rotated θ degrees clockwise. Note the `-h` option to `skew`, to override any existing skew header in the input image.

A **parallelepiped classifier** [Richards 1986] is perhaps the simplest mechanism for classifying a multiband image. Each class is characterized by a minimum and maximum pixel value in each band[15]. If the number of classes is limited to the number of bits in a pixel, then a straightforward implementation is possible in IPW. First, a lookup table is constructed for each input channel, mapping the pixel values in that band into a class code, which must be a power of 2:

```
interp | mklut -i inbits -o nclasses > class_lut.i
class_1_min 1
class_1_max 1
class_1_max+1 0
class_2_min-1 0
class_2_min 2
class_2_max 2
...
```

Note that if *class_1_max* and *class_2_min* differ by more than 1, then the intervening values must be explicitly set to 0. This process is repeated for each input band. Then:

```
mux class_lut.* | lutx -i image | bitcom -o
```

`lutx` converts each band to the classes in that band. `bitcom` ORs the classes together into a single output class. Samples that have the same class in each band will have a single bit, the class code, set in the corresponding output pixel. Samples that could not be unambiguously classified will have more than one bit set in the corresponding output pixel.

4.4. COMMENTARY

There are many aspects of the IPW user level that contrast sharply with other image processing environments. The most general is the decoupling from a display device that is a natural consequence of IPW's pipes-and-filters user interface. In IPW, a display is no more than a "sink" at the end of a processing pipeline; the program at the end of the pipeline containing the only display-dependent code. By contrast, many display-based image processing systems (e.g., [Adams 1979]) require the presence of the

[15] The "parallelepiped" is the N-dimensional (N = number of bands) object whose vertices are these minima and maxima.

display device and its associated hardware to perform any image processing operations, even those whose results need not necessarily be displayed.

In many image processing systems (e.g., [LaVoie 1987]), selection of a subscene for processing, in either the spatial or spectral dimensions, is provided by all of the operators in the system (e.g., as part of the syntax of the command language). IPW isolates these functions into separate commands, which results in a conceptually much simpler system.

The ability of many IPW commands to treat ordinary images as processing masks is likewise unusual.

The presence of commands for dealing with point and vector data in IPW reflect IPW's origin in investigations where remote sensing data are frequently combined with point field measurements. The simplicity of these commands also reflects IPW's design goals. Basically, provisions are made for inserting and extracting point data into and from an image; all operations beyond these take place either on IPW images, or outside IPW using the textual representation of the point data. The exception to this is IPW's sole provision for vector or polygonal data - the `poly` primitive merely interpolates point data into the image pixels lying along a line segment or within a polygon.

The ability to control the precision of pixel representation at the binary level allows IPW to store both lookup table and histogram data as if they were single-line images, thus simplifying I/O on these common data structures, and also allowing them to be processed by the full range of IPW operations.

Certain generic operations are implemented in unique ways by IPW. Univariate point operations are lookup-table driven, requiring only the `lutx` command to implement them; this is in contrast to many other systems where each such operation is implemented by a separate command. Multivariate operations usually operate by combining bands of a single input image, rather than by combining multiple input images; this allows multivariate operations to be inserted into a processing pipeline.

CHAPTER 5: USER'S MANUAL

This chapter contains terse but complete references for all IPW commands normally used by a non-programmer. These commands constitute the primary user interface to IPW.

5.1. INTRODUCTION AND INITIALIZATION

Before accessing any IPW commands, the following elements of a user's UNIX environments must be appropriately initialized:

- IPW-specific environment variables
- command search path

IPW is normally installed in its own directory[16]. The full pathname of this directory must be accessible in the user's environment variable `IPW`. In addition, some IPW programs will create scratch files in the directory whose full pathname is in the environment variable `TMPDIR`. If this variable is not defined, the scratch files may be placed in a file system too small to accommodate large image files.

The directories containing the IPW commands must be included in the user's command search path (the `PATH` environment variable). These directories always include `$IPW/bin` and `$IPW/etc`, and may include additional site-specific directories.

At most IPW sites, the system administrator will have provided a file `$IPW/pub/cshrc` that correctly initializes the environment variables and search path for `csh` users[17]. IPW users need therefore only add the single line:

```
    source ~ipw/pub/cshrc
```

to their `~/.cshrc` files, and the proper IPW environment will be automatically initialized each time they log in.

Once the environment is initialized for IPW, the user accesses IPW commands simply by typing their names.

5.2. COMMANDS

This section contains copies of the on-line documentation for each IPW command used by non-programmers. The on-line documentation is stored as **header comments** in the corresponding IPW source files, and is displayed on an IPW user's terminal by the `ipwman` command. These constraints preclude the use of multiple fonts or non-ASCII characters, so the documentation is reproduced here in a single constant-width (‘‘typewriter’’) font.

In addition to standard format for program header comments[18], the following notational conventions are observed:

(*val1*, *val2*)
: Two values separated by a comma and enclosed in parentheses are an ordered pair, usually indicating a particular sample location in an image.

---

[16] See Chapter 8.

[17] Some IPW sites may also provide `shrc` and `kshrc` files for `sh` and `ksh` users.

[18] See §A.1.1.1.

"*text*"
Any *text* enclosed in double-quotes is literal; i.e., should be used exactly as shown.

`-` The `-` (minus sign), used as a file name, always indicates the standard input.

`*` An asterisk embedded in a file or command name is wild card, matching any number of characters (e.g., `mk*h` would match all IPW commands whose names begin with `mk` and end with `h`.)

NAME

bitcom -- bitwise band combination

SYNOPSIS

```
bitcom -{aox} [-m] [image]
```

DESCRIPTION

bitcom reads a multi-band image from {image} (default: standard input), and writes a single-band image to the standard output. The output image is a bitwise combination of all input bands.

OPTIONS

-a Compute bitwise AND of input bands.

-o Compute bitwise inclusive-OR of input bands.

-x Compute bitwise exclusive-OR of input bands.

Exactly one of -a, -o, or -x must be specified.

-m The last (highest-numbered) input band is used as a mask. Pixels in this band will first be truncated or padded to the same number of bits as the other input bands, then any nonzero pixels will be set to ˜0 (all bits on).

DIAGNOSTICS

single-band input image

The input image must have at least 2 bands.

different # bits / pixel: bands 0, {band}

All input bands must have the same number of bits per pixel (except the last band, if -m is specified).

EXAMPLES

To mask a DEM "elev" such that pixels outside the drainage basin "basin" are set to 0:

```
mux elev basin | bitcom -a -m
```

SEE ALSO

IPW: mux

CMPIMG **CMPIMG**

NAME

cmpimg -- compare two images

SYNOPSIS

```
cmpimg image1 image2
```

DESCRIPTION

cmpimg compares the pixel data from {image1} and {image2}, ignoring any headers. A message is printed on the standard output indicating whether the two input images differ or are identical.

DIAGNOSTICS

```
Images "{image1}", "{image2}" are identical

Images "{image1}", "{image1}" differ
```

FILES

$TMPDIR/cmpimg{NNNNN}

Temporary storage for a headerless copy of {image1}.

EXAMPLES

The following sequence might be used to test a command, such as transpose, that should be its own inverse:

```
transpose image | transpose | cmpimg - image
```

SEE ALSO

IPW: rmhdr

UNIX: cmp

NOTES

cmpimg is currently implemented as a shell script that invokes the UNIX cmp command. This has the following consequences:

- Since cmp has no notion of image structure, cmpimg gives no indication of the logical image locations (line, sample, band) where differences occur.

- Some implementations of cmp do not allow "-" (standard input) to be specified for {image2}.

The exit status of cmpimg does NOT indicate the result of the comparison.

NAME

cnhist -- convert IPW histogram to cumulative normalized text representation

SYNOPSIS

```
cnhist [begin [end]]
```

DESCRIPTION

cnhist reads an IPW histogram from the standard input and writes a cumulative normalized histogram as text on the standard output.

OPTIONS

begin Begin processing at the input value with this index number (default: 0). The first {begin} output values will be 0.

end Cease processing after the input value with this index number (default: last input value). The last {last - end} output values will be 1.

EXAMPLES

To view a cumulative histogram of "image":

```
hist image | cnhist | graph -a | plot
```

SEE ALSO

IPW: btoa, hist, histeq, rmhdr

UNIX: awk

NOTES

cnhist is currently implemented as a shell script that invokes the $AWK command to perform the normalization calculations.

The option syntax is nonstandard.

```
NAME
        convolve -- image convolution

SYNOPSIS
        convolve [-i image] [-c coefs]

DESCRIPTION
        convolve convolves an input image with a user-supplied
        kernel, writing the result to the standard output.

OPTIONS
        -i      Read image data from {image} (default: standard
                input).

        -c      Read kernel coefficients from {coefs} (default:
                standard input).  This is a text file, with the
                following format:

                        value           description
                        -----           -----------
                        1               #rows in kernel
                        2               #columns in kernel
                        3               coefficient[0][0]
                        ...             ...
                        #cols+2         coefficient[0][#cols-1]
                        #cols+3         coefficient[1][0]
                        ...             ...

                Values may be separated by any sequence of blanks,
                tabs, and newlines.  If the kernel does not sum to
                zero, it is normalized to sum to 1.0.

        At least one of -i and/or -c must be specified.
```

DIAGNOSTICS

{rows}x{cols} kernel is bigger than {nlines}x{nsamps} image

> The kernel cannot be bigger than the image.

sorry, only single-band input images accepted

> The input image may not have more than 1 band.

bad kernel size: {rows}x{cols}

> The kernel dimensions must be nonzero positive integers.

both kernel dimensions must be odd

EXAMPLES

To sharpen an image by adding its Laplacian, reading the kernel from the standard input:

```
convolve -i image
3 3
0 -1 0
-1 5 -1
0 -1 0
```

NOTES

There must be enough virtual memory to accommodate {rows} output image lines.

The normalization process can cause kernels that differ only slightly to produce radically different results.

```
NAME
        demux -- demultiplex (extract bands from) IPW image

SYNOPSIS
        demux -b band[,...] [image]

DESCRIPTION
        demux copies the specified bands from {image} (default:
        standard input) to the standard output.

OPTIONS
        -b      Extract the specified bands.  The bands will appear
                in the output image in the order specified; e.g., if
                the input image has 2 bands, then

                        demux -b 1,0

                outputs the same image but with the band order
                reversed.

DIAGNOSTICS
        {band}: bad input band number

                The input image does not contain the specified
                band.

        "{header}" header, band {band}: no such band

                The specified {header} in the input image pertains
                to a nonexistent band (i.e., the input image is
                corrupted).

EXAMPLES
                demux -b 0,1,2 image

        creates a 3-band image consisting of the first 3 bands from
        "image".

SEE ALSO
        IPW: mux
```

NAME

dither -- create bilevel image using ordered dithering

SYNOPSIS

```
dither [-r rank] [image]
```

DESCRIPTION

dither reads multi-bit pixels from {image} (default: standard input), converts them to single-bit pixels using ordered dithering, and writes the result to the standard output. Black (i.e. low-value) output pixels are assigned the value 0; white (i.e. high-value) output pixels are assigned the value 1.

OPTIONS

-r	Use a dither matrix of rank {rank} (default 4). Possible values are 4, 8, or 16, for simulating 16, 64, or 256 gray levels, respectively.

EXAMPLES

To display "image" on a monochrome Sun workstation under SunView:

```
        dither image | sunras | rastool
```

SEE ALSO

IPW: rastool, sunras

David F. Rogers, "Procedural Elements for Computer Graphics", McGraw-Hill, 1985, p. 107.

NOTES

Before using dither, make sure that the output device (e.g., PostScript printer) or display software (e.g., X) does not already incorporate a better halftoning algorithm.

NAME

edhdr -- edit image header

SYNOPSIS

```
edhdr image ...
```

DESCRIPTION

edhdr allows interactive editing of {image}’s headers. The headers are copied into a scratch text file and the editor specified in the EDITOR environment variable (default: vi) is invoked. After the editor exits, the image is reconstructed with the new headers.

DIAGNOSTICS

standard input and output must be a terminal

The default editor (vi) must be connected to a terminal device.

image not writable

The input image is also the output image, so it must be writable.

FILES

{image}.BAK	temporary copy of {image}
edhdr{NNNNN}	temporary copy of edited header

SEE ALSO

IPW: $IPW/h/*h.h

UNIX: vi (or, documentation for your $EDITOR)

NOTES

THERE IS NO SANITY CHECKING OF THE EDITED HEADER. DON’T USE THIS PROGRAM UNLESS YOU KNOW WHAT YOU ARE DOING.

The directory in which edhdr is invoked must have enough free space for a copy of each input image.

EDIMG **EDIMG**

```
NAME
        edimg -- replace image pixels

SYNOPSIS
        edimg [-r] [-k const] [-c coords] [-i image]

DESCRIPTION
        edimg copies the input {image} to the standard output,
        replacing specified pixel values.  Replacement values are
        read from the text file {coords}.  Each value is represented
        as a single line with the format:

                line sample constant

        indicating that the value of the pixel at location
        {line},{sample} is to be replaced with {constant}.  If
        {constant} is missing, a default value is used.

OPTIONS
        -r      Use quantized (raw) pixel values:  don’t convert
                input pixels to floating point (i.e. disregard any
                input "lq" headers), and treat replacement constants
                as quantized values.

        -k      Use {const} as the default replacement value
                (default: 0)

        -c      Read replacement values from {coords} (default:
                standard input).

        -i      Read image data from {image} (default: standard
                input).

        At least one of -c and/or -i must be specified.

DIAGNOSTICS
        invalid coordinates: {text}

                The {text} line is not a valid replacement value
                specification.

        unsorted coordinates: {line},{sample}

                The coordinates must be sorted in order of
                increasing line numbers.
```

EXAMPLES

To set the pixel(s) at (100,200) in "image" to 255:

```
edimg -r -i image
100 200 255
```

SEE ALSO

IPW: poly

UNIX: sort

NOTES

If the input image has more than one band, then ALL bands at a specified location are set to the corresponding replacement value.

The input {line} and {sample} values are always intrinsic (raw); that is, unaffected by any coordinate system headers ("geo", "win", etc.) in the input image.

NAME

flip -- flip IPW image

SYNOPSIS

```
flip [-l] [-s] [image]
```

DESCRIPTION

flip copies the input {image} (default: standard input) to the standard output, reversing the order of the lines and/or samples.

OPTIONS

-l	Reverse order of image lines.
-s	Reverse order of samples in each line.

At least 1 of -l and/or -s must be specified.

EXAMPLES

An image may rotated in multiples of 90 degrees as follows:

```
flip -s | transpose     #  90 degrees clockwise
flip -l -s              # 180
flip -l | transpose     # 270
```

SEE ALSO

IPW: transpose

NOTES

If -l is specified then the input image must fit into virtual memory.

If the output image does not have the canonical IPW line (top to bottom) and sample (left to right) order, then it will be given an orientation ("or") header describing the non-standard ordering.

NAME

gradient -- compute slope and aspect

SYNOPSIS

```
gradient [-s] [-a] [-i sbits[,abits]] [-d dl[,ds]] [image]
```

DESCRIPTION

gradient computes the slope and aspect (i.e. the magnitude and direction of the gradient) of each pixel in the input {image} (default: standard input).

The 2-band output image has slope as its first band and aspect as its second. Slopes are stored as sin(slope), quantized over [0..1]. Aspects are stored as radians from south, quantized over [-pi..pi), with negative values to the west and positive values to the east.

OPTIONS

-s Compute slopes only (default: slopes and aspects).

-a Compute aspects only (default: slopes and aspects).

-d Set input grid spacing to {dl} (default: obtained from {image}'s "geo" header, or set to 1 if otherwise unavailable). If {ds} is specified, use {dl} for the input line spacing and {ds} for the input sample spacing.

-i Use {sbits} bits per output pixel (default: 8). If {abits} is specified, use {sbits} bits per slope pixel and {abits} bits per aspect pixel.

DIAGNOSTICS

```
# bits must be >= 1
-d {delta},{delta} : must be positive

        The arguments to the -d and -i options must be
        positive nonzero integers.

input file has {nbands} bands

        The input image must have only 1 band.

Elevation file has no GEOH, spacing set to 1.0
input file has no LQH; raw values used
input units "{units}", should be "m"

        These deficiences in the input image will introduce
        linear errors into the slope calculations.

Elevation file should be standard orientation

        The output azimuth values will have a systematic
        bias corresponding to the non-standard orientation
        of the input image.

spacing in geodetic header ignored

        The -d option overrides any pixel spacing
        information in the input image.
```

SEE ALSO

IPW: demux, mkgeoh, mklqh

NOTES

gradient is optimized for terrain calculations (e.g., storing slopes as sines offers increased precision for shallow slopes, which are more common in nature), and may therefore be less than ideal as a generic image derivative program.

NAME

grhist -- graph an IPW histogram

SYNOPSIS

```
grhist [graph-options]
```

DESCRIPTION

grhist reads an IPW histogram from the standard input, and writes a graphic rendition of the histogram (using the UNIX command "graph") on the standard output.

OPTIONS

Any command-line arguments are passed to "graph".

EXAMPLES

To compute and plot a histogram on the default printer:

```
hist image | grhist | lpr -g
```

To plot a (precomputed) histogram on a Tektronix 4014:

```
grhist <histogram | plot -T4014
```

SEE ALSO

IPW: btoa, hist, rmhdr

UNIX: awk, graph, plot

NOTES

The option syntax (of "graph") is nonstandard.

The current implementation of grhist converts the histogram to a text stream and preprocesses it with "awk". This is flexible, but slow.

NAME

hist -- compute image histogram

SYNOPSIS

```
hist [-m mask] [image]
```

DESCRIPTION

hist reads an IPW {image} (default: standard input) and computes its histogram. The histogram is written to the standard output as a single-line IPW image, whose sample offsets represent the pixel values in the input image, and whose pixel values are frequency counts.

OPTIONS

-m Histogram only those pixels masked by nonzero values in the {mask} image.

DIAGNOSTICS

different size pixels: bands 0,{band}

All input bands must have the same number of bits and bytes per pixel. This is because the number of possible pixel values in the input image governs the number of samples in the single output line.

mask is not same size as input image

If -m is specified, then {mask} must have the same number of lines and samples as the input image.

EXAMPLES

Given a DEM "elev" and a drainage basin mask "basin", the following command will calculate a histogram of the elevations within the basin:

```
hist -m basin elev
```

To generate a single-band histogram and convert it to text for further processing by non-IPW software:

```
demux -b {band} | hist | rmhdr | btoa -4
```

SEE ALSO

IPW: btoa, grhist, histeq, rmhdr

NOTES

The output histogram has 32-bit pixels. This make it effectively unreadable by IPW programs that convert their input values to floating point (e.g., primg), since the floating-point conversion routine will attempt, and probably fail, to allocate a 2**32-element lookup table.

hist will eventually be modified so that the output pixel size in each band is the minimum necessary to accommodate the largest output value.

HISTEQ **HISTEQ**

NAME

histeq -- make histogram-equalization look-up table

SYNOPSIS

```
histeq [-n size] [-o min,max]
       [-i min,max] [-f floor] [-c ceil]
```

DESCRIPTION

histeq reads an IPW histogram from the standard input and writes an IPW lookup table to the standard output. The lookup table may applied to the original image (with the IPW lutx command) to produce an image with a nearly-flat (i.e. equalized) histogram.

OPTIONS

-n	There are {size} elements in the histogram (default: 256).
-i	Use only frequencies for pixels between {min} and {max} in the histogram (default: 0, {size}-1)
-o	Output only values between {min} and {max} (default: 0, {size}-1)
-f	Have the output lookup table map input values less than the specified minimum to {floor} (default: output minimum).
-c	Have the output lookup table map input values greater than the specified maximum to {ceil} (default: output maximum).

DIAGNOSTICS

bad option causes 0-bit output values
bad option causes {nbytes}-byte output values

These messages indicate that one or more of the options would have resulted in a lookup table with pixels outside the permissible size range of 1..32 bits.

EXAMPLES

To produce a histogram-equalized version of "image":

```
        hist image | histeq | lutx -i image
```

If "image" has 12-bit pixels, and the output image should have 8-bit pixels, then replace the histeq command above with:

```
        histeq -n 4096 -o 0,255
```

To avoid having extreme values overwhelm the output mapping, you could exclude them by specifying an input range:

```
        histeq -n 4096 -o 0,255 -i 1,4094
```

SEE ALSO

IPW: cnhist, hist, lutx

UNIX: awk

Pratt, W. K., "Digital Image Processing", Wiley, New York, 1978, pp 311-318.

NOTES

The histogram is always read from the standard input (i.e. no histogram file operand is accepted).

The -n option exists only because histeq is currently implemented as shell script and does not read the histogram’s header.

NAME

hor1d -- compute angles to local horizon along image rows

SYNOPSIS

hor1d [-b] [horizon options] [image]

DESCRIPTION

hor1d reads elevations from {image} (default: standard input) and writes an image of along-line horizons to the standard output.

hor1d is almost always invoked indirectly by the horizon command, which allows horizons to be computed along arbitrary azimuths.

OPTIONS

All of horizon’s options are recognized by hor1d, plus:

-b	Compute backward horizons (default: forward).

The value of the "-a" option is not used by hor1d, but it is essential for interpreting hor1d’s output, so it is stored in the header of the output image.

SEE ALSO

IPW: horizon

NAME

horizon -- compute angles to local horizon at given azimuth

SYNOPSIS

```
horizon -a phi [-z zen] [-u cos] [-d delta] [image]
```

DESCRIPTION

horizon reads elevations from {image} (default: standard input) and writes (to the standard output) an image whose pixels encode the local horizon angles in the direction {azimuth} degrees (-180..180) from south (positive east). The value of each output pixel is the cosine of the angle from the zenith to the pixel’s horizon in the forward (increasing sample coordinates) direction. (Note that this value is also the sine of the angle from true horizontal to the pixel’s horizon.)

OPTIONS

-a The direction of forward azimuth (i.e. increasing samples along a line) is {phi} degrees east of south (-180..180).

-d The input grid spacing is {delta} (default: from "geo" header, or 1 if input image has no "geo" header). Must be >0. The units should be the same as for the elevations.

The following options change the output from linearly quantized cosines to a 1-bit mask in which 1’s indicate horizon angles greater than a specified threshold. They are typically used to specify a solar zenith angle, the output being a mask of pixels where the sun is visible.

-u Mask horizon angles with cosines greater than {cos}.

-z Mask horizon angles greater than {zen} degrees (0..90).

DIAGNOSTICS

spacing in geodetic header ignored

> A -d option overrides any spacing information in the image header.

both -u and -z specified, -z over-ridden

> If both -u and -z are specified then -z is ignored.

input file has {nbands} bands

> The input image must have only 1 band.

only 1 line in input image
only 1 samp in input image

> The input image must have at least 2 lines and 2 samples.

input file has no LQH, raw values used
no geodetic header, spacing set to 1.0

> These deficiences in the input image will introduce linear errors into the horizon calculations.

FILES

$TMPDIR/horizon{NNNNN}

> temporary command file, removed when horizon exits

EXAMPLES

To compute northwest horizons:

```
horizon -a -135
```

To produce a mask of all northwest horizon angles greater than 45 degrees:

```
horizon -a -135 -z 45
```

(i.e., any pixels that would be shadowed by adjacent terrain at this solar zenith and azimuth will be masked as 0.)

SEE ALSO

IPW: hor1d, skew, transpose

Dozier, J., Bruno, J., and P. Downey, "A faster solution to the horizon problem", Computers & Geosciences, vol. 7, pp. 145-151, 1981.

NOTES

horizon is shell script that skews and/or transposes the input image to orient its scan lines in the direction {azimuth}, then calls hor1d to perform the actual horizon calculations.

NAME

interp -- interpolate between breakpoints

SYNOPSIS

```
interp
```

DESCRIPTION

interp copies pairs (X, Y) of text integers from the standard input to the standard output. If successive Xs differ by more than +- 1, then the intervening X values are also printed, along with linearly interpolated integral Ys.

DIAGNOSTICS

input line must have exactly 2 fields

EXAMPLES

The following input:

```
0 0
8 4
```

produces the following output:

```
0 0
1 1
2 1
3 2
4 2
5 3
6 3
7 4
8 4
```

To construct an IPW lookup table linearly mapping 12-bit pixels into 8-bit pixels:

```
interp | mklut -i 12 -o 8
0 0
4095 255
```

SEE ALSO

IPW: interp, mklut

UNIX: awk

NOTES

All Xs and Ys must be integers.

NAME

ipw -- list IPW commands

SYNOPSIS

```
ipw
```

DESCRIPTION

ipw displays a nicely-formatted listing of available IPW commands.

FILES

$IPW/lib/bins

This file contains a list of directories, relative to $IPW, that contain executable IPW commands and scripts. The format of this file is:

```
name    description
```

The default version of this file is:

```
bin     general-purpose
etc     maintenance and support
```

EXAMPLES

The command:

```
ipw
```

produces the following output:

```
IPW general-purpose commands:

bitcom    edhdr     histeq    lqhx      mksath  ...
cmpimg    edimg     hor1d     lutx      mksunh  ...
...

IPW maintenance and support commands:

atob    btoa    install ipwlint ipwmake ipwman  ...
...

----------------------------------------------
Type "command -H" for a synopsis of "command".
```

NAME

ipw2hds -- convert IPW image to HDS sixel format

SYNOPSIS

```
ipw2hds [image]
```

DESCRIPTION

ipw2hds reads an IPW {image} (default: standard input) and writes an equivalent image in HDS sixel format on the standard output. The output is suitable for display on an HDS 3200 series terminal without further processing.

DIAGNOSTICS

input image must be single-band

input image must have 1-bit pixels

EXAMPLES

To display "image" on your HDS 3200 terminal:

```
        dither image | ipw2hds
```

SEE ALSO

IPW: demux, dither

"HDS3200 Programmer's Reference Manual", Human Designed Systems, Philadelphia, 1988, # DN-13c4-8802-1.

NOTES

ipw2hds MAY be usable with other sixel-oriented devices (e.g., DEC printers) -- this has NOT been tested.

NAME

ipw2ps -- convert IPW image to PostScript

SYNOPSIS

```
ipw2ps [-r] [-h height] [-w width] [image]
```

DESCRIPTION

ipw2ps reads an IPW {image} (default: standard input) and writes a PostScript version of the image to the standard output.

OPTIONS

-r	Rotate the image 90 degrees on output (e.g., to align the long axis of the image with the long axis of the PostScript device).
-h	The PostScript image should be no more than {height} inches high (default: 9.5).
-w	The PostScript image should be no more than {width} inches wide (default: 7.0).

Note that -h and -w define a bounding box for the PostScript image -- they do NOT change the image’s aspect ratio.

DIAGNOSTICS

image height {height} too large (max 11 inches)
image width {width} too large (max 8.5 inches)

The output PostScript image must fit on an 8.5 by 11 sheet of paper.

image height {height} too small
image width {width} too small

{height} and {width} must be greater than 0.

input image must have only 1 band
input image must have only 1 byte per pixel

These are limitations imposed by PostScript’s "image" operator.

EXAMPLES

To render an image on a PostScript printer:

```
ipw2ps | lpr -PPostScript
```

SEE ALSO

UNIX: lpr

NOTES

Bilevel PostScript output devices use halftone screening to simulate multiple gray levels. Aliasing will occur as the pixel density of the output image approaches the halftone screen frequency.

The 8.5 inch by 11 inch output restriction will be relaxed.

PostScript is a trademark of Adobe Systems, Inc.

NAME

ipw2sun -- convert IPW image to Sun rasterfile

SYNOPSIS

```
ipw2sun [image]
```

DESCRIPTION

ipw2sun reads an IPW {image} (default: standard input) and writes it to the standard output in Sun Microsystems, Inc.’s "rasterfile" format.

DIAGNOSTICS

```
input image has {nbands} bands (only 1 allowed)
input image has {nbits}-bit pixels (only 1 or 8 allowed)
input image has {nbytes}-byte pixels (only 1 allowed)
```

These restrictions are inherent in the (current) Sun rasterfile format.

EXAMPLES

To display an 8-bit IPW image in a SunView window on a monochrome display:

```
dither | ipw2sun | rastool
```

SEE ALSO

IPW: dither, rastool

rasterfile(5), in "UNIX Interface Reference Manual", part number 800-1303-04, Sun Microsystems, Inc.

NAME
        ipw2xim -- prepare IPW image for X display

SYNOPSIS
        ipw2xim [image]

DESCRIPTION
        ipw2xim reads an IPW {image} (default: standard input) and writes its equivalent in "xim" format on the standard output. An image in xim format may be displayed in an X window by the xim or xxim commands.

DIAGNOSTICS
        image must have 1 band

                ipw2xim currently supports only single-band images.

        band 0 has more than 256 levels per pixel

                The xim format does not support more than 8 bits per color.

EXAMPLES
        To display "image" in an X window:

```
ipw2xim image | xim
```

SEE ALSO
        IPW: xim

NOTES
        xim and xxim automatically requantize their input (e.g., by dithering) to the resolution the X display device.

        ipw2xim should modified to create color xim images. Since the xim color image format is band sequential, the easiest way to do this would be to have ipw2xim process 3 single-band input images:

```
ipw2xim red green blue
```

```
NAME
        ipwman -- generate IPW manual pages from source file header
                  comments

SYNOPSIS
        ipwman command ...
        ipwman function ...

DESCRIPTION
        ipwman simulates the UNIX "man" command for IPW, by
        extracting the header comments from the IPW source files
        corresponding to the specified {command}s or library
        {function}s, and writing them to the standard output.

        Leading "*"s or "#"s are stripped from the header comments.
        Each group of header comments associated with a particular
        command or function has the following appended:

         - the date the source file was last modified

         - a form feed (ASCII NP) character

DIAGNOSTICS
        No information for:
                {command}
                ...

                The specified {command}s either do not exist or have
                no appropriate source file comments.

FILES
        $IPW/src/*/{command}/main.c
        $IPW/src/*/{command}/{command}.sh

                These files are searched for command header
                comments.

        $IPW/src/lib/lib*/{function}.c
        $IPW/src/lib/lib*/*/{function}.c

                These files are searched for function header
                comments.
```

EXAMPLES

To print the manual pages for the "gradient" and "shade" commands:

```
ipwman gradient shade | pr | lpr
```

("pr" is used so that the hardcopy will be paginated and dated.)

SEE ALSO

IPW: ipw

UNIX: lpr, man, pr

NOTES

ipwman’s output is essentially verbatim C source or shell script comments; no attempt is made to introduce fancy formatting (multiple fonts, points sizes, etc.).

ipwman will be modified to search user-specified directories for source files containing header comments.

NAME

lincom -- linear combination of bands

SYNOPSIS

```
lincom [-c coef,...] [-n nbits] [image]
```

DESCRIPTION

lincom reads a multi-band {image} (default: standard input) and writes a linear combination of its bands on the standard output. For each input sample, the corresponding output sample is:

```
p[0] * k[0] + ... + p[nbands-1] * k[nbands - 1]
```

where p is the input pixel value, k is a user-specified coefficient, and nbands is the number of input bands.

OPTIONS

-c	per-band coefficients (default: 1 / nbands). If only one coefficient is specified, it is applied to all input bands. Otherwise, the number of coefficients must be a multiple of nbands. Each successive group of nbands coefficients will be used to create a new output band.
-n	Use {nbits} bits per output pixel (default: maximum number of bits per input pixel).

FILES

$TMPDIR/lin{NNNNN}	temporary copy of the output image

EXAMPLES

To create an average of all of the input bands:

```
lincom
```

To subtract the band 1 from the band 0 of a 2-band image:

```
lincom -c 1,-1
```

To create a 2-band output image from a 2-band input image, in which the first output band is a sum and the second output band is a difference:

```
lincom -c .5,.5,1,-1
```

SEE ALSO

IPW: bitcom, mult

NOTES

lincom creates a temporary copy of the output image since it must make two passes over its output, one to determine the minima and maxima in each band, and another to quantize it accordingly.

NAME

lqhx -- transform image values using new headers

SYNOPSIS

```
lqhx [-h hdrs] [-i image]
```

DESCRIPTION

lqhx copies {image} to the standard output, requantizing its pixels according to the quantization parameters of the {hdrs} image. Specifically, the output bands will have the same pixel sizes as {hdrs}, and will receive (and be quantized according to) any corresponding "lq" (linear quantization) headers in {hdrs}.

OPTIONS

-h Read quantization parameters from {hdrs} (default: standard input). Pixel size information (number of bytes, number of bits) is obtained from the BIH (basic image header). Any "lq" headers in {hdrs} (there MUST be at least one) are copied to the output image. Any other headers or image data in {hdrs} are ignored.

-i Read image data from {image} (default: standard input).

At least one of -h and/or -i must be specified.

DIAGNOSTICS

new LQH not valid

The {hdrs} file has an invalid "lq" header, or does not have any "lq" headers.

input image and LQH file have different # bands

The {hdrs} and {image} files must have the same number of bands.

EXAMPLES

To requantize "image2" to the same pixel sizes and ranges of values as in "image1":

```
        lqhx -h image1 -i image2
```

To requantize a single-band "image" such that the input values 0..1 are distributed over 10 bits:

```
        mkbih -s 1 -l 1 -i 10 -f |
                mklqh -m 0,0,1023,1 -f |
                lqhx -i image
```

Note that the -s and -l options are required by mkbih, even though those header fields are subsequently ignored.

SEE ALSO

IPW: mkbih, mklqh

NOTES

lqhx does not check that the quantization borrowed from {hdrs} is appropriate for {image}, i.e., that the pixel values in {image} lie in the ranges specified by the "lq" headers in {hdrs}. Pixels below (above) the range of output values will be set to the lowest (highest) output value.

NAME

lutx -- apply lookup table to image

SYNOPSIS

```
lutx [-l lut] [-i image]
```

DESCRIPTION

lutx copies the input {image} to the standard output, transforming its pixel values according the lookup table {lut}.

A lookup table, like a histogram, is a single-line IPW image. To transform a pixel from a given band through a lookup table, the value of the pixel is interpreted as an image sample coordinate. The appropriate band at that sample in the lookup table supplies the replacement pixel value.

OPTIONS

-i	Read image from {image} (default: standard input).
-l	Read lookup table from {lut} (default: standard input).

At least one of -i and/or -l must be specified.

DIAGNOSTICS

{n}-element LUT can’t map {nbits}-bit pixels

The number of elements (samples) in the lookup table must be at least as large as the number of possible pixel values in any band of the input image.

image and LUT have different # bands

{image} and {lut} must have the same number of bands.

not a lookup table (# lines > 1)

{lut} must be a 1-line IPW image.

EXAMPLES

To produce a histogram-equalized version of "image":

```
hist image | histeq | lutx -i image
```

To convert "image" with 12-bit pixels to 8-bit pixels, with linear scaling:

```
interp | mklut -i 12 | lutx -i image
0 0
4095 255
```

SEE ALSO

IPW: hist, histeq, interp, mklut

NOTES

All input image headers are copied to the output image, even those (such as "lq" headers) whose contents may be invalidated by lutx’s arbitrary modifications to the input pixel values. We see no simple solution to this problem.

NAME

mkbih -- make an IPW basic image header

SYNOPSIS

```
mkbih -l nlines -s nsamps [-b nbands]
      [-y nbytes,...] [-i nbits,...] [-a annot,...]
      [-f] [data]
```

DESCRIPTION

mkbih creates an IPW basic image header (BIH) and writes it to the standard output. The contents of {data} are then copied to the standard output. mkbih therefore allows an IPW header to be prepended to image data from a non-IPW source.

OPTIONS

-l The BIH will indicate {nlines} lines per image.

-s The BIH will indicate {nsamps} samples per image line.

-l and -s must always be specified.

-b The BIH will indicate {nbands} bands per image sample (default: 1).

-y The BIH will indicate {nbytes} bytes per pixel (default: 1, or the minimum necessary to accommodate the specified {nbits}).

-i The BIH will indicate {nbits} bits per pixel (default: 8, or all bits in the specified {nbits}).

-y and -i may have either 1 or {nbands} arguments. If there is 1 argument and {nbands} is greater than 1, then the argument applies to all bands.

-a The BIH will indicate {annot} as the annotation (i.e. commentary) for each band (default: no annotation). If {nbands} is greater than 1 and -a is specified, then it must have exactly {nbands} arguments.

-f Force header output only; i.e., do not attempt to copy {data} to the standard output. This allows creation of a standalone header to which image data may later be appended, or which may be passed as control information to another IPW program (e.g. lqhx).

DIAGNOSTICS

{nbits} won’t fit in {nbytes} bytes

Only the following combinations of {nbytes} and {nbits} are allowed:

nbytes	nbits
1,2,4	1..8
2,4	9..16
4	17..32

input file not allowed with "-f" option

EXAMPLES

To prepend an IPW BIH to a 512 line by 512 sample single-band image with 8-bit pixels:

```
mkbih -l 512 -s 512 image
```

To prepend an IPW BIH to a 6000 line by 7000 sample single-band 8-bit image being read directly from tape:

```
mkbih -l 6000 -s 7000 -f >image
dd bs=7000 <{tape-device} >>image
```

Note the use of ">>" to append the image data to the file containing the standalone header.

SEE ALSO

IPW: lqhx, mk*h, prhdr, rmhdr

UNIX: dd

NOTES

The annotation string may not contain any commas (they will be interpreted as option argument separators).

```
NAME
        mkgeoh -- add a geodetic header to an image

SYNOPSIS
        mkgeoh -c csys -u units -o u,v -d du,dv
               [-f] [-b band,...] [image]

DESCRIPTION
        mkgeoh makes an IPW geodetic ("geo") header.  {image}
        (default: standard input) is then copied to the standard
        output with the "geo" header inserted.

OPTIONS
        -c      The geodetic coordinate system identifier is {csys}
                (e.g., "utm").

        -u      {u}, {v}, {du}, and {dv} are specified in {units}
                (e.g., "meters").

        -o      The coordinates of image line 0 and sample 0 in
                {csys} are {u} and {v}, respectively.

        -d      The distances between adjacent image lines and
                samples in {csys} are {du} and {dv}, respectively.

        -c, -u, -o, and -d must always be specified.

        -f      Force header output only; i.e., do not copy any
                pixel data from {image} to the standard output.
                Note that there must still be at least an input BIH,
                and any input headers (except superseded "geo"
                headers) will still be copied to the output.

        -b      The "geo" header will be applied only to the
                specified {band}s (default: all).

DIAGNOSTICS
        bad band number: {band}

                A nonexistent input band was specified with -b.

        band {band}: replacing previous GEO header

                Band {band} already had a "geo" header, which was
                replaced by the newly-created one.
```

EXAMPLES

The following command creates a "geo" header appropriate for a 5-meter grid located in the Sierra Nevada, California:

```
        mkbih -c utm -u meters -o 4051800,349350 -d -5,5
```

Note the negative line spacing: UTM northings run in the opposite direction from IPW line numbers.

SEE ALSO

IPW: flip, gradient, hor1d, horizon, mkbih, prhdr, rmhdr, transpose, viewcalc, window, zoom

NOTES

There are not (yet) any standard identifiers for {csys} and {units}, although some IPW programs (e.g., gradient) assume that any {units} beginning with "m" are meters.

NAME

mklqh -- add a linear quantization header to an image

SYNOPSIS

```
mklqh -m in,out[,in,out,...] [-u units] [-i interp]
      [-f] [-b band,...] [image]
```

DESCRIPTION

mklqh makes an IPW linear quantization ("lq") header. {image} (default: standard input) is then copied to the standard output with the "lq" header inserted.

OPTIONS

-m Construct a linear mapping between the breakpoints {in,out},... At least 1 {in,out} pair must be supplied. The breakpoint pairs 0,0 and 2^nbits-1,0 are assumed unless explicitly overridden.

-m must always be specified.

-u The floating-point pixel values are expressed in units of {units} (e.g., "W m^-2 sr^-1 nm^-1") (default: none; this field is for annotation only).

-i Use {interp} to interpolate floating-point values between breakpoints (default: "linear"; this is the only currently supported value).

-f Force header output only; i.e., do not copy any pixel data from {image} to the standard output. Note that there must still be at least an input BIH, and any input headers (except superseded "lq" headers) will still be copied to the output.

-b The "lq" header will be applied only to the specified {band}s (default: all).

DIAGNOSTICS

bands {band1} and {band2} have different # bits / pixel

> All bands to which the "lq" header is applied must have the same number of bits per pixel.

must specify pixel,fpixel pairs for -m

> -m must have an even number of arguments.

no band %d

> A nonexistent input band was specified with -b.

replacing band {band} "lq" header

> Band {band} already had an "lq" header, which was replaced by the newly-created one.

EXAMPLES

To construct an "lq" header that will map 8-bit pixels (0..255) into the floating-point range 0..1:

```
mklqh -m 255,1
```

Note that the breakpoint 0,0 is assumed.

To construct an "lq" header that will map 12-bit pixels (0..4095) into the range 2762..3417:

```
mklqh -m 0,2762,4095,3417
```

SEE ALSO

IPW: gradient, hor1d, lincom, lqhx, mkbih, mstats, mult, prhdr, rmhdr, shade, viewcalc, wedge

NOTES

There are not (yet) any standard identifiers for {units}.

The default breakpoints can lead to unexpected quantization mappings or errors from mklqh. For example, the mapping

```
mklqh -m 0,0,127,1
```

succeeds for 7-bit pixels, but for 8-bit pixels there would be an additional implicit breakpoint at 255,0, which would make the mapping non-monotonic.

NAME

mklut -- make look-up table

SYNOPSIS

```
mklut [-i ibits] [-o obits] [-k bkgd]
```

DESCRIPTION

mklut creates an IPW lookup table and writes it to the standard output. The lookup table is loaded with text values read from the standard input. Each line of input must have the form:

```
in      out
```

mklut sets the {in}'th element (0-relative) of the lookup table to {out}. The input lines must be sorted so that the {in} values are in numerically ascending order.

OPTIONS

-i {ibits} bits per input value (default: 8). The output lookup table will contain 2ˆ{ibits} entries.

-o {obits} bits per output value (default: 8). The output lookup table will contain {obits}-bit pixels.

-k Initialize the lookup table to {bkgd} (default: 0).

-i, -o, and -k must all have positive nonzero integer arguments.

DIAGNOSTICS

{ibits}: too many bits per input pixel

A 2ˆ{ibits}-entry lookup table won't fit in memory.

EXAMPLES

The following pipeline will convert 12-bit pixels to 8-bit pixels, with linear scaling:

```
interp | mklut -i 12 | lutx -i in_image > out_image
0 0
4095 255
```

SEE ALSO

IPW: interp, lutx

UNIX: sort

NOTES

mklut currently will create only single-band LUTs.

NAME

mksath -- add a satellite header to an image

SYNOPSIS

```
mksath [-p platform] [-s sensor] [-l location]
       [-t yr,mon,day[,hr[,min[,sec]]]]
       [-f] [-b band,...] [image]
```

DESCRIPTION

mkgeoh makes an IPW satellite ("sat") header. {image} (default: standard input) is then copied to the standard output with the "sat" header inserted.

OPTIONS

-s	The image data were acquired by {sensor} (e.g., "TM", "AVIRIS", "ASAS", etc.).
-p	{sensor} was mounted on {platform} (e.g., "Landsat-5", "ER-2", "C-130", etc.).
-l	The image data were acquired at or over {location}. This should NOT be a geodetic specification (use mkgeoh), but a sensor-, platform-, or project-specific identifier (e.g., experimental site name, Landsat path/row, etc.).
-t	The image data were acquired on date {yr}/{mon}/{day} at time {hr}:{min}:{sec.s...} GMT. {yr} must be fully specified (i.e. 90 means 90 A.D., not 1990). {hr} is 24-hour time.

At least 1 of -p, -s, -l, and/or -t must be specified.

-f	Force header output only; i.e., do not copy any pixel data from {image} to the standard output. Note that there must still be at least an input BIH, and any input headers (except superseded "sat" headers) will still be copied to the standard output.
-b	The "sat" header will be applied only to the specified {band}s (default: all).

DIAGNOSTICS

bad band number: {band}

> A nonexistent input band was specified with -b.

band {band}: replacing previous "sat" header

> Band {band} already had a "sat" header, which was replaced by the newly-created one.

EXAMPLES

SEE ALSO

IPW: mkbih, prhdr, rmhdr, tmpt

NOTES

There are not (yet) any standard identifiers for {sensor}, {platform}, or {location}. The {date} and {time} formats are recommended but not enforced.

NAME

mksunh -- add a solar position header to an image

SYNOPSIS

```
mksunh -z cosz -a azm [-f] [image]
```

DESCRIPTION

mksunh makes an IPW solar position ("sun") header. {image} (default: standard input) is then copied to the standard output with the "sun" header inserted.

OPTIONS

```
-z      {cosz} is the cosine of the solar zenith angle.

-a      {azm} is the solar azimuth in radians from south,
        positive east.

-z and -a must always be specified.

-f      Force header output only; i.e., do not copy any
        pixel data from {image} to the standard output.
        Note that there must still be at least an input BIH,
        and any input headers (except superseded "sun"
        headers) will still be copied to the standard
        output.
```

EXAMPLES

The following generates a "sun" header for a Landsat image acquired with a solar elevation of 57 degrees and a solar compass bearing of 119 degrees:

```
        mksunh -z 0.838671 -a 1.064651 image

where 0.838671 = cos(90 - 57)
  and 1.064651 = (180 - 119) * (PI / 180)
```

SEE ALSO

IPW: mkbih, prhdr, rmhdr

NOTES

The "sun" header will always be applied to ALL bands of the input image.

mksunh should be a little more forgiving about the angular units that it accepts ...

NAME

mkwinh -- add a window header to an image

SYNOPSIS

```
mkwinh [-b line,samp] [-d dl,ds] [-f] [image]
```

DESCRIPTION

mkwinh makes an IPW window ("win") header. {image} (default: standard input) is then copied to the standard output with the "win" header inserted.

OPTIONS

-b {line},{samp} are the window coordinates of line 0 and sample 0 in the input image (default: 0, 0).

-d {dl,ds} are the window line and sample spacings in the input image (default: 1, 1).

-f Force header output only; i.e., do not copy any pixel data from {image} to the standard output. Note that there must still be at least an input BIH, and any input headers (except superseded "win" headers) will still be copied to the standard output.

EXAMPLES

To add a window header indicating that the window (line, sample) coordinates of the image origin are (7, 5):

```
mkwinh -b 7,5
```

SEE ALSO

IPW: flip, mkbih, prhdr, rmhdr, transpose, window, zoom

NOTES

The "win" header will always be applied to ALL bands of the input image.

NAME

mstats -- image multivariate statistics

SYNOPSIS

```
mstats [-c classes] image
```

DESCRIPTION

mstats computes the basic multivariate statistics (per band means and variances and interband covariances) for the input {image} (default: standard input). The statistics are written as text on the standard output, in the following format:

```
#<stats>

nbands

mean mean ...

variance
covariance variance
...

* class

nsamps
```

"#<stats>" is a magic cookie that introduces each set of statistics.

{nbands} is the number of bands in the input image.

The {mean}s are the mean pixel values of each input band.

The {variance}s and {covariance}s are the lower triangle of the variance-covariance matrix, with the variance of band i at (i, i) and the covariance of bands i and j at (i, j).

{class} (always preceded by "*") is the class number for which this set of statistics was computed.

{nsamps} is the number of input samples in {class}.

OPTIONS

-c Read classes from {classes} image. This must be a single-band image with the same dimensions as the input image. A separate set of statistics is accumulated and output for each unique pixel value in the class image. The pixel at (line, sample) in the input image is assigned to the class of the pixel at (line, sample) in the class image.

If -c is not specified then all input pixels are reported as being in class "0".

EXAMPLES

If "image" is a multiband satellite image, and "basin" is a registered mask of a drainage basin, then

```
mstats -c basin image
```

will compute the separate multivariate statistics for areas inside and outside the basin.

SEE ALSO

IPW: demux, edimg, hist, mux, rmhdr

NOTES

mstats will always use any linear quantization ("lq") headers in the input image to transform the input values. If statistics for the quantized (raw) pixel values are desired, then any "lq" headers must be removed from the input image before it is passed to mstats.

Computing variances involves accumulating a sum of squares of all input values. The larger the input image, the more likely that the output variances and covariances will contain rounding errors.

means and (co-)variances will be printed with the maximum precision supported by the host architecture; the low-order digits should probably be ignored.

There is considerable overlap between mstats and hist; they may be integrated someday.

NAME
```
        mult -- multiply or divide bands
```

SYNOPSIS
```
        mult [-n nbits] [-r bands,...] [image]
```

DESCRIPTION
```
        mult reads a multi-band {image} (default: standard input)
        and writes the product of its bands on the standard output.
        For each input sample, the corresponding output sample is:

                p[0] * p[1] * ... * p[nbands-1]

        where {p} is the input pixel value and {nbands} is the
        number of input bands.
```

OPTIONS
```
        -r      Use the reciprocal of the pixel value in the
                specified {band}s.  To avoid division by 0, any 0
                values in the specified {band}s will be set to 1.

        -n      Use {nbits} bits per output pixel (default: maximum
                number of bits per input pixel).
```

DIAGNOSTICS
```
        -r {band}: not that many bands

                A nonexistent input band was specified with -r.

        Image must have more than one band.
```

FILES
```
        $TMPDIR/mult{NNNNNN}    temporary copy of the output image
```

EXAMPLES
```
        To multiply two single-band images together:

                mux image1 image2 | mult

        To divide the first band of a 2-band image by the second
        band:

                mult -r 1
```

SEE ALSO
```
        IPW: demux, mux, lincom
```

NOTES

mult creates a temporary copy of the output image since it must make two passes over its output, one to determine the minima and maxima in each band, and another to quantize it accordingly.

NAME

mux -- band-interleave images

SYNOPSIS

```
mux image ...
```

DESCRIPTION

mux combines 1 or more input {image}s into a single output image that contains all of the input bands:

```
input bands             output bands

image1: 0..n1           0..n1
image2: 0..n2           n1+1..n1+n2

        ... etc. ...
```

DIAGNOSTICS

image size differs from 1st image

All input images must have the same number of lines and samples.

FILES

```
$TMPDIR/mux{NNNNNN}     temporary copy of all input headers
```

EXAMPLES

To combine "red", "green", and "blue" images into a single "color" image:

```
mux red green blue >color
```

SEE ALSO

IPW: demux

NOTES

The maximum number of input images is limited by the number of files that a program may have open simultaneously. This limit can be worked around by piping one mux into another.

```
mux image1 ... imageN | mux - imageN+1 ...
```

NAME

poly -- digitize vectors and fill polygons

SYNOPSIS

```
poly [-s x,y] [file]
```

DESCRIPTION

poly reads a list of vector vertices from the text {file} and writes the digitized vectors to the standard output.

The input is a list of integer (x, y) coordinates, one pair per line:

```
x0 y0
x1 y1
... etc. ...
```

The output is a similar list, including all points that would lie on a line between each successive input coordinate pair.

OPTIONS

-s Fill the input polygon. The input coordinates must be the vertices of a closed polygon, (i.e. the last coordinate pair must be the same as the first), and the specified ({x}, {y}) must be inside the polygon. The output will contain all points inside the polygon, as well as all points on the polygon’s boundary.

EXAMPLES

The following command:

```
poly
0 0
3 3
```

yields:

```
0 0
1 1
2 2
3 3
```

If "roi" contains the corners (in intrinsic line and sample coordinates) of a region-of-interest then the following command will inscribe the boundaries of the region onto "image":

```
poly roi | edimg -i image
```

SEE ALSO

IPW: edimg

NOTES

The output coordinates appear in order of increasing (x, y), regardless of the order of the input coordinates.

Output coordinates on polygon edges are generated by a Bresenham algorithm. If -s is specified, then output coordinates inside a polygon are generated by a flood-fill algorithm, using ({x}, {y}) as a seed location.

NAME

prhdr -- print IPW image headers

SYNOPSIS

```
prhdr [image ...]
```

DESCRIPTION

prhdr copies the IPW headers of the input {image}s (default: standard input) to the standard output. If more than one {image} is specified, then each group of output headers will be preceded by:

```
::::::::::::::
{image}
::::::::::::::
```

EXAMPLES

The command:

```
prhdr image
```

might produce the following output:

```
!<header> basic_image_i -1 $Revision: 1.8 $
byteorder = 3210
nlines = 480
nsamps = 640
nbands = 1
!<header> basic_image 0 $Revision: 1.8 $
bytes = 1
bits = 8
!<header> image -1 $Revision: 1.4 $
```

SEE ALSO

IPW: mk*h, rmhdr

UNIX: more, sed

NOTES

IPW headers are printable text and are always separated from the image data by a form feed (ASCII NP) character. You can therefore view the header of any IPW image directly with a pagination command such as "more" that pauses when it encounters a form feed.

NAME

primg -- print image pixel values as text

SYNOPSIS

```
primg [-a] [-r] [-c coords] [-i image]
```

DESCRIPTION

primg prints input image pixels as text on the standard output. Each sample is printed on a separate line. The pixel values for each band are printed in band order from left to right, separated by white space.

OPTIONS

-a Print all pixels in image (default: print only pixels specified in {coords}).

-c Read pixel coordinates from {coords} (default: standard input).

At most one of -a or -c may be specified.

-i Read image data from {image} (default: standard input).

At least one of -a, -c, and/or -i must be specified.

-r print quantized (raw) pixel values, bypassing any conversion to floating-point (e.g. via input "lq" headers).

DIAGNOSTICS

bad coordinate file line: {text}

{coords} contains a line {text} than cannot be parsed as two non-negative integers.

bad coordinates (not on image): {line},{sample}

{line},{sample} are illegal coordinates for the input image.

unsorted coordinates: {line},{sample}

The input coordinates must be sorted in ascending line order (sample order is unimportant).

EXAMPLES

To interactively examine pixel value in "image", type the command:

```
primg -i img
```

then type the pixel coordinates on the standard input (but remember, the coordinates must be typed in increasing line order).

If "basin" contains the (line, sample) coordinates of the corners of a drainage basin in the DEM "dem", and {line},{samp} are the coordinates of an arbitrary point within the drainage basin, then:

```
poly -s {line},{samp} basin | primg -i dem
```

will print all of the pixel value in "dem" that lie within the drainage basin.

SEE ALSO

IPW: poly

UNIX: sort

NOTES

-a without -r can be very slow.

primg should allow pixel coordinates to be specified with reference to window ("win") or geodetic ("geo") headers in the input image.

NAME

random -- generate random numbers

SYNOPSIS

```
random -n nlines -r min,max[,...] [-p prec] [-s seed]
```

DESCRIPTION

random prints random numbers as text on the standard output.

OPTIONS

-n Print {nlines} lines of output.

-r Output values will occupy the range {min} to {max} inclusive. Each {min},{max} pair controls an output column.

-n and -r must always be specified

-p Print output values using {prec} digits of precision (default: 0, meaning discard any fractional part).

-s Initialize the random number generator with {seed} (default: obtain seed from system clock). This option can be used to obtain the same output from multiple invocations of random.

DIAGNOSTICS

```
# of values must be > 0
```

-n was specified with a 0 or negative {nlines}.

```
range(s) must be specified by min,max pairs
```

-r was specified with an odd number of arguments.

```
output precision must be >= 0
```

-p was specified with a negative {prec}

EXAMPLES

The command:

```
random -n 5 -r 0,10,0,10
```

might yield:

```
7 1
5 8
7 2
1 8
0 2
```

To obtain the values of 100 randomly selected pixels from a 512 line by 512 sample "image":

```
random -n 100 -r 0,511,0,511 |
        sort -n |
        primg -i image
```

Note that the random coordinates must be sorted before being passed to primg.

SEE ALSO

IPW: primg

UNIX: random, sort

NOTES

random uses the random(3) functions from 4.3BSD UNIX. The source for these functions is normally included with IPW.

NAME

rastool -- display a Sun rasterfile in a SunView window

SYNOPSIS

```
rastool [-b line,samp] [-c line,samp] [-e line,samp]
        [-n nlines,nsamps] [rasterfile]
```

DESCRIPTION

rastool displays the Sun rasterfile {rasterfile} (default: standard input) in a SunView window. {rasterfile} may be monochrome (ras_maptype = RMT_NONE), or it may possess an RGB color table (ras_maptype = RMT_EQUAL_RGB). {rasterfile}'s type must be RT_STANDARD; no encoded formats are currently supported, to allow disk scrolling of images. If {rasterfile} is read from the standard input, then it must be small enough to scroll in memory.

OPTIONS

-b Display a subimage of {rasterfile} beginning at ({line}, {samp}).

-c Display a subimage of {rasterfile} centered at ({line}, {samp}).

-e Display a subimage of {rasterfile} ending at ({line}, {samp}).

-n Display an {nlines} lines by {nsamps} samples subimage of {rasterfile}.

No more than two of -b, -c, -e, and/or -n may be specified.

DIAGNOSTICS

Bad subimage definition

The window specified by the -b, -c, -e, and/or -n options is not a proper subset of {rasterfile}.

EXAMPLES

To display a single-band IPW image on a monochrome display running SunView:

```
dither | sunras | rastool
```

SEE ALSO

IPW: sunras, window

NOTES

rastool was written by Jud Harward, Center for Remote Sensing, Boston University

rastool does not yet support disk scrolling.

RMHDR **RMHDR**

```
NAME
        rmhdr -- delete image headers

SYNOPSIS
        rmhdr [-d header,...] [image]

DESCRIPTION
        rmhdr reads an an IPW {image} (default: standard input) and
        writes only the image data to the standard output.

OPTIONS
        -d      Delete only the specified {header}s (default: all).
                {header} should be the name of the header EXACTLY as
                it appears in the header itself (e.g. "lq" for a
                linear quantization header).  Nonspecified headers
                are copied to the standard output.

EXAMPLES
        For a single-band image with {nbytes} bytes per pixel,

                rmhdr | btoa -{nbytes}

        would be equivalent to

                primg -r -a

        To delete the the "lq" header from an image before running
        mstats (so the statistics will be computed for the quantized
        pixel values):

                rmhdr -d lq | mstats

SEE ALSO
        IPW: btoa, mk*h, prhdr

NOTES
        The default output of rmhdr is NOT a valid IPW image, since
        it has no BIH (nor any other header).  However, if -d is
        specified, then rmhdr’s output IS a valid IPW image, UNLESS
        "basic_image" or "basic_image_i" are explicitly specified as
        {header} arguments.  Some may find this confusing ...

        For the header names that will be recognized by the -d
        option, see the {header}H_HNAME macro definition in
        $IPW/h/{header}h.h.
```

NAME

shade -- calculate cosine of local illumination angle

SYNOPSIS

```
shade [-z zenith] [-u cos] -a azimuth [image]
```

DESCRIPTION

shade read a 2-band slope and aspect image (i.e. output from gradient) from {image} (default: standard input), and writes an image of local illumination cosines (relative to a specified solar position) to the standard output.

OPTIONS

-z The solar zenith angle is {zen} [0..90) degrees.

-u The cosine of the solar zenith angle is {cos} (0..1].

At least one of -z or -u must be specified. If both are specified, then -z is ignored.

-a The solar azimuth is {azm} degrees (-180..180) from south (positive east, negative west).

DIAGNOSTICS

input file has no LQH

An "lq" header is necessary to convert the quantized slopes and aspects to their actual values.

band 0 of input not slope

The range of floating-point values in band 0 is inappropriate for quantized slopes. shade will continue executing, but the output values will almost certainly be bogus.

EXAMPLES

To compute a nice-looking shaded-relief map from a DEM (i.e. one with the sun in the cartographically traditional, but, for the northern hemisphere, physically impossible position of 45 degrees above the northwest horizon):

```
gradient | shade -z 45 -a -135
```

SEE ALSO

IPW: gradient

Dozier, Jeff, and James Frew, "Rapid Calculation of Terrain Parameters for Radiation Modeling from Digital Elevation Data", in "IGARSS ’89 12th Canadian Symposium on Remote Sensing", vol. 3, pp. 1769-1774, 1989.

NOTES

NAME

skew -- skew image lines

SYNOPSIS

```
skew [-a angle] [-h] [image]
```

DESCRIPTION

Skew copies {image} (default: standard input) to the standard output, skewing the origin of successive lines by a specified angle.

OPTIONS

-a Skew lines through {angle} degrees (-45..45). The left edge of the output version of {image} will be tilted {angle} degrees clockwise from vertical. The resulting dead space in the output image is filled with 0-valued pixels.

-h Ignore any skew header in the input image.

If -a IS specified, then the input image must NOT contain a skew ("skew") header (unless -h is specified), and a "skew" header is written to the output image.

If -a is NOT specified, then the input image MUST contain a "skew" header. The skew indicated by this header removed from the output image, and NO "skew" header is written to the output image.

DIAGNOSTICS

image is already skewed

-a was specified, and the input image contains a "skew" header.

image is not skewed

-a was not specified, and the input image does not contain a "skew" header.

band {band} has no skew header
different skew angles: bands 0, {band}

If -a is not specified, then all input bands must have the same "skew" header

FILES

```
$TMPDIR/skew{NNNNNN}    temporary copy of all input headers
```

EXAMPLES

The command

```
skew -a 30
```

causes the following transformation:

```
+-----------+           +---------------+
|           |           |000/          /|
|   input   |           |00/  output  /0|
|   image   |           |0/   image  /00|
|           |           |/          /000|
+-----------+           +---------------+
```

SEE ALSO

IPW: flip, horizon, transpose, viewf

NOTES

skew may be used in conjunction with flip and transpose to reorient an image’s scan lines at an arbitrary angle with respect to the original scan lines.

NAME

sunang -- calculate sun angles

SYNOPSIS

```
sunang -t yr,mon,day,hr[,min[,sec]] [-z min] [-y]
       -b deg[,min[,sec] -l deg[,min[,sec]]
       [-s slope] [-a azm] [-r] [-d]
```

DESCRIPTION

sunang calculates the sun angle (the azimuth and zenith angles of the sun’s position) for a given date, time, and geodetic location. The sun angle is written to the standard output in the format:

```
GMT {weekday} {month} {day} {hr}:{min}:{dec} {year}
-z {degrees} -u {cos(z)} -a {degrees}
```

The first line contains the date and time. The second line contains the zenith angle, the cosine of the zenith angle, and the solar azimuth. The second line is suitable for direct substitution into the command lines of several other IPW programs (horizon, shade, etc.).

```
OPTIONS
        -t      Calculate sun angle on date {yr}/{mon}/{day} at time
                {hr}:{min}:{sec} GMT.  {yr} must be fully specified
                (i.e. 90 means 90 A.D., not 1990).  {hr} is 24-hour
                time.

        -b      Calculate sun angle at latitude {deg},{min},{sec}.
                South latitudes are negative.

        -l      Calculate sun angle at longitude {deg},{min},{sec}.
                West longitudes are negative.

        -t, -b, and -l must always be specified.  {min} and {sec}
        default to 0 if not specified.

        -s      Calculate sun angle for a surface with a slope of
                {deg} (0..90) degrees (default: 0).

        -a      Calculate sun angle for a surface with an azimuth of
                {deg} (-180..180) degrees (default: 0).

        -r      All angles are specified in radians (default:
                degrees, minutes, seconds for -b and -l; decimal
                degrees for -s and -a).

        -z      Use a time zone {min} west of Greenwich (default: 0)
                for the time specified with the -t option.

        -y      Use daylight savings time (default: standard time)
                for the time specified with the -t option.

        -d      Print earth-sun radius vector in addition to sun
                angle.
```

EXAMPLES

To calculate the sun angle in Santa Barbara on 15 February 1990 at 12:30 PM Pacific Standard Time:

```
sunang -b 34,25 -l -119,54 -t 1990,2,15,12,30 -z 480
```

The output would be:

```
GMT Thu Feb 15 20:30:00 1990
-z 47.122 -u 0.680436 -a -5.413
```

Given a DEM image for the Santa Barbara area, the following could be used to generate an index of local beam irradiance (discounting atmospheric effects):

```
gradient {dem} |
        shade ‘sunang {options as above} | tail -1‘
```

SEE ALSO

IPW: gradient, hor1d, horizon, shade

UNIX: tail

Wilson, W. H., "Solar ephemeris algorithm", Reference 80-13, 70 pp., Scripps Institution of Oceanography, University of California at San Diego, La Jolla, CA, 1980.

NOTES

-r does NOT change the output representations of zenith and azimuth; they are always printed as decimal degrees.

-b and -l should accept decimal degrees as well as radians and degrees,minutes,seconds.

NAME

transpose -- transpose an image

SYNOPSIS

```
transpose [image]
```

DESCRIPTION

transpose reads {image} (default: standard input) and writes its transpose on the standard output. Geodetic ("geo") and window ("win") headers will be adjusted appropriately. An orientation ("or") header will be written if the orientation of the output image is non-standard.

DIAGNOSTICS

output image won’t fit in memory

The entire image must fit into (virtual) memory.

EXAMPLES

To rotate an image 90 degrees clockwise:

```
flip -s | transpose
```

SEE ALSO

IPW: flip, skew

NOTES

The in-memory transpose algorithm is quite fast, but it does limit the size of the image that can be transposed.

NAME

```
        viewcalc -- compute sky view and terrain configuration
                    factors
```

SYNOPSIS

```
        viewcalc [-s gradient] [-h horizons]
```

DESCRIPTION

```
        viewcalc reads a gradient image (calculated by "gradient")
        and a horizon angle image (calculated by "horizon") and
        writes a 2-band image of the corresponding sky view and
        terrain configuration factors to the standard output.

        The input horizon angle image should be multiband, with each
        band representing the horizon in one of {nbands}
        equiangularly spaced directions (i.e., the result of
        "mux"ing together several output images from "horizon").

        Band 0 of the output image is the sky view factor, defined
        as the fraction (0..1) of a pixel’s overlying hemisphere
        (centered at the pixel’s zenith) that is subtended by sky
        (as opposed to surrounding terrain).

        Band 1 of the output image is the terrain configuration
        factor.  defined as:

                1 + cos(slope)
                --------------  -  sky view factor
                      2
```

OPTIONS

```
        -s      Read slopes and aspects from {gradient} (default:
                standard input).

        -h      Read horizon angles from {horizons} (default:
                standard input).

        At least one of -s and/or -h must be specified.
```

DIAGNOSTICS

```
# samples different in two input files
files unequal # pixels
```

The gradient and horizon angle images must have the same number of lines and samples.

```
band 0 of input not slope
input slope/azimuth file must have 2 bands
slope/azm image has no LQH
```

The gradient image must be a valid output from the gradient program.

```
horizon image has no HORH
horizon image has no LQH
```

The horizon image must be composed of valid outputs from the horizon program.

EXAMPLES

viewcalc is almost always invoked by viewf, which first calculates the necessary gradient and horizon images.

SEE ALSO

IPW: gradient, hor1d, horizon, viewf

Dozier, Jeff, and James Frew, "Rapid Calculation of Terrain Parameters for Radiation Modeling from Digital Elevation Data", in "IGARSS ’89 12th Canadian Symposium on Remote Sensing", vol. 3, pp. 1769-1774, 1989.

NAME

```
        viewf -- compute sky view and terrain configuration factors
                 directly from elevation image
```

SYNOPSIS

```
        viewf image
```

DESCRIPTION

viewf reads elevation data from {image} (default: standard input) and writes the corresponding sky view and terrain configuration factors to the standard output.

viewf is a script that invokes gradient and hor1d to calculate the necessary gradient and (16) equiangularly spaced horizon images, which are then passed to viewcalc.

DIAGNOSTICS

(see: gradient, hor1d, mux, skew, transpose, viewcalc)

FILES

```
        $TMPDIR/viewf.{NNNNNN}/*

                temporary directory containing intermediate gradient
                and horizon files.

        $TMPDIR/mux{NNNNNN}     (mux temporary file)
        $TMPDIR/skew{NNNNNN}    (skew temporary file)
```

EXAMPLES

The command:

```
        viewf image
```

is equivalent to:

```
        gradient image >{grad}
        horizon -a {azm01} image >{horz01}
        ...
        horizon -a {azm16} image >{horz16}
        mux {horz??} | viewcalc -s {grad}
```

SEE ALSO

IPW: gradient, hor1d, mux, skew, transpose, viewcalc

NOTES

The use of 16 horizon images is hard-coded in viewf. In practice this has proven to yield sufficient resolution even in very rugged terrain.

NAME

wedge -- linear combination of line and sample coordinates

SYNOPSIS

```
wedge -c lcoef,scoef[,...] [-n nbits] [image]
```

DESCRIPTION

wedge reads the headers from {image} (default: standard input) and writes an image with the same number of lines and samples to the standard output. The value of each output pixel is a linear combination of the pixel's raw line and sample coordinates.

OPTIONS

-c Output pixels are assigned the value:

```
line * {lcoef}  +  sample * {scoef}
```

Each pair of {lcoef},{scoef} will be used to generate a new output band.

-n Use {nbits} bits per output pixel (default: log2(max(nlines,nsamps)), where {nlines} and {nsamps} are the dimensions of {image}).

DIAGNOSTICS

must specify an even number of coefficients

-c must have an even number of arguments

band {band}: both coefficients cannot be 0

EXAMPLES

To produce a 512 by 512 single-band diagonal wedge with 8-bit pixels increasing in intensity from the upper left corner:

```
mkbih -l 512 -s 512 -f | wedge -c 1,1 -n 8
```

Note that the input image can be a standalone BIH since only the header is required.

To produce an 8-bit image with 2 wedge channels, the first increasing horizontally, and the second decreasing horizontally, sized to match an existing "image":

```
wedge -c 0,1,0,-1 -n 8 image
```

SEE ALSO

IPW: mkbih

NOTES

Any optional headers, other than "lq" headers, will be copied from the input image to the corresponding bands of the output image.

```
NAME
        window -- extract image window

SYNOPSIS
        window [-b line,samp] [-c line,samp] [-e line,samp]
               [-n nlines,nsamps] [-w band] [-g band] [image]

DESCRIPTION
        window reads an image from {image} (default: standard input)
        and writes a specified window (i.e., subimage) to the
        standard output.
```

```
OPTIONS
        -b      The beginning (i.e. upper-left corner) of the window
                is ({line}, {samp}).

                If ONLY -b is specified, then the window extends
                diagonally from ({line}, {samp}) to the last sample
                on the last line of the input image.

        -c      The center of the window is ({line}, {samp}).  The
                center of an even number of lines or samples is the
                larger of the two possible values; e.g., the center
                line of lines 0..511 is 256, not 255.

                If ONLY -c is specified, then the window is largest
                possible odd number of input lines and samples
                centered on ({line}, {samp}).

        -e      The end (i.e. lower-right corner) of the window is
                ({line}, {samp}).

                If ONLY -e is specified, then the window extends
                diagonally from the first sample on the first line
                of the input image, to ({line}, {samp}).

        -n      The window has {nlines} lines and {nsamps} samples
                per line.

                If ONLY -n is specified, then the window begins at
                the first sample of the first line of the input
                image.

        At least one, and no more than two, of of -b, -c, -e or -n
        must be specified.

        -w      The arguments to -b, -c, and/or -e are specified in
                the coordinates of band {band}'s window ("win")
                header.

        -g      The arguments to -b, -c, and/or -e are specified in
                the coordinates of band {band}'s geodetic ("geo")
                header.

        At most one of -w or -g may be specified.
```

DIAGNOSTICS

specified window exceeds boundaries of input image

> Windows are not clipped to fit the input image; the specification must be correct to start with.

no geodetic header for band {band}
no window header for band {band}

> -g or -w was specified and there was no corresponding header for the specified input band.

FILES

```
$TMPDIR/windo{NNNNNN}   temporary copy of all input headers
```

EXAMPLES

To produce a 2x enlargement of the center of a 512x512 image:

```
window -c 256,256 -n 256,256 | zoom -l 2 -s 2
```

To extract a 1 km by 1 km window from a 5-meter DEM, beginning at UTM northing 4051800 and easting 349350:

```
window -g 0 -b 4051800,349350 -e 4050805,350345
```

SEE ALSO

IPW: mkgeoh, mkwinh, zoom

NOTES

Input "win" and "geo" headers are transformed correctly; other input headers are copied verbatim to the output image.

NAME

zoom -- zoom image by pixel replication or subsampling

SYNOPSIS

```
zoom [-l zline] [-s zsamp] [image]
```

DESCRIPTION

zoom reads {image} (default: standard input) and writes a copy to the standard output, with lines and/or samples either replicated or skipped. Any geodetic ("geo") or window ("win") headers are adjusted as necessary.

OPTIONS

-l Replicate each line {zline} times. If {zline} is negative, then select every {zline}'th line, beginning with line 0.

-s Replicate each sample {zsamp} times. If {zsamp} is negative, then select every {zsamp}'th sample, beginning with sample 0, from each selected line.

At least 1 of -l and/or -s must be specified.

DIAGNOSTICS

line zoom factor must be non-zero
sample zoom factor must be non-zero

Zoom factors must be nonzero integers.

EXAMPLES

To zoom an image by a factor of 2 in both directions:

```
        zoom -s 2 -l 2
```

To histogram every 10th pixel of a large image:

```
        zoom -s -10 -l -10 | hist
```

Zooming by non-integer factors is accomplished by expressing the desired zoom factor as a fraction, and then using the pipeline:

```
        zoom -{direction} {numerator} |
                zoom -{direction} -{denominator}
```

For example, to shrink an image by 3/4 horizontally (perhaps to accommodate a display with a 4:3 aspect ratio):

```
        zoom -s 3 | zoom -s -4
```

SEE ALSO

IPW: window

NOTES

The fractional-zoom pipeline should specified in the order shown (replication before subsampling). Reversing the order (subsampling before replication) may be faster, but results in much greater loss of spatial resolution.

CHAPTER 6: THE PROGRAMMER LEVEL

This chapter contains a description of the C programmer's interface to IPW.

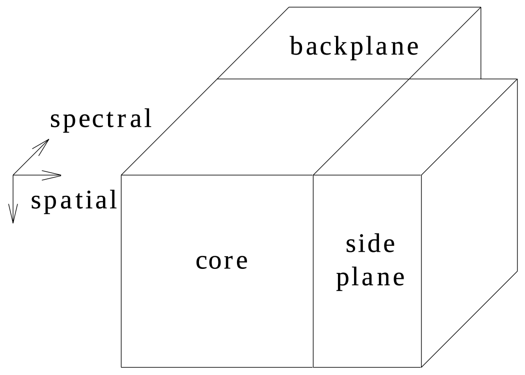
We begin with a detailed description of image data formats; this elaborates on the brief description presented in §3.1. Next, the source-level structure of a generic IPW primitive is presented. IPW shell script programming is briefly described, and finally the IPW programming tools are documented.

The descriptions in this chapter complement the detailed ''manual page'' documentation in chapter 7. The material in chapter 7 is organized for ease of reference, whereas the material in this chapter is designed to present the programming features of IPW in a more tutorial fashion, in the order in which they would normally be used.

6.1. IMAGE STRUCTURE

At the lowest level, the external representation of an IPW image is an unstructured stream of bytes. The ability to pass image data through UNIX pipes is fundamental to IPW, and pipes allow only sequential access, so any logical structure imposed on this byte stream must be amenable to strictly sequential processing.

The most fundamental structuring of the image data stream is the separation between header and pixel data. Both header and pixel data are homogeneous (i.e., they are not interspersed with one another), and all header data precede all pixel data in the byte stream. To borrow the terminology of [Martin 1988], IPW allows no ancillary data in the ''backplane'' or ''side planes'' of an ''image cube'':



6.1.1. Headers

The term **header** in IPW has two distinct meanings, depending on the context in which it is used. The phrase ''image header data'' refers to the non-pixel portion of the image data stream. The header data consist of at least one ''header'', which in this context refers to a logically related subset of the header data. All programmer access to an image's header data is via functions that deal with these logical headers.

6.1.1.1. Storage format

Image header data are structured into **records**, each of which is a variable-length text string terminated by a newline. Each record contains one or more data **fields**, separated by **white space** (i.e., any combination of blanks and horizontal tabs).

The first record in a header is always a **preamble** record. The preamble record contains 4 fields, which appear in the following order:

- a **magic string**, currently `!<header>`, which identifies the record as preamble record;
- the **header name**, which identifies the type of this header (e.g., `win`, `geo`, etc.);
- the (positive integer) image **band number** with which this header is associated;
- the **version number** of the header[19].

The records following the preamble contain the actual data associated with this header. Each of these records contains a single keyword-value pair, in the following format:

keyword=*value...*

The keyword, =, and value(s) must be separated by white space. Any white space at the beginning or end of the record is ignored.

The validity of a keyword and its associated value(s) are defined by the parsing routines for that header. Thus it is possible (but not recommended) for the same keyword to have different meanings in different headers. The presence of an unrecognized keyword or value, or the absence of a required keyword or value, is usually treated as an error.

There are two headers that violate some of the restrictions described above. One is the **BIH** (basic image header), described in detail in the next section. The other is the **image preamble**, a preamble record whose presence in the image data stream signals the end of the header data and the beginning of the pixel data. The image preamble has the following special properties:

- Since it is not associated with any particular band, its band number field is set to `-1`.
- It always contains a form-feed character (ASCII `014`) immediately preceding the terminating newline (see §3.1.1).
- It is followed immediately by pixel data.

The BIH and the image preamble are the only headers that **must** be present. All other headers are optional, and are associated with a single image band; e.g., if geodetic information is required for *N* bands, then there must be *N* geodetic headers.

The order of headers in the data stream must observe the following constraints:

- The BIH must be first.
- The image preamble must be last.
- Optional headers with the same name must be contiguous.

A collection of contiguous same-name optional headers is a **header group**. Header groups may appear in any order, and the headers within a group need not be sorted by band (or any other characteristic).

[19] In the current implementation, this is the RCS `Revision` string from the header's *header*`.h` file. Because of the way RCS keywords are expanded, this field MAY contain embedded blanks.

6.1.1.2. Basic image header

The BIH was originally designed to contain just enough information to allow the location and extraction of any single pixel in the image. The current implementation of the BIH also contains annotation and processing history records; these were placed in the BIH since they are maintained automatically, and the BIH is the only header that the IPW software may assume will be present.

The BIH is unique in that it has a **per-image** component (named `basic_image_i`), in addition to the per-band components (named `basic_image`). The per-image component avoids replication of fundamental data that by definition cannot vary from band to band. The following keywords must appear in the per-image component:

- `nlines`: number of image lines
- `nsamps`: number of samples per image line
- `nbands`: number of bands per image sample
- `byteorder`: byte order on the host where the image was created

Like the image preamble, the preamble of the per-image component has a `-1` in the band number field.

The following keywords may appear in the per-band component(s):

- `bytes`: number of bytes per pixel
- `bits`: number of significant bits per pixel
- `annot`: annotative text
- `history`: processing history

`bytes` and `bits` are required. Both are needed since IPW pixels always begin on byte boundaries.

`annot` and `history` records are optional, and may appear more than once per component. An `annot` record typically contains descriptive text supplied by the user when the image was ingested into IPW, although some programs write to this field automatically. Each `history` record contains a copy of the command line of an IPW program that has written the image. In both the `annot` and `history` records, the entire line following the = is treated as a single value.

6.1.1.3. Optional headers

IPW images may contain an unlimited number of optional headers. As long as a header adheres to the format described in §6.1.1.1 above, it will be recognized as a header by all IPW programs. Note that recognition is NOT the same as interpretation: there is no guarantee that a particular IPW program will be able to make use of the contents of any particular optional header. However, any IPW program will be able to either skip over, or copy unchanged to another image data stream, any input header.

As of this writing, IPW supports the following optional headers:

header name	**description**	**see programs**
`geo`	geodetic coordinates	`mkgeoh`
`hor`	line azimuth	`hor1d, viewcalc`
`lq`	pixel quantization parameters	`mklqh`
`or`	image orientation	`flip, transpose`
`sat`	sensor platform parameters	`mksath`
`skew`	scan line skew angle	`skew`
`sun`	solar geometry	`mksunh`
`win`	extrinsic image coordinates	`mkwinh, window`

The "see programs" column indicates program documentation in §5.2 that may be consulted for more information.

6.1.2. Pixels

A **pixel** in IPW is the atomic unit of image data. Pixels are scalar quantities. In the context of remote sensing, a pixel usually represents a single measurement of some physical phenomenon (e.g., radiance).

A **sample** is a location in an image raster. With each sample is associated as many pixels as there are bands in the image. A **sample coordinate** is the offset of a sample relative to the first sample in an image **line**.

The term "pixel" is also used in a more specialized sense, referring to a particular representation of pixel data within a program; see §6.2.4.2.

6.1.2.1. Storage format

Pixels are stored externally as unsigned integers; i.e., an N-bit pixel may assume integer values from 0 to 2^N-1 inclusive. The following constraints on pixel configuration are imposed by the pixel I/O routines (see §6.2.4.2):

- Pixels must begin on a byte boundary; i.e., any given pixel must occupy an integral number of bytes, and no other pixel data may occupy those bytes.
- No pixel may occupy more than `sizeof(pixel_t)` bytes, where `pixel_t` is an implementation-defined type (usually `unsigned int`).

In addition to simplifying pixel I/O, these constraints help make the pixel data more portable. Moving pixel data between IPW implementations on different machine architectures requires no more than (possibly) rearranging a pixel's byte order[20]. Similarly, exporting IPW pixel data to other software systems requires no more of the foreign software than the ability to rearrange bytes.

Not all of the bits in the bytes occupied by a pixel need be utilized, since IPW stores a bit as well as a byte count in the BIH. Unused high-order bits are ignored. It is advantageous to limit the number of bits per pixel to those strictly necessary, since integral pixel values are often used as indices into lookup tables, which double in size for each additional pixel bit.

[20] The `byteorder` field in the BIH helps automate this process.

6.1.2.2. Band interleaving

If we consider a multiband image as a 3-dimensional array, then the three possible pixel indexing or **interleaving** schemes[21] may be expressed as follows, in C-language notation:

band sequential (BSQ)	`image[band][line][sample]`
band interleaved by line (BIL)	`image[line][band][sample]`
band interleaved by pixel (BIP)	`image[line][sample][band]`

Each class of generic operation described in Chapter 4 requires a certain **context**: a number of pixels previously read that must be retained in memory in order to compute the current output pixel value. The size of the context may be influenced by the interleaving scheme selected. For univariate point operations, any interleaving scheme suffices, since no context is required. For geometric operations, the context cannot be formulated *a priori*, since it varies from none (for `window`) to the entire image (for `flip` and `transpose`). However, for multivariate point and neighborhood operations, the context is predictable, and varies substantially according to the interleaving scheme used:

operation	**BSQ**	**BIL**	**BIP**
multivariate point	*LS(B-1)*	*S(B-1)*	*B-1*
neighborhood	*US*	*USB*	*USB*

where:

L	= number of lines
S	= number of samples
B	= number of bands
U	= number of lines in a kernel

BIP interleaving is clearly superior for multivariate point operations, outweighing the advantage of BSQ for neighborhood operations. Hence IPW uses BIP interleaving for image pixel data. Externally, the interleaving occurs at the byte level; e.g., a 1-byte pixel for band 0 followed by a 2-byte pixel for band 1, etc. Internally, BIP interleaving is preserved as the pixel values are converted to uniform integer or floating-point types.

6.2. PROGRAM STRUCTURE

In this section we describe the structure of an IPW primitive at the source code level. Sufficient information will be presented for a competent C programmer to understand the source code for existing IPW primitives, and to develop new ones.

6.2.1. Skeleton

Each IPW primitive has its own source directory. The following files will always be present; there may be additional source files, depending on how complicated the primitive is. Within the source file, sequences of text between `@`s (at-signs) are to be replaced by program-specific text.

[21] We consider transpositions of lines and samples to be equivalent.

main.c contains the source for main, the program's entry point. This file is described in detail in the §6.2.2.

pgm.h is #included by all of the source files comprising the primitive. This file contains:

- #defines for program-specific macros
- extern declarations for program-specific functions
- a typedef for the PARM_T structure. This structure contains all of the global variables used by the primitive; it is described in detail in §6.2.2.3.

parm.c contains the defining instance of parm, the global variable of type PARM_T.

headers.c contains the code which processes the input and output image headers. These tasks are described further in §6.2.3.

primitive.c contains the code that actually implements the function of the primitive. While there is no ''skeleton'' code for this, a major portion will certainly be devoted to image I/O, which is described in §6.2.4.

Finally, there is a Makedefs file, to allow ipwmake to build the primitive.

If a new primitive is being developed, ''skeleton'' versions of all of these files may be copied from the directory $IPW/skel/pgm.

All IPW source files must #include the header file ipw.h. This is an ''umbrella'' header file that #includes many other header files, among them <stdio.h>, <limits.h>, and <float.h>. If a source file contains multiple #includes, then ipw.h must be first.

6.2.2. Main

Every IPW main must address the following specific tasks:

- The header comments for the program (see §A.1.1.1) must appear near the beginning of the main.c file.
- A description of the command line expected by the program must be created and passed to the IPW initialization routine.
- Global parameters in the parm data structure must be initialized.
- Any files required by the program must be opened.

Following these conventional tasks, each of which is elaborated in the following sections, main typically calls headers, to process any image headers, and then *primitive* to perform the actual image processing.

6.2.2.1. Header comments

Every main.c file in IPW must contain **header comments**, in the format described in §A.1.1.1. These comments constitute the principal documentation of the primitive for both users and programmers. While programmers will typically examine the source file directly, users usually access the comments indirectly via the ipwman command, which extracts the header comments and displays them in manner similar to the UNIX man command. The command-specific documentation in Chapter 5 is essentially identical to the output of ipwman.

6.2.2.2. Command-line arguments

IPW programs accept standard UNIX command lines:

```
command -option optarg operand
```

The command-line arguments are either options, optargs, or operands.

An **option** is a single character preceded by a minus sign. An option may be **boolean** (i.e., its presence or absence is the extent of the information provided to the program), or it may have one or more **optarg**s (option arguments) associated with it.

operands appear after any options. In IPW, operands are always the names of input files.

An IPW `main` manages its command line by:

- initializing an **option descriptor** data structure for each desired option;
- calling `ipwenter` with an array pointers to the option descriptors;
- possibly checking for missing or conflicting options by calling `opt_check`;
- extracting the information returned in the option descriptors.

The data structures, macros, and functions that manage option descriptors are declared in `getargs.h`, which must be `#included` by every IPW `main.c`.

Data structures

An IPW `main` must define an option descriptor for every option the program is prepared to accept. The format of an option descriptor definition is:

```
static OPTION_T opt_letter = {
        'letter', "description",
        optarg_type, "optarg description",
        required_code, min_#_optargs, max_#_optargs
};
```

The definition must have storage class `static` because the address of `opt_`*letter* is used to initialize another data structure, described below[22]. Also, portions of `static` data structures not explicitly initialized are implicitly initialized to `0`, which the option processing routines take advantage of.

letter is the option letter that will be recognized on the command line. In lieu of '*letter*', the macro `OPERAND` may be specified, to bind this descriptor to the command-line operands.

description is the description of this option that will appear in the program's **help message**. The help message is generated whenever a program is invoked with the special option `-H`, and also as a result of some common command-line errors (missing required option, missing optargs, etc.).

letter and *description* are sufficient to initialize an option descriptor for a boolean option. If any of the following are specified, they indicate that optargs are expected.

optarg_type indicates the type to which any optargs will be converted. It can be one of the following macros:

---

[22] Also, many versions of C do not allow non-`static` data structures to be initialized.

macro	optarg type	may be copied into
INT_OPTARGS	aint_t	int
REAL_OPTARGS	areal_t	double
STR_OPTARGS	astr_t	char *
LONG_OPTARGS	along_t	long

An analogous set of *type*_OPERANDS macros may be specified in an operand descriptor.

optarg description is printed as a placeholder for the optarg in the program's help message.

required_code is either OPTIONAL or REQUIRED, and indicates whether or not the option must be specified.

min_#_optargs is the minimum number of optargs that will be accepted for this option. If not specified, it defaults to 1.

max_#_optargs is the maximum number of optargs allowed for this option. If *max_#_optargs* is not specified, the number of optargs is unlimited[23].

Here are examples of complete option descriptor definitions, from the window primitive:

```
static OPTION_T opt_b = {
        ’b’, "begin line,sample",
        REAL_OPTARGS, "coord",
        OPTIONAL, 2, 2
};

static OPTION_T opt_g = {
        ’g’, "window specified re: this band’s geodetic header",
        INT_OPTARGS, "band",
        OPTIONAL, 1, 1
};

static OPTION_T operands = {
        OPERAND, "input image file",
        STR_OPERANDS, "image",
        OPTIONAL, 1, 1,
};
```

These definitions indicate that -b, if specified, must have exactly two floating-point optargs, while -g, if specified, must have exactly one integer optarg. There may be no more than one string-valued operand.

Functions

Command-line arguments, available in the argc and argv parameters to main, are parsed by the ipwenter function. argc, argv, and an array of pointers to the option descriptors must be passed to ipwenter. The array is defined as follows:

[23] The operating system usually imposes some (comfortably large) limit on the size of a command line.

```
static OPTION_T *optv[] = {
        &opt_letter,
        &opt_letter,
        ...
        &operands,
        0
}
```

i.e., the address of each option descriptor are used to initialize the `optv` array.

A call to `ipwenter` must be the first executable statement in `main`. The call looks like this:

```
ipwenter(argc, argv, optv, "description");
```

where *description* is a one-line text description of the purpose of the program, such as would appear in the `NAME` section of the header comments.

`ipwenter` will terminate execution with a help message if the command line is obviously incorrect, or if the `-H` option was specified. Otherwise, on return, it has placed the requested options, optargs, and operands in hidden variables within the option descriptors.

There are certain interactions between options that can be checked mechanically but cannot be specified solely via option descriptors. For example, one option may require the presence of another, or two particular options may be incompatible. These kinds of interactions can be checked by the `opt_check` function, whose calling sequence is:

```
opt_check(n_min, n_max, n_opts,
          &opt_letter,  &opt_letter, ... )
```

n_opts indicates the number of pointers to option descriptors that follow as arguments. `opt_check` checks that at least *n_min* and at most *n_max* of these options were specified. If either of these tests fail, `opt_check` cause program termination with an appropriate error message. If necessary, `opt_check` may be called more than once.

After the call to `ipwenter` and any calls to `opt_check`, the options, optargs, and operands may be accessed by using the corresponding option descriptor as an argument to one of the following macros:

macro	**returns**
`got_opt(opt_`*letter*`)`	TRUE if this option was specified, else FALSE
`n_args(opt_`*letter*`)`	number of optargs specified with this option
`int_arg(opt_`*letter*`, `*i*`)`	*i*'th `int` optarg for this option
`long_arg(opt_`*letter*`, `*i*`)`	*i*'th `long` optarg for this option
`real_arg(opt_`*letter*`, `*i*`)`	*i*'th `double` optarg for this option
`str_arg(opt_`*letter*`, `*i*`)`	*i*'th `char *` optarg for this option
`int_argp(opt_`*letter*`)`	array of `int` optargs for this option
`long_argp(opt_`*letter*`)`	array of `long` optargs for this option
`real_argp(opt_`*letter*`)`	array of `real` optargs for this option
`str_argp(opt_`*letter*`)`	array of `char *` optargs for this option

Note that optargs may be accessed either by value, or indirectly via an array of optargs associated with each option. There is no type checking performed by these macros, so you must be sure to use the macro that corresponds to the *type*_OPTARGS specified in the option descriptor definition.

6.2.2.3. Parameter initialization

Software engineers generally prefer argument lists, as opposed to global variables, for passing data between modules, since argument lists tend to make the coupling between modules more explicit, and also are not vulnerable to accidental access outside the context of intermodule communication. However, if argument lists are the only means of intermodule communication, they can become so long as to be excessively prone to programming errors, as well as difficult for a programmer to read.

IPW attempts to solve this problem by distinguishing between **variable** arguments, which are likely to assume different values every time a function is called, and **parametric** arguments which, once set, retain the same value throughout the execution of the program. Variable arguments are passed between modules in function argument lists, whereas parametric arguments are collected in a single global data structure. By enclosing parametric arguments in a data structure, the risk of accidental access is drastically reduced, and instances of access are visually highlighted by the data structure notation.

Data structures

Parametric arguments are always kept in a global structure parm, which has the typedef'd type PARM_T. The programmer decides which variables belong in parm and edits the declaration of PARM_T in pgm.h accordingly. pgm also contains an extern declaration for parm, which is defined in the separate file parm.c. For example, here are the relevant lines from the pgm.h file for the flip primitive:

```
typedef struct {
        int     i_fd;   /* input image file descriptor  */
        int     o_fd;   /* output image file descriptor */
        bool_t  lines;  /* ? flip lines                 */
        bool_t  samps;  /* ? flip samples               */
} PARM_T;

extern PARM_T   parm;
```

Since pgm.h is #included by all source files in the program, any module may access the parameters via parm.*membername*.

Option settings and converted optarg values are typical candidates for inclusion in a parm structure. In the example above, the members lines and samps correspond to the presence or absence of the -l and -s options to flip. Thus the main for flip contains:

```
parm.lines = got_opt(opt_l);
parm.samps = got_opt(opt_s);
```

6.2.2.4. File access

All IPW file access is via UNIX **file descriptors**. This is the lowest level of file access provided by UNIX. File descriptors are used instead of higher-level objects (e.g., `FILE` pointers) because:

- File access at the file descriptor level is available, and behaves predictably, on all UNIX systems. By contrast, the so-called "portable" `stdio` routines have notable inconsistencies between implementations (e.g., in mechanisms for buffer size specification), and vary dramatically in the efficiency of binary I/O (`fread` and `fwrite`).
- File descriptors are `int`s by definition, and thus may be used as array indices. This greatly simplifies the implementation of the I/O routines, since a single "handle" with a simple type may be used to access a variety of internal data structures.

The IPW I/O subsystem is organized into 3 layers, each of which is described in detail in a subsequent section:

- `uio`: unstructured I/O on byte streams;
- `pixio`: integer pixel I/O
- `fpio`: floating-point pixel I/O

There are also read and write routines provided for each header type.

In `main` an IPW program's interaction with the I/O subsystem is limited to obtaining file descriptors, either for named files specified as optargs or operands, or for the standard input or standard output. File descriptors are returned by the following `uio` routines:

- `uropen`: open a named file for reading
- `uwopen`: open a named file for writing
- `ustdin`: returns file descriptor for standard input
- `ustdout`: returns file descriptor for standard output

Note that, in order to ensure proper `uio` initialization, file descriptors for the standard input and output should be obtained from `ustdin` or `ustdout`, rather than by directly accessing the presumed defaults (typically 0 and 1).

Note also that files are opened for reading OR writing, but not both; this is in keeping with the strictly sequential processing model enforced by UNIX pipes.

The following is a typical minimal example of how an IPW `main` would access its input and output files. The input file is either specified by name as a command line operand, or the standard input is used. The output file is the standard output. The file descriptors are stored in the `parm` structure, since they are used by other modules and do not change once they are set.

```
/*
 * access input file
 */
       if (!got_opt(operands)) {
               parm.i_fd = ustdin();
       }
       else {
               parm.i_fd = uropen(str_arg(operands, 0));
               if (parm.i_fd == ERROR) {
                       error("can’t open
                             str_arg(operands, 0));
               }
       }

       no_tty(parm.i_fd);
/*
 * access output file
 */
       parm.o_fd = ustdout();
       no_tty(parm.o_fd);
```

The function `no_tty` checks whether the argument file descriptor is connected to the user’s terminal; if it is, program execution is terminated with an error message. This catches the common error of the user failing to redirect the standard input and output.

6.2.3. Header processing

In a typical IPW primitive, the programmer supplies a `headers` function that:

- processes any input image headers;
- copies information from selected input headers to the **parm** structure;
- creates any new output image headers;
- writes any output image headers.

Of course, not every `headers` function will perform all of these tasks: some primitive have no input images, or produce no output image, or ignore all image headers except the BIH. In other primitives, it is impossible to perform all header processing in a single `headers` function, for example if input pixels must be processed before the output headers can be created.

A basic decision to be implemented in the `headers` function is the disposition of optional input image headers (i.e., any headers other than the BIH). An input header may be:

- **skipped** (i.e., discarded);
- **copied** directly to the output image;
- **ingested**- its contents placed in a data structure so as to be accessible to the program.

It is important to realize that a header must be explicitly ingested before its contents can be examined. Similarly, once a header has been ingested, it will not appear in the output image unless explicitly written to it.

The BIH is treated differently from optional headers in that it is always explicitly ingested and explicitly written, whereas optional headers may be disposed of *en masse* using functions described below. explicitly ingested, This distinction exists primarily because the act of ingested or writing a BIH initializes essential state information in the I/O subsystem, without which the automatic processing of the optional headers could not occur.

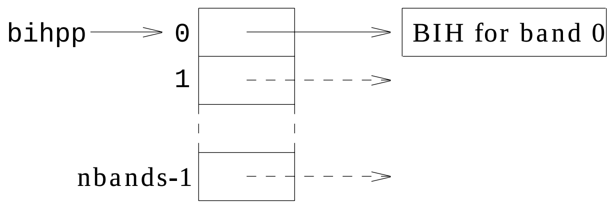
6.2.3.1. Data structures

Ingested and newly-created image headers are accessed via arrays of pointers to structures of type *HEADER*`H_T`, which is `typedefd` in the *header*`h.h` file for each header. Arrays are used since there may be as many headers of a specific type as there are bands in the image, while pointers are used since the in-memory representations of the headers are dynamically allocated.

For example, the BIH is described in `bih.h`, which contains a `typedef` for `BIH_T`. The ingest and creation functions for a BIH return a pointer to an array of pointers to BIHs:

```
BIH_T **bihpp;  /* The "pp" suffix is customary */
```

where `bihpp[`*i*`]` is a pointer to the BIH for band *i*.



To avoid explicit reference to structure member names within an IPW program, each *header*`h.h` file also defines a set of function macros that take (at minimum) a pointer to a header as an argument, and return the value of a header field. These macros all have names of the form *header*`h_`*member*. For example,

```
bih_nbytes(bihpp[0])
```

returns the number of bytes per sample in image band 0.

Note that the in the special case of the BIH, the per-image values will be replicated for each band; e.g., `bih_nlines(bihpp[`*i*`]) = bih_nlines(bihpp[`*j*`])` for any valid *i* and *j*. To lessen confusion, programs should always access these replicated fields via band 0.

6.2.3.2. Functions

In this section we will illustrate possible image header processing sequences with a series of code fragments. These examples assume that the following members of the `parm` structure have been initialized:

```
int i_fd;  /* input image file descriptor  */
int o_fd;  /* output image file descriptor */
```

and that the following local variables have been declared:

```
BIH_T **i_bihpp;  /* -> array of input BIHs  */
BIH_T **o_bihpp;  /* -> array of output BIHs */
```

All IPW headers are provided with a standard set of I/O and utility functions, declared in the *header*`h.h` file. Thus, for example, there is a `winhread` function for `win` headers analogous to the `bihread` function demonstrated below.

The first operation performed on an input image is to read its BIH:

```
i_bihpp = bihread(parm.i_fd);
if (bihpp == NULL) {
        error("can't read BIH");
}
```

The first operation performed on an output image is to write its BIH. In this simple case, we write a duplicate of the input BIH:

```
o_bihpp = bihdup(i_bihpp);
if (o_bihpp == NULL) {
        error("can't duplicate input BIH");
}

if (bihwrite(parm.o_fd, o_bihpp) == ERROR) {
        error("can't write BIH");
}
```

Once the input BIH has been read and the output BIH written, all remaining input headers may be copied directly to the output image by calling `copyhdrs`:

```
nbands = bih_nbands(o_bihpp[0]);
copyhdrs(parm.i_fd, nbands, parm.o_fd);
```

Only headers for band numbers less than `nbands` will be copied. This is useful if the output image has fewer bands than the input image.

Alternatively, if all optional input headers should be ignored,

```
skiphdrs(parm.i_fd)
```

will read and discard any remaining input headers.

If it is desired to ingest some of the optional headers, then a mechanism similar to that used for command-line argument specification is employed. A **header request** data structure, similar to an option descriptor, is declared and initialized for each type of header that is to be either ingested or skipped. Then, a **header request vector** is initialized with the addresses of these data structures, and is passed to the function `gethdrs`.

Here is a sample code fragment:

```
static GETHDR_T h_lqh = {LQH_HNAME, (ingest_t)lqhread};
static GETHDR_T h_winh = {WINH_HNAME};
static GETHDR_T *hv[] = {&h_lqh, &h_winh, NULL};

LQH_T **i_lqhpp;
...
/* read BIH from i_fd; write BIH to o_fd */
...
gethdrs(parm.i_fd, hv, nbands, parm.o_fd);

if (got_hdr(h_lqh)) {
        i_lqhpp = (LQH_T **) hdr_addr(h_lqh);
}
```

The header request h_lqh specifies that the header named LQH_HNAME is to be ingested using the function lqhread, while the request h_winh specifies that the header named WINH_HNAME is to be ignored (no ingest function is specified). The call to gethdrs reads headers from parm.i_fd, ingests or skips any that are specified in hv, and copies any others whose band numbers are less than nbands to parm.o_fd.

After gethdrs returns, the macro got_hdr may be used to check whether a requested header was ingested; if it was, then the macro hdr_addr may be used to fetch the address of the array of header pointers.

Note that copyhdrs, skiphdrs, gethdrs, and the associated macros and type declarations are all declared in gethdrs.h, which must be #included by any source files accessing these functions.

The easiest way to create a new header is to duplicate an existing one. Unfortunately, this approach fails if:

- no header of the desired type has been ingested;
- existing headers of the desired type have the wrong number of bands.

Therefore, every header has an associated *header*hmake function, which creates a **single** header (NOT an array of headers) of the desired type, and returns a pointer to it.

Because an array of headers, one per band, is usually required, the following code fragment is typical. winhmake is used for illustration; be aware that each headerhmake function has a unique calling sequence depending on the contents of the particular header.

```
o_winhpp = (WINH_T **) ecalloc(nbands, sizeof(WINH_T *));
if (o_winhpp == NULL) {
        error("can't allocate WIN header pointers");
}

for (band = 0; band < nbands; ++band) {
        WINH_T *winhp;

        winhp = winhmake(bline, bsamp, dline, dsamp);
        if (winhp == NULL) {
                error("band %d: can't make WIN header",
                      band);
        }

        o_winhpp[band] = winhp;
}
```

Note that if the same `win` header were to be written to all output bands, there could be a single call to `winhmake` before the `for` loop, and the loop could simply set all elements of `o_winhpp` to the same value.

`copyhdrs`, `skiphdrs`, and `gethdrs` all leave the input data stream positioned just before the first pixel in the image[24]. However, header **output** must be explicitly terminated by calling `boimage` with the output image file descriptor:

```
if (boimage(parm.o_fd) == ERROR) {
        error("can't terminate header output");
}
```

This writes an image preamble (see §6.1.1.1) to the output image, indicated that any subsequent output to `parm.o_fd` will be pixel data.

6.2.4. Pixel processing

Image pixel data may be read and written with the `uio`, `pixio`, or `fpio` layers of the I/O subsystem. The decision as to which layer to use is largely dictated by the particular application: whether the application needs to access individual pixel values, and whether those values need to be converted between integral and floating-point representations.

Calls to different I/O layers may **not** be mixed on the same file descriptor, since each layer maintains separate state information and internal buffers.

6.2.4.1. uio

`uio` functions are seldom called directly when processing pixel data, since they present data exactly as they appear externally; i.e., without any unpacking of the pixel bits into accessible data types. However, in addition to the file access functions described in §6.2.2.4, `uio` functions may be invoked if access to the individual pixel values is not required, as in bulk copying of pixel data from an input to the output

[24] Neither return an error code; instead, they cause program termination with an error message if any errors are encountered.

image (e.g., by the window primitive). uio also supports line-oriented text I/O, which is used by primitives that process text data (e.g., convolve, edimg, etc.).

Functions

The fundamental uio operations are uread and uwrite, whose usage closely parallels their UNIX namesakes:

```
int     n_do, n_done;
addr_t buf;

n_done = uread(parm.i_fd, buf, n_do);

n_done = uwrite(parm.o_fd, buf, n_do);
```

where n_do is the number of bytes to transfer, n_done is the number of bytes successfully transferred (or ERROR), and buf is a pointer[25] to at least n_do bytes of memory, containing the data read or to be written.

Often it is desirable to skip an arbitrary number of bytes of input data. With UNIX system calls this could be accomplished quite efficiently with lseek, but this will not work on a pipe, nor is it compatible with uio's internal buffering. Therefore, the routine urskip is provided to simulate a forward seek on an input file:

```
long n_do, n_done;

n_done = urskip(parm.i_fd, n_do);
```

Note that the byte counts are long, to accommodate large files.

Since one of the main reasons for accessing the uio layer directly is to simply copy an arbitrary number of bytes from one file descriptor to another, the function ucopy is provided to encapsulate this operation:

```
long n_do, n_done;

n_done = ucopy(parm.i_fd, parm.o_fd, n_do);
```

copies n_do bytes from parm.i_fd to parm.o_fd.

For reading and writing line-oriented text, uio provides the functions ugets and uputs, analogous to the stdio functions fgets and fputs, respectively. The line-oriented functions may be freely intermixed with uread or uwrite, although this is uncommon[26].

The following example demonstrates the use of the line-oriented uio in conjunction with the C library functions sscanf and sprintf; this is the recommended strategy for parsing text I/O in IPW:

---

[25] addr_t is the generic IPW pointer type; see §6.2.6.

[26] The low-level image header I/O routines take advantage of this.

```
char i_text[MAX_INPUT], o_text[MAX_INPUT];
int  val1, val2;

if (ugets(parm.i_fd, i_text, sizeof(i_text)) == ERROR) {
        error("read error");
}

if (sscanf(i_text, "%d %d", &val1, &val2) != 2) {
        uferr(parm.c_fd);  /* see §6.2.5.5 */
        error("bad input: %s", i_text);
}

 ... some processing here ...

(void) sprintf(o_text, "%d %d", val1, val2);

if (uputs(parm.o_fd, o_text) == strlen(o_text)) {
        error("read error");
}
```

`ugets` reads bytes into `i_text` until either a newline is read, or `sizeof(i_text)-1` bytes have been read, or end-of-file is reached on `parm.i_fd`. A null byte (`'\0'`) is appended to the last byte read, so the `i_text` buffer may be treated like a C string. `uputs` writes bytes from `o_text` to `parm.o_fd` until a null byte is encountered (the null byte is not written). Both of these functions return the number of bytes read or written, or `ERROR`.

6.2.4.2. pixio

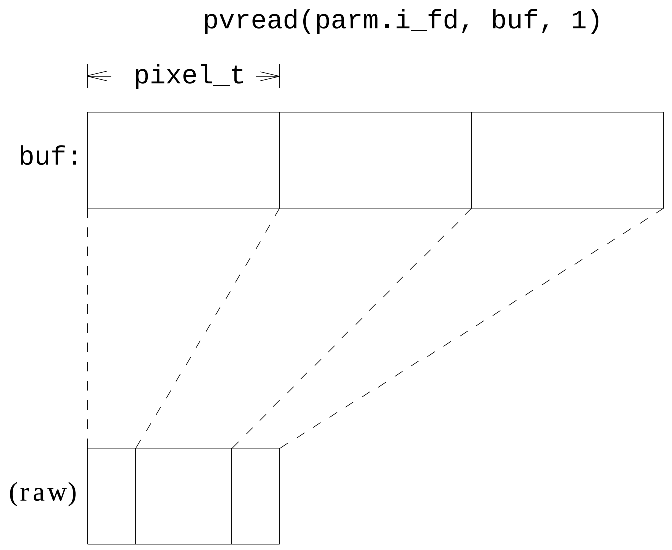
The `pixio` layer, which calls the `uio` layer, presents pixels as unsigned integers of type `pixel_t` (installation-dependent, but usually `unsigned int`). Thus the `pixio` layer is appropriate for operations that require access to individual pixels and/or their quantized (i.e., raw) values.

Any source files accessing any of the `pixio` routines must contain a:

```
#include "pixio.h"
```

line. A BIH must have been read from or written to a file descriptor before any `pixio` operations may be performed on it.

The `pixio` routines transfer data in units of **pixel vectors**. A pixel vector is the collection of pixel values associated with a single image sample; i.e., there are as many pixels in a pixel vector as there are bands in the image. The `pixio` routines handle the conversion between the external byte-aligned and the internal `pixel_t`-aligned representation of a pixel vector:



Functions

pvread and pvwrite are the only two pixio functions that are normally used:

```
pixel_t *buf;
int      n_do, n_done;
...

buf = (pixel_t *) ecalloc(nsamps * nbands, sizeof(pixel_t));
if (buf == NULL) {
        error("can’t allocate pixel I/O buffer");
}
...

n_done = pvread(parm.i_fd, buf, n_do);

n_done = pvwrite(parm.o_fd, buf, n_do);
```

where n_do is the number of pixel vectors to read or write, and n_done is the number of pixel vectors actually read or written.

Note that since the units of transfer are pixel vectors, the calculation of the buffer size to use must take account of both the desired number of samples to transfer, and the number of pixels per sample.

Accessing the individual pixels in a one-dimensional pixel_t buffer is typically done as follows:

```
int      samp;
pixel_t *bufp;
...

bufp = buf;
for (samp = 0; samp < nsamps; ++samp) {
        int band;

        for (band = 0; band < nbands; ++band) {
                /* do something with bufp[band] */
        }

        bufp += nbands;
}
```

Alternatively, pixel vectors may be read into a 2-dimensional buffer, allowing random access to any pixel in the buffer:

```
pixel_t **buf;
...

buf = (pixel_t **) allocnd(sizeof(pixel_t), 2, nsamps, nbands);
if (buf == NULL) {
        error("can’t allocate pixel I/O buffer");
}
...

n_done = pvread(parm.i_fd, buf[0], n_do);

/* do something with buf[samp][band] */
```

See §6.2.6.1 for a discussion of `allocnd`.

6.2.4.3. `fpio`

The `fpio` layer, which calls the `pixio` layer, presents pixels as floating-point values of type `fpixel_t` (installation-dependent, but usually `float`). Thus the `fpio` layer is appropriate for operations that require access to the de-quantized (i.e., “real world”) pixel values.

Any source files accessing any of the `fpio` routines must contain a:

```
#include "fpio.h"
```

line.

The use of the `fpio` layer imposes the following additional execution overhead, with respect to the `pixio` layer:

- data copying between the `fpio` and `pixio` layers;
- conversion between integer and floating-point data representations;
- software-emulated floating-point operations, on systems without floating-point hardware.

The IPW programmer should be aware of these costs in deciding which I/O layer to use in implementing a particular application.

Data structures

The conversion between `pixel_t` and `fpixel_t` values in `fpio` is controlled by a set of lookup tables or **map**s, one per band. While these maps are maintained internally by `fpio`, there are various implicit and explicit actions the programmer can take to affect these maps.

The maps are initialized on the first call to an `fpio` routine. There are two possible initializations:

- If an `lq` header has been ingested (by `lqhread`) for the band whose table is map initialized, then the mapping described by the `lq` header is used to initialize the map.
- Otherwise, the map is initialized with a 1:1 mapping (i.e., a `pixel_t` value of 100 maps into an `fpixel_t` value of 100.0.

The `lq` headers may be ingested explicitly by including a header request of the form:

```
static GETHDR_T h_lqh = {LQH_HNAME, (ingest_t)lqhread};
```

in the vector passed to `gethdrs`. Alternatively, there is a function `fphdrs` that behaves identically to `copyhdrs`:

```
fphdrs(parm.i_fd, nbands, parm.o_fd);
```

except that any `lq` headers in the input image are used to initialize the `fpio` maps for both the input and the output image.

Functions

Prior to reading or writing and pixel data, the `fpio` maps may be accessed by the `fpmap` function:

```
fpixel_t **map;
int       *maplen;

map = fpmap(parm.i_fd);
maplen = fpmaplen(parm.i_fd);
```

if it is necessary to modify or otherwise use any of the values therein. The `fpixel_t` value corresponding to the `pixel_t` value *value* in band *band* is in `map[`*band*`][`*value*`]`. The array `maplen` contains the length of each row of `map` (the lengths may vary since the number of significant bits per pixels is band-dependent).

The maximum and minimum `fpixel_t` values for each band may be accessed with:

```
fpixel_t *fmin, *fmax;

fmin = fpfmin(fd);
fmax = fpfmax(fd);
```

where the minimum value for band *band* is in `fmin[`*band*`]`, etc. This information can be useful for algorithms that need to know the range of input values before doing any processing.

The actual input and output are handled by:

```
fpixel_t *buf;
int       n_do, n_done;
...

n_done = fpvread(parm.i_fd, buf, n_do);

n_done = fpvwrite(parm.o_fd, buf, n_do);
```

The calling sequences are identical to pvread and pvwrite. As with those functions, the units of input and output are pixel vectors, and either a one- or two-dimensional array may be used as the I/O buffer.

6.2.5. Error handling

IPW error handling is simple and straightforward. Since C provides no run-time error checking, IPW code must explicitly check the result of any operation that could possibly fail, and call an error-handling function if failure is detected. Only synchronous errors are handled: there is no IPW strategy for dealing with asynchronous errors (signals), nor for restarting failed operations

When an error is detected, it may be either reported immediately, or logged for future reporting (“deferred”). If immediate reporting is desired, then one of the following functions is called:

- warn
- error
- bug

These functions use their parameters, and some internal state information maintained by IPW, to construct an error message, which is written to the standard error output. bug accepts a single string parameter, while warn and error accept printf-style variable-length parameter lists (the first parameter is always a format string, and subsequent parameters, if present, must agree in type and number with the %-keys in the format string.)

The function macros:

- CHECK
- REQUIRE

may be used instead of bug in the specific situations described in §6.2.5.4.

Except for warn, all of the functions and function macros listed so far cause program termination with a nonzero exit status once the appropriate message has been written.

If it is desirable to log as much error information as possible, but postpone calling one the functions listed above (perhaps to return to a higher level in the calling hierarchy), then one or more of the following functions may be called:

- syserr
- uferr

- `usrerr`

These functions save system, file, and user-supplied error information. Program execution continues, but the saved information will appear in the next error message.

6.2.5.1. Warnings

Warnings are issued to report non-obvious program actions that do not require program termination. A typical situation requiring a warning is when a program defaults a value that would normally be obtained from a missing input image header. For example, `gradient` normally determines the grid spacing of the input image from its `geo` geodetic header; if no such header is present, then a default grid spacing is assumed. The user of `gradient` might not be aware that the input image lacks a `geo` header, or that `gradient` expects one, so a warning is issued:

```
if (!got_hdr(h_geo)) {
        warn("Elevation file has no GEOH, spacing set to 1.0");
}
```

produces the following message:

```
gradient: WARNING:
        Elevation file has no GEOH, spacing set to 1.0
```

Note that newlines are supplied automatically when the message is written; they need not appear in the format string parameter.

Warnings should **not** be used to report program actions based on defaults for missing command-line arguments, since such defaults should be obvious from both the program's usage message and its on-line documentation.

6.2.5.2. Errors

`error` is called when an external condition outside the program's specification (nonexistent file, unacceptable image attributes, incorrect command line arguments, etc.) requires program termination. Such conditions are always user-correctable; i.e., `error` should only be called for condition that could be mitigated by reissuing the command with different inputs, outputs, or arguments. A typical call would be:

```
fd = uropen(filename);
if (fd == ERROR) {
        error("can't open \"%s\"", filename);
}
```

which would print:

```
program: ERROR:
        can't open "filename"
        (UNIX error is: No such file or directory)
```

(The "`UNIX error`" message is explained in §6.2.5.5.)

6.2.5.3. Bugs

An IPW bug is an error in program logic that is detected by a test implanted in the source code. These tests are designed to catch "can't happen" situations; e.g., the `default` branch of a `switch` for which the `case`s account for all possible values of the `switch` condition, or a result that is obviously wrong even though all input values are correct. Bugs by definition represent a failure of the program to perform as specified, and can only be fixed by modifying the source code. For example:

```
sin_slope = do_slope();
if (sin_slope < 0.0 || sin_slope > 1.0) {
        bug("impossible slope");
}
```

prints:

```
program: BUG:
        impossible slope
        (file "filename", line number)
        (command line: program ...)
```

The output generated by `bug` includes the source file name and line number in which `bug` was called, and a copy of the program's command line; this information is often useful to a programmer attempting to reproduce the bug.

6.2.5.4. Assertion violations

Two special cases of a `bug` message occur frequently enough that macros are provided to handle them explicitly. These are **assertion violations**, which occur when a logical statement of a fundamental property of the program (i.e., an assertion) is found to be false. Assertions are useful checkpoints, both in terms of documenting the underlying assumptions of a program, and for guaranteeing that the program terminates if these assumptions are violated.

In IPW, assertions are checked with the `CHECK` macro. `CHECK`'s single parameter is logical expression which, if false, triggers a bug condition:[27]

```
sin_slope = do_slope();
CHECK(sin_slope >= 0.0 && sin_slope <= 1.0);
```

produces the following output:

```
program: BUG:
        Assertion "sin_slope >= 0.0 && sin_slope <= 1.0" failed
        (file "filename", line number)
        (command line: program ...)
```

Compare this example to the one in §6.2.5.3 — the code is certainly cleaner, and the actual test that failed is reproduced in the error message.

An even more specific bug occurs when a library function called with invalid parameters. This constitutes a **precondition violation** since valid parameters are a precondition for a function's successful execution [Meyer 1988]. The failure of a function to execute properly when passed invalid parameters is not a bug in the function, it is a bug in the caller for allowing the invalid parameters to be passed. To avoid executing with invalid parameters, all IPW library functions validate their parameters with the `REQUIRE` macro. For example, the library function `uropen` must not be passed a `NULL` filename pointer:

[27] This is based on a facility introduced in Version 7 UNIX [BTL 1983].

```
int
uropen(name)
        char            *name;            /* UNIX file name        */
{
        ...
        REQUIRE(name != NULL);
        ...
}
```

If the precondition `name != NULL` is false, then the program terminates with following error output:

```
program: BUG:
        Precondition "name != NULL" violated
        (file "uropen.c", line number)
        (command line: program ...)
```

This alerts the application programmer to a probable bug in the application code, as opposed to the IPW library.[28]

Since both `CHECK` and `REQUIRE` reproduce their expression parameter verbatim in the bug message, it is important that the expression be self-documenting. Variable names should be meaningful; in the case of `REQUIRE`, they should match the parameter names used in the library function's header comment, which is used to generate the external documentation for the function.

6.2.5.5. Deferred errors

The deferred error functions allow the separation of error detection and error handling. This is often necessary since a function detecting an error may not have enough information to construct a complete error message. Calling one or more of the deferred error functions allows a function to record the available error information and then return, ‘‘deferring’’ the error-handling responsibility to the caller. Of course, a function detecting an error must still return an error indication, since the caller cannot independently detect that an error has occurred; but, the error indication need not be specific, and the caller need not attempt to ‘‘decode’’ it.

The `syserr` function notifies the error handler that a UNIX system call has returned an error indication. This causes the error handler to include a UNIX system error message in the output produced by a subsequent call to `warn`, `error`, or `bug`. To elaborate on an earlier example, a program calls `uropen` to establish byte-level read access to named UNIX file. `uropen` contains the following code:

```
fd = open(name, O_RDONLY);
if (fd == SYS_ERROR) {
        syserr();
        return (ERROR);
}
...
return (fd);
```

[28] Probable, but not certain, since IPW library functions do call each other.

If the UNIX system call `open` fails, then `uropen` calls `syserr` and returns an error indication. The return value of `uropen` must be checked by the caller:

```
fd = uropen(filename);
if (fd == ERROR) {
        error("can’t open %s", filename);
}
```

If `uropen` fails, then `error` is called, since failure to access a file is almost always correctable by the user. The resulting error message is:

```
program: ERROR:
        can’t open "filename"
        (UNIX error is: No such file or directory)
```

The parenthetical "UNIX error" message is the result of calling `syserr`.

The `uferr` function passes a UNIX file descriptor to the error handler, causing the filename associated with the descriptor to be incorporated in the next error output. The `usrerr` function saves a user-supplied error message, using the same `printf`-like syntax as `warn` and `error`.

In the following fragment, taken from the `geohread` library function, the band number of the header just ingested is compared against the valid band numbers for the current image:

```
if (band < 0 || band >= nbands) {
        uferr(fd);
        usrerr("\"%s\" header: bad band \"%d\"",
                GEOH_HNAME, band);
        return (NULL);
}
```

Functions like `geohread` that ingest optional headers are usually called by the following fragment in function `gethdrs`:

```
p->hdr = (*p->ingest) (fdi);
if (p->hdr == NULL) {
        error("can’t ingest header");
}
```

So, if the range test on `band` fails, the following error message is generated:

```
program: ERROR:
        can’t ingest header
        (File: filename)
        (IPW error is: "geo" header: bad band "band")
```

If `warn` is called after `syserr`, `uferr`, or `usrerr`, then the saved information is cleared after the message is printed.

IPW programmers are encouraged to use `uferr` and `usrerr` in their own functions. `syserr` will probably be less useful since the UNIX system calls required by IPW are already embedded in the IPW library.

6.2.6. Memory management

IPW programs use dynamically allocated memory wherever possible, to maximize performance on memory-limited systems. IPW provides its own interfaces to the UNIX and C library memory allocation functions, which should not be called directly.

IPW programs use the defined type `addr_t` to hold a generic memory address[29]. Values of this type are returned by the IPW memory allocation functions, and are expected as arguments by functions that are normally passed arrays of otherwise unspecified type. The only valid contexts for the `addr_t` type are as:

- a cast in a function argument list;
- a pointer to a buffer whose contents will not be examined.

In other words, an `addr_t` pointer may never be dereferenced.

6.2.6.1. Functions

The basic IPW memory allocation function is `ecalloc`:

```
addr_t mem;
int    nelem, elsize;
...

mem = ecalloc(nelem, elsize);
if (mem == NULL) {
        error("can't allocate %d-element array", nelem);
}
```

`ecalloc` differs from the ANSI C library function `calloc` in that its arguments are `int`s instead of `unsigned`s, which helps avoid the use of otherwise superfluous casts. `ecalloc` also supplies failure information to the IPW error handling routines.

`allocnd` may be used to allocate arrays with an arbitrary number of dimensions. Here is an example allocating a 3-dimensional integer array:

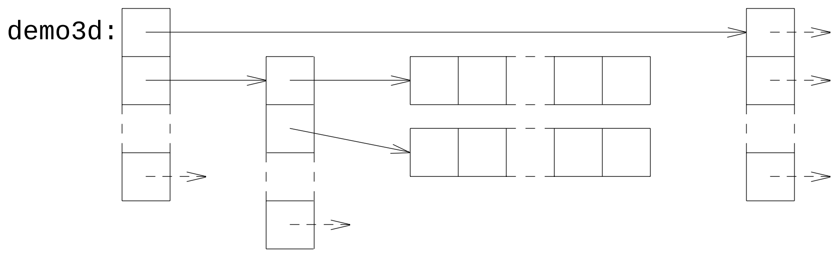
```
int ***demo3d;

demo3d = (int ***) allocnd(sizeof(int), 3, dim1, dim2, dim3);
```

The first two arguments are the size in bytes of an array element and the number of dimensions. The remaining arguments are the size of the array in each dimension. Note that the return value must be placed in a pointer typed for the correct number of levels of indirection.

The array returned by `allocnd` is indexed by ‘‘dope vectors’’:

29 This is analogous to `void *` in ANSI C.



There are two major benefits to this indexing scheme:

- the array (or any subset thereof) may be passed as an argument to a function, without the function having to specify the array's size in the argument's declaration;
- since the array data are stored contiguously, the address of the first element in the array is also the address of the entire array (this can be used to advantage by routines like `memcpy` that operate on 1-dimensional arrays).

A specialized allocation function `strdup`[30] combines the functionality of `ecalloc` and `strcpy`:

```
char *old_s, *new_s;
...

new_s = strdup(old_s);
```

`new_s` will point to a newly-allocated copy of the string pointed to by `old_s`.

6.3. SHELL SCRIPT SUPPORT

The IPW program model encourages the combination of primitives at the command level into complex applications. The ease with which command scripts may be created may give an illusion of impermanence, but in fact, many scripts wind up being as generally useful as primitives. IPW therefore includes some support for writing command scripts that are robust enough to be indistinguishable from primitives to an IPW user.

6.3.1. Skeleton

A skeleton shell script is provided in `$IPW/skel/sh/PGM.sh`. Some of the more important features of this script are described in this section.

The first line in an IPW shell script is always:

```
: ${IPW?}
```

This serves two purposes. The `:` as the first character of the script is needed on some systems to force the script to be executed by `sh` instead of `csh`[31]. The `${IPW?}` causes the script to abort if the `IPW` environment variable is not set, so that script features that depend on this variable need not incorporate separate validity checks.

The script should then contain standard IPW program headers comments[32], with

[30] This function is based on a C library function provided on some XENIX systems.

[31] See §8.4.1 for more on this problem.

[32] See §A.1.1.1.

leading ##s so they can be recognized by the `ipwman` command.

The next lines are usually:

```
PATH="$PATH:$IPW/lib"
. ipwenv
```

The script's command search path must include `$IPW/lib`, which contains the script support functions described in the next section. The file `$IPW/lib/ipwenv` initializes system-dependent environment variables.

There is some standard "boiler-plate" for dealing with command-line arguments, to help standardize the script's user interface. The UNIX command `getopt` is used to regularize the command line, and the IPW command `usage` (described in the next section) generates a standard help message if the command line is incorrect. The following example is taken from the `mklut` command:

```
optstring='i:o:k:'
synopsis='[-i in_nbits] [-o out_nbits] [-k bkgd]'
description='make look-up table'

set - `getopt "$optstring" $* 2>/dev/null` ||
        exec usage $0 "$synopsis" "$description"
```

Once the command line has been standardized by `getopt`, it is parsed with a `while` loop. The argument "`--`" is supplied by `getopt` to mark the end of the options. The use of `while` and `shift`s leaves the operands intact, for use by subsequent programs invoked by the script. The following example is also from `mklut`:

```
while :; do
        case $1 in
        --)     shift
                break
                ;;
        -i)     ibits=$2
                shift
                ;;
        -o)     obits=$2
                shift
                ;;
        -k)     const=$2
                shift
                ;;
        *)      exec sherror $pgm '"getopt" failed'
                ;;
        esac
        shift
done
```

The remainder of a script is wholly dependent on the particular application. Existing script in `$IPW/bin` should be examined by anyone developing a new IPW script.

6.3.2. Usage, sherror

In addition to object libraries and command-specific data files, `$IPW/lib` contains programs that are intended to be invoked by shell scripts, rather than directly by IPW users. Two of these are `usage` and `sherror`, which are intended to standardize the error messages generated by a shell script.

`usage` is invoked as:

```
usage command-name "synopsis" "description"
```

command-name is always obtainable from the positional parameter `$0`. The *synopsis* and *description* strings should be the same as those in the `SYNOPSIS` and `NAME` sections of the header comments. For example:

```
usage $0 '[-i in_nbits] [-o out_nbits] [-k bkgd]' \
         'make look-up table'
```

produces the following output:

```
command -- make look-up table

Usage: command [-i in_nbits] [-o out_nbits] [-k bkgd]
```

`usage` is usually `exec`'d so that it will cause the program to exit.

The `sherror` command produces a standardized error message. It may be invoked with either 2 or 3 arguments. The command:

```
sherror $0 "can't write image to a terminal"
```

produces

```
command: ERROR:
         can't write image to a terminal
```

If a third parameter is supplied then it is assumed to be a file name:

```
sherror $0 "can't open" file
```

produces:

```
command: ERROR:
         File "file": can't open
```

6.4. DEVELOPMENT TOOLS

IPW relies heavily on standard UNIX program development tools, including:

- `make`, for coordinating program compilation and installation;
- `lint`, for checking the portability and type-correctness of C source code;
- RCS, for maintaining multiple revisions of IPW source files[33].

[33] RCS, though currently not part of any commercial UNIX system, is freely available in source form.

IPW provides initialization files and shell script ‘‘wrappers’’ for these utilities to simplify their use by IPW application programmers.

Perhaps the most useful tools provided to an IPW programmer are the large body of existing header files and library functions, all available online in source as well as object form. Any IPW programmer should refer regularly to the contents of `$IPW/h` (for header files) and `$IPW/src/lib/libipw` (for the `libipw` library routines).

6.4.1. ipwmake

All IPW script, programs, and libraries are compiled and installed using `ipwmake`, which is an IPW-specific ‘‘front-end’’ for the UNIX `make` command [Feldman 1979]. `ipwmake` standardizes the vast majority of options and rules normally required in a `Makefile`; instead, the programmer supplies a far simpler `Makedefs` file, which `ipwmake` incorporates into a much more elaborate, invisible `Makefile` that is actually fed to the `make` command.

Most of the information needed by `ipwmake` in order to construct a `Makefile` is kept in a set of configuration files in the directory `$IPW/lib/make` The configuration files are usually tailored for a specific IPW installation once, by the IPW administrator[34], and do not normally need to be accessed by an IPW application programmer. However, application programmers should be aware of the system dependencies accounted for in these files, so they do not build redundant complexity into their own `Makedefs` files. System dependencies handled by the configuration files include:

- all path name dependencies (e.g., destination directories, paths to `#include` files, etc.);
- appropriate compiler options for optimization, profiling, and debugging;
- presence or absence of specific hardware (e.g., MC6888x math coprocessor on Motorola MC680x0 systems).

Given the amount of ‘‘canned’’ information in `$IPW/lib/make`, an actual `Makedefs` file is quite simple. Here is a sample `Makedefs` file for the `mstats` command:

```
PGM=    mstats
OBJS=\
        accum.o headers.o init.o main.o mcov.o mstats.o parm.o

SRCS=\
        accum.c headers.c init.c main.c mcov.c mstats.c parm.c

default: pgm
install: install-pgm

$(OBJS): pgm.h

# $Header: Makedefs,v 1.1 90/02/16 15:31:25 frew Exp $
```

This represents the minimum information required in a `Makedefs` file for a compiled program (i.e., primitive). The macro definitions should be self-explanatory; they indicate the program name, constituent object files, and constituent source files,

[34] See §8.5.

respectively.

The `default` target indicates the action to perform if `ipwmake` is invoked without a command-line target. Possible actions for the `default` target are:

`lib`: create or update an object library from C sources;

`pgm`: compile and link a primitive from C sources;

`script`: install a shell script.

The `install` target indicates the action to perform if `ipwmake` is invoked with the command-line target `install`. The possible actions are the same as for `default`, with the phrase `install-` prepended. Other standard targets, that need not be specified in `Makedefs`; include:

`clean`: delete any files that `ipwmake` knows how to rebuild (e.g., `$(OBJS)`).

`lint`: run `lint`, with appropriate options, on `$(SRCS)`.

By default, when `ipwmake` compiles source files or links programs, it uses a standard set of optimization options (e.g., `-O`). This may be changed by the following command-line options to `ipwmake`:

`-D`: compile sources with debugging options, and link programs with debugging libraries;

`-P`: compile sources with profiling options, and link programs with profiling libraries.

Skeletal `Makedefs` files for primitives, shell scripts, and libraries may be found in subdirectories of `$IPW/skel`.

6.4.2. ipwlint

IPW programmers are encouraged to use `lint` [Johnson 1978] to verify their C source code. There are two important reasons for this:

- Since C normally does no intermodule type checking, use of `lint` is the only way to guarantee that IPW library functions are being called correctly.
- Source-level portability is an important IPW design goal, and `lint` catches many nonportable constructs that a C compiler might accept.

The `ipwlint` command, or the `lint` target in `ipwmake`, are used to run `lint` on IPW source code, using the appropriate local `lint` options and libraries. The `-D` option to either of these commands invokes a more-than-usually strict set of `lint` options.

Users of `lint` know that some messages it generates are inherently superfluous, such as complaining about casting pointers returned by memory allocation functions. In IPW source code, statements known to generate such messages should be preceded by a `/*NOSTRICT*/` comment (see §A.2). Such a comment was specified in the original description of `lint` to have the effect of disabling type checking for the subsequent expression, although to our knowledge, this feature was never implemented in any standard version of `lint`. Nonetheless, it is an acceptable way to indicate to human readers of IPW source code that the author was aware of the alleged nonportability of a particular construct, and has determined that the `lint` warning should be ignored.

The IPW administrator has the option of installing `lint` libraries for each IPW function library. If these libraries exist they will be consulted. It should be noted, however, that the procedures for creating `lint` libraries are inherently nonportable[35],

[35] Some may find this ironic, since `lint` is intended to promote portability ...

and may not have been implemented at any given IPW installation. As ANSI-conforming C compilers become more widespread, IPW will eventually be converted to use ANSI C function prototypes [ANSI 1989] , which will obviate the need for `lint` libraries.

6.5. COMMENTARY

The programmer's interface to IPW reflects two basic aspects of the software's structure. First, the overall design of IPW at the module level has been **data-driven**, in the sense that the packaging of the functions reflects the flow of image data through an IPW program. The `uio`, `pixio`, and `fpio` function packages implement successive layers of processing on the binary pixel data. A similar hierarchy exists for the image header data, corresponding to text lines, then parsed header tokens, and finally a filled-in data structure for each ingested header.

Secondly, the programmer's interface is strongly influenced by each IPW command being a stand-alone UNIX program. This requires some additional effort on the part of the programmer to set up a new command, when compared against the effort required to add a new function to a self-contained, monolithic software system. A good example of this extra effort is the initialization of the structures in the `main` function that specify the command line arguments expected by the program, and the subsequent function calls required to obtain these arguments.

CHAPTER 7: PROGRAMMER'S MANUAL

This chapter contains terse but complete references for all IPW library routines and header files used by IPW application programmers. Documentation is also included for IPW commands used exclusively to support program development. Taken together, the features described in this chapter constitute the primary programmer interface to IPW.

7.1. PROGRAMMER COMMANDS

This section contains copies of the on-line documentation for each IPW command used to support program development. The format of this documentation is described in §5.2 and §A.1.1.1.

NAME

atob -- convert text integers to binary

SYNOPSIS

```
atob [-1] [-2] [-4] [file ...]
```

DESCRIPTION

atob reads text integers from {file} (default: standard input) and writes their binary equivalents on the standard output. If more than one {file} are specified, then they are read in sequence.

OPTIONS

-1	Convert input to 1-byte integers (C type: char).
-2	Convert input to 2-byte integers (C type: short).
-4	Convert input to 4-byte integers (C type: long).

Exactly one of -1, -2, or -4 must be specified.

DIAGNOSTICS

{file}: bad input

An input value could not be converted to an integer.

EXAMPLES

To convert 256 text integers to an IPW lookup table:

```
atob -1 | mkbih -l 1 -s 256 -y 1
```

SEE ALSO

IPW: btoa

NOTES

atob is not an IPW program, but is provided with IPW for use in shell scripts.

Negative input values, or input values too large to be represented in the specified number of output bytes, will be converted to binary in a machine-dependent fashion.

atob will be renamed text2bin.

NAME

btoa -- convert binary integers to text

SYNOPSIS

```
btoa [-1] [-2] [-4] [file ...]
```

DESCRIPTION

btoa reads binary integers from {file} (default: standard input) and writes their text equivalents on the standard output, one value per line. If more than one {file} are specified, then they are read in sequence.

OPTIONS

-1	Convert 1-byte input integers (C type: char).
-2	Convert 2-byte input integers (C type: short).
-4	Convert 4-byte input integers (C type: long).

EXAMPLES

To print the first 200 quantized pixel values from an IPW image with 2-byte pixels:

```
rmhdr | btoa -2 | head -100
```

SEE ALSO

IPW: atob, rmhdr

NOTES

btoa is not an IPW program, but is provided with IPW for use in shell scripts.

Input values are assumed to be unsigned.

btoa will be renamed text2bin.

NAME

install -- install commands, libraries, etc.

SYNOPSIS

```
install [-c] [-m mode] [-o owner] [-g group] [-s]
        file destination
```

DESCRIPTION

install moves (or copies: see option "-c" below) {file} to {destination}, optionally changing its owner, group, or protection. If {destination} is a directory, {file} will be placed in that directory.

OPTIONS

-c "cp" {file} to {destination} (default: use "mv").

-m Set protection of installed file to {mode} (default: same as {file}).

-o Set owner of installed file to {owner} (default: same as {file}).

-g Set group of installed file to {group} (default: same as {file}).

-s Strip symbol table from installed file (default: don’t). This only makes sense if {file} is a compiled executable program.

EXAMPLES

(install is almost always invoked indirectly via ipwmake.)

SEE ALSO

IPW: ipwmake

UNIX: chgrp, chown, cp, install, mv, strip

NOTES

install is a substitute for a UNIX command of the same name, that often requires special privileges to use.

The -o option may fail unless the invoker has "root" privileges.

The -g option may fail unless the invoker is a member of "group". On some UNIX systems, the group of the installed file will automatically be changed to that of the destination directory.

IPWLINT **IPWLINT**

NAME

ipwlint -- run lint on specific IPW source files

SYNOPSIS

```
ipwlint [options ...] file ...
```

DESCRIPTION

ipwlint runs the UNIX C source code checker "lint" on the specified IPW source {file}s.

The OPTIONS, DIAGNOSTICS, and FILES are the same as for ipwmake.

EXAMPLES

SEE ALSO

IPW: ipwmake

UNIX: lint, make

NOTES

The use of ipwlint is discouraged -- it is much safer to use the command:

```
ipwmake [options ...] lint
```

which checks all of the source files mentioned in the "Makedefs" file in the current directory.

Lint’s behavior varies wildly across UNIX implementations. Many lint diagnostics are totally spurious, and the absence of lint diagnostics is no guarantee of portability.

NAME

ipwmake -- IPW "make" command

SYNOPSIS

```
ipwmake [-D] [-P] [option ...] [target ...]
```

DESCRIPTION

ipwmake is a front-end for the UNIX "make" command.

ipwmake constructs a makefile for an IPW program or library from generic files in the directory $IPW/lib/make, plus a minimum of specific information in a "Makedefs" file in the current directory. The make command is then invoked with this makefile, and any {option}s and {target}s specified on the ipwmake command line.

The makefile created by ipwmake always contains the following targets:

default	Build the command or library whose source is in the current directory. This is the default target if none is specified.
install	Move the command or library to its destination directory.
clean	Delete any files in the current directory that can be reconstructed from source files.

In directories containing C source files, the "lint" target will run the lint utility on those files, sending the diagnostics to the standard output.

There are other standard targets which are less commonly used; see $IPW/lib/make/rules.

If the environment variable MAKEDEFS is set, then the file it names is used in place of the default "Makedefs" file. In particular, setting MAKEDEFS to "-" causes ipwmake to read control information from its standard input.

```
OPTIONS
        -D      Compile target for debugging: use debugging options
                when compiling and linking, and link against IPW
                libraries compiled with these options.

        -P      Compile target for profiling: use profiling options
                when compiling and linking, and link against IPW
                libraries compiled with these options.

        Only one of -D or -P may be specified.  If neither are
        specified, the default compile options usually invoke an
        optimizer, and the default libraries are usually those whose
        members were compiled with an optimizer.

DIAGNOSTICS
        no Makedefs file

                There is no "Makedefs" file in the current
                directory.

FILES
        ./Makedefs

                specific information about targets in the current
                directory.

        $IPW/lib/make/local     locally-tuned macros
        $IPW/lib/make/std       standard macros
        $IPW/lib/make/debug     debugging macros
        $IPW/lib/make/profile   profiling macros
        $IPW/lib/make/rules     rules

        $IPW/skel/lib/Makedefs
        $IPW/skel/pgm/Makedefs
        $IPW/skel/sh/Makedefs

                skeleton "Makedefs" files for object libraries,
                executable programs, and shell scripts.
```

EXAMPLES

The following is the "Makedefs" file for the "mux" command:

```
PGM=mux
OBJS=\
        main.o mux.o muxhdrs.o muximg.o

SRCS=\
        main.c mux.c muxhdrs.c muximg.c

default: pgm
install: install-pgm

$(OBJS): $(PGM).h
```

Typing

```
ipwmake install clean
```

in the directory containing this file will:

- create the object files main.o, mux.o, muxhdrs.o, and muximg.o, by individually compiling the corresponding source files;

- link the object files with the appropriate IPW and UNIX libraries to create the executable file "mux";

- move "mux" to the default IPW executable directory (usually $IPW/bin), strip its symbol table, and set appropriate permissions;

- delete the object files.

SEE ALSO

IPW: ipwlint, objs

UNIX: make

NOTES

Only a sampling of ipwmake’s capabilities can be given here. The generic control files in $IPW/lib/make, and the skeleton files $IPW/skel/*/Makedefs, should be consulted for further information.

NAME

mc -- multi-column output filter

SYNOPSIS

```
mc [-w width] [file ...]
```

DESCRIPTION

mc reads text lines from {file} (default: standard input) and writes them in multiple columns on the standard output, in column-major order (i.e. the first input lines appear in the first column of output). If multiple {file}s are specified, then they are read in sequence.

OPTIONS

-w	Output lines shall contain no more than {width} characters (default: 80, or the width of the display, if the output is a terminal device). The terminating newline is counted as 1 character.

DIAGNOSTICS

{width}: bad width

The specified {width} is less than 8 columns.

{inwidth}-col input too wide for {outwidth}-col output

An input line is too wide for the specified (or default) output width.

EXAMPLES

To generate a multi-column listing of the files in the current directory:

```
ls | mc
```

SEE ALSO

UNIX: ls

NOTES

mc is not an IPW command, but is provided with IPW for use in shell scripts.

mc was originally written (and posted to USENET) by Dan Ts’o, Dept.of Neurobiology, Rockefeller Univ.

NAME

```
        srcs -- show potential IPW source files
        objs -- show potential IPW object files
```

SYNOPSIS

```
        srcs [dir ...]
        objs [dir ...]
```

DESCRIPTION

srcs displays on the standard output the names of the IPW source files in the specified directories (default: current directory).

objs displays the same file names, but with a ".o" suffix instead of the source language suffix (e.g., ".c").

These commands are typically invoked while editing an ipwmake "Makedefs" file, to initialize the "SRCS" and "OBJS" macros.

FILES

```
        *.[cfF]
                possible source files

        $IPW/skel/lib/Makedefs
        $IPW/skel/pgm/Makedefs
```

skeleton "Makedefs" files, containing instructions for the use of the srcs and objs commands from within the vi editor.

EXAMPLES

The command sequence:

```
                cd $IPW/src/bin/mux
                objs
```

displays:

```
                main.o mux.o muxhdrs.o muximg.o
```

SEE ALSO

IPW: ipwmake

UNIX: sed

NOTES

7.2. LIBRARY FUNCTIONS

This section contains copies of the on-line documentation for the IPW library functions most often used in application programs. The format of this documentation is described in §5.2 and §A.1.1.2.

Certain of the image header manipulation routines are similar enough that they have been documented in terms of a single routine pertaining to an imaginary ‘‘XX’’ header. Documentation for such routines should be understood to apply to a separate instance of the routine for each supported IPW header type; e.g., the documentation for `xxhdup` applies to the functions `bihdup`, `lqhdup`, etc.

All functions that accept file descriptor arguments cause assertion violations if the value of the file descriptor is outside the valid range for the host environment. Functions that perform I/O cause assertion violations if the file accessed by the file descriptor has not be opened for reading or writing, as appropriate.

ADDSV **ADDSV**

NAME

addsv, delsv, dupsv, walksv -- manipulate string vectors

SYNOPSIS

```
STRVEC_T *addsv(p, s)
STRVEC_T *p;
char *s;

STRVEC_T *delsv(p, i)
STRVEC_T *p;
int i;

STRVEC_T *dupsv(p)
STRVEC_T *p;

char *walksv(p, reset)
STRVEC_T *p;
bool_t reset;
```

DESCRIPTION

These routines manipulate string vectors. A string vector is a linked list of arbitrary-length character strings.

addsv appends a copy of the EOS-terminated string pointed to by {s} to the string vector pointed to by {p}. If {p} is NULL, then a new string vector is created with {s} as its first string.

delsv deletes the {i}'th string from the string vector pointed to by {p}.

dupsv duplicates the string vector pointed to by {p}.

walksv sequentially accesses the strings in the string vector pointed to by {p}. If {reset} is TRUE, then the first string is accessed. If {reset} is FALSE, then the next string is accessed.

RETURN VALUE

```
success:  pointer to the new, current, or duplicate string
          vector, or to the current string.

failure:  NULL
```

APPLICATION USAGE

The most frequent use of string vectors in application programs is to pass an arbitrary number of strings via a single argument to a library function. For example, the following string vector:

```
        STRVEC_T *annot;
        ...
        annot = addsv((STRVEC_T *) NULL,
                      "image analyzed by A. Hacker");
        annot = addsv(annot,
                      "showing extent of medfly damage");
```

could be passed to the bihmake function, causing both strings to appear as separate annotation strings in the newly created basic image header.

SEE ALSO

IPW: bihmake

NOTES

delsv does not return an error indication if the {i}’th string doesn’t exist.

Interspersing calls to addsv or delsv with calls to walksv may confuse walksv’s notion of what the "next" string is. Calling walksv with {reset} == TRUE always works.

ALLOCND **ALLOCND**

NAME

allocnd --- allocate a multidimensional array

SYNOPSIS

```
addr_t allocnd(elsize, ndim, dim1, ..., dimn)
int elsize, ndim, dim1, ..., dimn;
```

DESCRIPTION

allocnd dynamically allocates an {ndim}-dimensional array, where sizeof(each element) is {elsize}. The particular dimensions are given by the {dim1}, ..., {dimn} arguments. The returned address should be cast into a pointer to the proper type of object.

The dynamically allocated array is indexed via dope vectors (i.e., all but the rightmost dimension are implemented by arrays of pointers).

All space allocated by allocnd is initialized to 0. The memory occupied by array data (as opposed to the dope vectors) will be logically contiguous, so the address of the first element in the array may also be treated as the base address of a single 1-dimensional array.

RETURN VALUE

success: a pointer to the newly-allocated array

failure: NULL

ERRORS

size overflow during allocation

The total number of data bytes in the array could not be expressed as an int.

APPLICATION USAGE

Use allocnd() as a general-purpose multidimensional array allocator. For example:

```
double  a[3][4];
double  **b;

b = (double **)allocnd(sizeof(double), 2, 3, 4);
```

References of the form b[i][j] will behave equivalently to a[i][j].

SEE ALSO

IPW: ecalloc

UNIX: malloc

NOTES

Because of the extra space required for the dope vectors, arrays allocated by allocnd() will consume more memory than equivalently-dimensioned arrays allocated statically.

NAME
        bihmake -- make a basic image (BI) header

SYNOPSIS
```
        #include "bih.h"

        BIH_T *bihmake(nbytes, nbits, history, annot, bihp,
                       nlines, nsamps, nbands)

        int nbytes, nbits, nlines, nsamps, nbands;
        STRVEC_T *history, *annot;
        BIH_T *bihp;
```

DESCRIPTION
        bihmake allocates a single-band basic image (BI) header.
        The header is initialized from bihmake’s arguments:

{nbytes}	number of bytes per pixel. If {nbytes} is zero, it will be made just large enough to contain {nbits}.
{nbits}	number of bits per pixel. If {nbits} is zero, it will be set to ({nbytes} * CHAR_BIT).

        (i.e., {nbytes} and {nbits} cannot BOTH be zero.)

{history}	pointer to a vector of strings to be used as history records in this header.
{annot}	pointer to a vector of strings to be used as annotation in this header.
{bihp}	pointer to another BI header. If {bihp} is non-null, then the per-image component of *{bihp} will be linked into the new header.

        {nlines}, {nsamps}, and {nbands} are ignored if {bihp} is
        non-null.

{nlines}	number of lines per image
{nsamps}	number of samples per image line
{nbands}	number of bands per image sample

RETURN VALUE

```
        success:  pointer to new BI header

        failure:  NULL
```

SEE ALSO

```
        IPW: addsv, mkbih, xxhmake
```

NOTES

The sharing of a "per-image" component is unique to the BI header -- all other IPW headers are strictly per-band.

NAME
        boimage -- mark end of headers on output image

SYNOPSIS
```
        #include "hdrio.h"

        int boimage(fd)
        int fd;
```

DESCRIPTION
        boimage writes an "image preamble" to the output image
        accessed via file descriptor {fd}, which marks the
        transition from header to pixel data.

RETURN VALUE
        success: OK

        failure: ERROR

APPLICATION USAGE
        You must call boimage after all headers, and before any
        pixels, are written to an output image.

SEE ALSO
        IPW: gethdrs, skiphdrs, xxhwrite

NOTES
        boimage is currently implemented as a macro.

NAME
```
        dtoa, ftoa, itoa, ltoa -- convert numbers to strings
```

SYNOPSIS
```
        char *dtoa(s, n)
        char *s;
        double n;

        char *ftoa(s, n)
        char *s;
        float n;

        char *itoa(s, n)
        char *s;
        int n;

        char *ltoa(s, n)
        char *s;
        long n;
```

DESCRIPTION
```
        These functions write the ASCII representation of {n},
        followed by an EOS (’ ’), into the character array pointed
        to by {s}.
```

RETURN VALUE
```
        {s}
```

SEE ALSO
```
        UNIX: sprintf
```

NOTES
```
        The array pointed to by {s} must be large enough to hold the
        ASCII representation of {d}, plus a trailing EOS.  The
        consequences of overflowing {s} are unpredictable.
```

NAME

ecalloc -- memory allocator

SYNOPSIS

```
addr_t ecalloc(nelem, elsize)
int nelem, elsize;
```

DESCRIPTION

ecalloc is the IPW interface to the standard C library function calloc.

ecalloc returns a pointer to enough contiguous memory to hold {nelem} {elsize}-byte objects. The memory is initialized to zeros.

RETURN VALUE

success: pointer to allocated memory.

failure: NULL

APPLICATION USAGE

IPW programs should call ecalloc wherever they would otherwise call malloc or calloc.

SEE ALSO

IPW: allocnd, usrerr

UNIX: malloc

NOTES

ecalloc is a "wrapper" around the calloc function that:

- verifies that {nelem} and {elsize} are both > 0.

- makes system error information available to the IPW error function if the allocation fails (the programmer must still call error explicitly).

NAME
        error, warn -- IPW error handlers

SYNOPSIS
```
        void error(format, ...)
        char *format;

        void warn(format, ...)
        char *format;
```

DESCRIPTION
        These functions print the error message described by the printf-style {format}, and optional additional arguments, on the standard error output. If usrerr has been called, its argument is also printed. If syserr has been called, a UNIX system error message is also printed.

        error causes program termination after the error messages have been printed.

APPLICATION USAGE
        error is the most common IPW error handler. A typical usage would be:

```
                fd = uropen(filename);
                if (fd == ERROR) {
                        error("can’t open %s", filename);
                }
```

        which would produce the following standard error output:

```
                {program}: ERROR:
                        can’t open "{filename}"
                        (UNIX error is: {perror-message})
```

        and then cause the program to terminate.

        warn is used to notify the user of unusual conditions that do not affect program execution, but may produce unexpected output; e.g., an insufficient number of bits per input pixel to satisfy the precision requirements of a particular algorithm.

SEE ALSO
        IPW: ipwenter, usrerr

        UNIX: errno, perror

NOTES

These functions both call the low-level function _error, which does the actual error message formatting and output.

NAME

fpmap, fpmaplen, fpfmax, fpfmin --
access to floating-point conversion parameters

SYNOPSIS

```
#include "fpio.h"

fpixel_t **fpmap(fd)
int fd;

int *fpmaplen(fd)
int fd;

fpixel_t *fpfmax(fd)
int fd;

fpixel_t *fpfmin(fd)
int fd;
```

DESCRIPTION

These functions provide access to the fpio subsystem parameters that control the conversion between integer and floating-point pixel values.

fpmap returns a pointer to an array of pointers to the type conversion arrays associated with the image accessed by file descriptor {fd}. The array indices are unsigned integer (pixel_t) pixel values, and the array elements are the corresponding floating point (fpixel_t) pixel values. There is a separate array for each image band.

fpmaplen returns a pointer to an array containing the lengths of the type conversion arrays associated with the image accessed by file descriptor {fd}. The array indices are band numbers: the value of the i’th element of the array pointed to by the value of fpmaplen is the length of the i’th array pointed to by the value of fpmap. fpmaplen is necessary since the lengths of the conversion arrays vary according to the number of significant bits per integer pixel in each band.

fpfmin and fpfmax return pointers to arrays of the maximum or minimum floating-point pixel values in the image accessed by file descriptor {fd}. The arrays indices are band numbers.

RETURN VALUE

```
        success:  pointer to the particular fpio data structure

        failure:  NULL
```

GLOBALS ACCESSED

```
        _fpiocb[fd]     fpio control block for file descriptor {fd}
```

APPLICATION USAGE

Application programs typically use these functions to modify the existing conversion(s) between integer and floating-point pixel values.

SEE ALSO

IPW: fpvread, lqhmake, lqhx, mklqh, mnxfp

NOTES

If these functions are called BEFORE any LQ headers are read from or written to {fd}, then they will cause the default (1:1) conversion between integer and floating-point pixel values to be established, which will be UNAFFECTED by any subsequent LQ header I/O on {fd}.

NAME

fpvread, fpvwrite -- floating-point pixel I/O

SYNOPSIS

```
#include "fpio.h"

int fpvread(fd, buf, npixv)
int fd, npixv;
fpixel_t *buf;

int fpvwrite(fd, buf, npixv)
int fd, npixv;
fpixel_t *buf;
```

DESCRIPTION

These routines are analogous to pvread and pvwrite, except that instead of operating on integer pixel values, they represent pixel values as the floating-point type fpixel_t.

fpvread reads {npixv} floating-point pixel vectors from the image accessed by file descriptor {fd} into the buffer pointed to by {buf}.

fpvwrite writes {npixv} floating-point pixel vectors from the buffer pointed to by {buf} to the image accessed by file descriptor {fd}.

If an LQ header has been read from or written to {fd}, then the pixel values are converted between integer and floating-point representations according to the mapping specified in the LQ header. Otherwise, the conversion is 1:1; i.e., an integer pixel value of 123 is mapped into a floating-point value of 123.0, and vice versa.

RETURN VALUE

success:	number of floating-point pixel vectors read or written
failure:	ERROR

GLOBALS ACCESSED

_fpiocb[fd]	fpio control block for file descriptor {fd}

APPLICATION USAGE

It is often necessary for applications to treat pixels as floating-point quantities, either because they represent some "real world" quantity (e.g., elevation, solar irradiance, etc.), or because an algorithm is more easily implemented for floating-point than for integer quantities.

Note that the application can control the mapping between integer and floating-point values by whether it allows an LQ header to be read from or written to a particular file descriptor.

SEE ALSO

IPW: fpmap, lqhmake, lqhx, mklqh, mnxfp, pvread, uread, uropen

NOTES

It is important to remember that the {npixv} argument to these functions is the number of floating-point pixel VECTORS to be read or written, and that the corresponding number of INDIVIDUAL PIXELS is {npixv} * (number of bands per image sample).

Calls to fpio, pixio, and uio I/O functions should not be intermixed on the same file descriptor.

NAME

frand, frinit -- uniform random number generator

SYNOPSIS

```
float frand()

void frinit()
```

DESCRIPTION

frand returns a uniform random number from the range [0,1).

frinit initializes frand’s random number generator with a "seed" derived from the current system time-of-day. If frinit is not called, then successive calls to frand will return the same "random" sequence for each program invocation.

RETURN VALUE

the next random number

SEE ALSO

UNIX: random

NAME
        geohmake -- make a geodetic (GEO) header

SYNOPSIS
```
#include "geoh.h"

GEOH_T *geohmake(bline, bsamp, dline, dsamp, units, csys)
double bline, bsamp, dline, dsamp;
char *units, *csys;
```

DESCRIPTION
        geohmake allocates a single-band geodetic (GEO) header. The header is initialized from geohmake's arguments:

{bline}	geodetic coordinate of image line 0
{bsamp}	geodetic coordinate of image sample 0
{dline}	geodetic distance between image lines
{dsamp}	geodetic distance between image samples
{units}	units in which geodetic coordinates are expressed (e.g., "meters").
{csys}	name of geodetic coordinate system (e.g., "UTM").

RETURN VALUE
        success: pointer to new GEO header

        failure: NULL

SEE ALSO
        IPW: mkgeoh, window, xxhmake

NOTES
        There are not yet any standard values for {units} and {csys}.

NAME

gethdrs -- process image headers

SYNOPSIS

```
#include "gethdrs.h"

void gethdrs(fdi, request, nbands, fdo)
int fdi, nbands, fdo;
GETHDR_T **request;
```

DESCRIPTION

gethdrs provides a high-level interface to the IPW xxhread routines. Image headers are read from file descriptor {fdi} and written to file descriptor {fdo}, subject to the interpretation of {request} and {nbands}.

{request} points to an array of pointers to GETHDR_T structures. The last element in the array must be a null pointer.

Each GETHDR_T structure specifies the disposition of a particular type of image header. The first two fields of the structure should be initialized to the name of the header to be processed, and a pointer to the header's ingest routine. If the ingest routine pointer is NULL, then the header will be skipped; otherwise, the header will be ingested and its address will be available via the hdr_addr macro when gethdrs returns. Ingested headers are not written to {fdo}.

Headers that are not mentioned in the structures pointed to by {request}, and that pertain to band numbers less than {nbands}, are copied to file descriptor {fdo}. You must therefore read the BI header from {fdi}, and write a BI header to {fdo}, before calling gethdrs.

If {nbands} is set to the macro NO_COPY, then NO headers will be written to {fdo}.

APPLICATION USAGE

In the following code fragment, we have arranged to ingest an LQ header, skip an OR header, and copy any other headers encountered to fdo. Note that we must explicitly write an LQ header if we want one to be placed in the output image. The GETHDR_T structures and the {request} array are declared static so that the may be initialized.

```
static GETHDR_T h_lqh = {LQH_HNAME,
                         (ingest_t) lqhread};
static GETHDR_T h_orh = {ORH_HNAME};
static GETHDR_T *request[] = {&h_lqh, &h_orh, NULL};
...
i_bihpp = bihread(i_fd);
...
bihwrite(o_fd, o_bihpp);
gethdrs(i_fd, request, bih_nbands(o_bihpp), o_fd);
i_lqhpp = (LQH_T **) hdr_addr(h_lqh);
...
lqhwrite(o_fd, o_lqhpp);
boimage(o_fd);
```

SEE ALSO

IPW: boimage, ipwenter, skiphdrs, xxhread, xxhwrite

NOTES

gethdrs terminates program execution with an error message if any errors are encountered.

NAME

        hbit -- find highest-order 1 bit in an integer

SYNOPSIS

```
        int hbit(arg)
        unsigned arg;
```

DESCRIPTION

        hbit finds the index (1-relative) of the highest-order 1 bit in {arg}.

RETURN VALUE

        the index of the highest-order 1 bit in {arg}. If {arg} is 0 then 0 is returned.

APPLICATION USAGE

        The value of hbit may be interpreted as the minimum number of bits needed to represent {arg}.

NAME
```
        hdralloc -- allocate memory for image header components
```

SYNOPSIS
```
        addr_t hdralloc(nelem, elsize, fd, hname)
        int n, size, fd;
        char *hname;
```

DESCRIPTION

hdralloc is a special interface to the ecalloc function, for allocating the components of an image header data structure.

Like ecalloc, hdralloc returns a pointer to enough contiguous memory to hold {nelem} {elsize}-byte objects. The memory is initialized to zeros.

Unlike ecalloc, hdralloc terminates program execution if it is unable to allocate the memory requested. Prior to termination, {hname} is passed to the usrerr function, and {fd} (if it is a valid file descriptor) is passed to the uferr function. {hname} is usually the macro XXH_HNAME, as defined in the header’s xxh.h file, which {fd} is the file descriptor of the image to which the header will be written. Pass ERROR as the value of {fd} if you want hdralloc to ignore it.

If hdralloc returns successfully, then {fd} and {hname} are ignored.

RETURN VALUE

pointer to allocated memory

ERRORS
```
        can’t allocate "{name}" header
        can’t allocate array of "{name}" header pointers
```

APPLICATION USAGE

In application programs, hdralloc is usually called to allocate an array of pointers to the headers associated with each image band. See the documentation for xxhmake for an example.

SEE ALSO

IPW: ecalloc, xxhmake

NAME

hnbytes, hnbits -- image pixel size parameters

SYNOPSIS

```
int hnbytes(fd, band)
int fd, band;

int hnbits(fd, band)
int fd, band;
```

DESCRIPTION

These functions return the number of bytes or bits in a single pixel of band {band} in the image accessed by file descriptor {fd}.

RETURN VALUE

number of bytes or bits per pixel

GLOBALS ACCESSED

_bih[fd]	BIH associated with file descriptor fd

APPLICATION USAGE

In an application program, an image’s file descriptor typically has a much broader scope than the image’s BIH pointer. Therefore, these functions allow the per-band parameters normally accessed via the BIH pointer to be accessed via the file descriptor.

SEE ALSO

IPW: hnlines, imgsize, xxhread, xxhwrite

NOTES

These functions will cause an assertion violation if they are called before a BIH is read from or written to {fd}.

NAME

hnlines, hnsamps, hnbands -- per-image parameters

SYNOPSIS

```
int hnlines(fd)
int fd;

int hnsamps(fd)
int fd;

int hnbands(fd)
int fd;
```

DESCRIPTION

These functions return the number of lines, samples per line, or bands per sample in the image accessed by file descriptor {fd}.

RETURN VALUE

number of lines, samples, or bands in the image on {fd}

GLOBALS ACCESSED

```
_bih[{fd}]      BIH associated with file descriptor {fd}
```

APPLICATION USAGE

In an application program, an image's file descriptor typically has a much broader scope than the image's BIH pointer. Therefore, these functions allow the per-image parameters normally accessed via the BIH pointer to be accessed via the file descriptor.

SEE ALSO

IPW: hnbytes, imgsize, xxhread, xxhwrite

NOTES

These functions will cause an assertion violation if they are called before a BIH is read from or written to {fd}.

NAME

horhmake -- make a horizon (HOR) header

SYNOPSIS

```
#include "horh.h"

HORH_T *horhmake(azimuth)
double azimuth;
```

DESCRIPTION

horhmake allocates a single-band horizon (HOR) header. The header is initialized from horhmake's arguments:

{azimuth}	the azimuth of the image lines, measured in radians counterclockwise from the south.

RETURN VALUE

success: pointer to new HOR header

failure: NULL

ERRORS

horhmake: abs(azm) ({azimuth}) > pi

{azimuth} is outside the range [-PI..PI].

SEE ALSO

IPW: horizon, xxhmake

IMGCOPY IMGCOPY

NAME

    imgcopy -- copy all pixel data between image files

SYNOPSIS

```
int imgcopy(i_fd, o_fd)
int i_fd, o_fd;
```

DESCRIPTION

imgcopy copies all of the pixel data from the image accessed by file descriptor {i_fd} to the image accessed by file descriptor {o_fd}.

RETURN VALUE

success: OK

failure: ERROR

ERRORS

can’t calculate input image size

> The total number of bytes in the input image won’t fit in a signed long integer.

input image larger than header indicates

> An end-of-file was NOT detected on {i_fd} after the appropriate number of bytes were copied.

APPLICATION USAGE

imgcopy is typically called by programs that add, delete, or modify an image’s headers, but just pass image data through unaltered.

SEE ALSO

IPW: imgsize, ucopy

NOTES

imgcopy will cause an assertion violation if it is called before a BIH is read from {i_fd} or written to {o_fd}.

NAME
```
        imgsize, sampsize -- size of entire image or single sample
```

SYNOPSIS
```
        #include "bih.h"

        long imgsize(fd)
        int fd;

        int sampsize(fd)
        int fd;
```

DESCRIPTION

imgsize computes the total number of image data (pixel) bytes in the image accessed by file descriptor {fd}.

sampsize computes the total number of bytes in a single sample of the image accessed by the file descriptor {fd}.

RETURN VALUE

total number of pixel bytes in the image, or in a single image sample

GLOBALS ACCESSED
```
        _bih[{fd}]      BIH associated with file descriptor {fd}
```

APPLICATION USAGE

imgsize is useful only to programs that are copying all of the pixel data from an input IPW image to an output non-IPW image, in which case imgsize can provide the {ncopy} argument required by the ucopy function. If the output image has an IPW BIH, then imgcopy should be used instead of ucopy.

sampsize is useful to programs that do not need to interpret image pixel values, but do need access to individual samples (e.g. flip, transpose, window, etc.)

SEE ALSO

IPW: hnlines, hnbytes, imgcopy, ucopy, xxhread, xxhwrite

NOTES

These functions will cause an assertion violation if they are called before a BIH is read from or written to {fd}.

The value returned by imgsize does NOT include the bytes occupied by header data; e.g., for an image with 512 lines, 512 samples, 1 band, and 1 byte per pixel, the value returned by imgsize will be 262,144 (512*512).

NAME

ipwenter -- initialize an IPW main program

SYNOPSIS

```
#include "getargs.h"

void ipwenter(argc, argv, optv, descrip)
int argc;
char **argv, *descrip;
OPTION_T *optv[];
```

DESCRIPTION

ipwenter is called to initialize an IPW main program. Its chief function is to parse the command-line arguments in {argv} The caller provides, via {optv}, descriptions of each option that the program is prepared to accept, including the number and type of any "optargs" (option arguments) allowed for each option.

The string {descrip} should be a one-line description of the function of the program; it is saved externally for future usage message generation.

ERRORS

Too numerous to list. Basically, any incorrect command line causes ipwenter to terminate program execution with an explanatory diagnostic.

GLOBALS ACCESSED

_argv	set to {argv}
_descrip	set to {descrip}
_optv	set to {optv}

These globals are used by the error handling subsystem to construct standard-format error messages.

APPLICATION USAGE

A call to ipwenter must be the first executable statement in an IPW main program. Since the calling sequence of ipwenter is rather complicated, the following example is presented.

All C programs running in a UNIX environment have access to their command line arguments via the parameters {argc} and {argv}, which are passed to the program’s main function:

```
main(argc, argv)
        int             argc;
        char          **argv;
{
        ...
```

The application programmer specifies each option that the program is prepared to accept with an "option descriptor". An option descriptor is a structure containing the following fields, in order:

field	type	default
option letter	char	(must be specified)
option description	string	(must be specified)

(The following fields may be omitted if there are no optargs.)

optarg type	macro	NO_OPTARGS
optarg description	string	(must be specified)
required flag	macro	OPTIONAL
min. number of optargs	int	1
max. number of optargs	int	(unlimited)

The option descriptors are declared static so that they may be initialized. Here are some sample option descriptors:

This option descriptor specifies the "-a" option, which takes no arguments:

```
static OPTION_T opt_a = {
        ’a’, "(option description)",
};
```

This option descriptor specifies the "-b" option, which takes at least 3 integer arguments:

```
                static OPTION_T opt_b = {
                        ’b’, "(option description)",
                        INT_OPTARGS, "(arg description)",
                        OPTIONAL, 3,
                };
```

This option descriptor specifies the "-c" option, which must be present, and which takes at least 1 real argument,

```
                static OPTION_T opt_c = {
                        ’c’, "(option description)",
                        REAL_OPTARGS, "(arg description)",
                        REQUIRED,
                };
```

This option descriptor specifies the "-d" option, which must be present, and which takes at least 1 and no more than 4 string arguments:

```
                static OPTION_T opt_d = {
                        ’d’, "(option description)",
                        STR_OPTARGS, "(arg description)",
                        REQUIRED, 1, 4
                };
```

This option descriptor specifies 0 or more string operands. The "placeholder" is the single word placed on the synopsis line in the usage message. {REQUIRED,OPTIONAL} and the {min,max} number of "arguments" (i.e. operands) are interpreted analogously to options.

```
                static OPTION_T operands = {
                        OPERAND, "(operand description)",
                        STR_OPERANDS, "(placeholder)",
                };
```

Once all of the options and operands have been specified, a null-terminated array of pointers to the option and operand descriptors must be constructed. The pointer to the operand descriptor (if any) must be the last non-null pointer in the array. Like the option descriptors, this array is declared static so that it may be initialized:

```
                static OPTION_T *optv[] = {
                        &opt_a,
                        &opt_b,
                        &opt_c,
                        &opt_d,
                        &operands,
                        0                         };

                void ipwenter(argc, argv, optv,
                              "(program description)")
```

After ipwenter returns, the options are accessible by applying macros to the option descriptors. These macros include ({opt} is an option descriptor):

got_opt(opt)	TRUE if {opt} was specified on the command line.
n_args(opt)	number of optargs associated with {opt}.
int_arg(opt, i)	integer value of the {i}’th optarg (0-relative) associated with {opt}.
int_argp(opt)	pointer to array of integer-valued optargs associated with {opt}.

There are also str, long, and real versions of the _arg and _argp macros.

SEE ALSO

IPW: gethdrs, ipwexit, opt_check

UNIX: getopt

K. Hemenway and H. Armitage, 1984. "Proposed syntax standard for UNIX system commands". UNIX World, vol. 1, no. 3, pp. 54-57.

NOTES

The calling program must NOT call UNIX library function getopt.

Non-string operands (i.e., values other than STR_OPERANDS) are not yet supported.

IPWEXIT **IPWEXIT**

NAME

ipwexit -- terminate an IPW program

SYNOPSIS

```
void ipwexit(status)
int status;
```

DESCRIPTION

ipwexit terminates the execution of an IPW program. Open uio files are flushed and closed. {status} is passed to the operating system via the UNIX system call exit.

APPLICATION USAGE

All IPW application programs should call ipwexit as their last executable statement.

SEE ALSO

IPW: ipwenter, uclose

UNIX: exit

NOTES

Ipwexit never returns.

NAME

lqhmake -- make a linear quantization (LQ) header

SYNOPSIS

```
#include "lqh.h"

LQH_T *lqhmake(nbits, nbkpts, ival, fval, units, interp)
int nbits, nbkpts;
pixel_t *ival;
fpixel_t *fval;
char *units, *interp;
```

DESCRIPTION

lqhmake allocates a single-band linear quantization (LQ) header. The header is initialized from lqhmake's arguments:

{nbits}	number of bits per pixel.
{nbkpts}	number of breakpoints in the {ival} and {fval} arrays
{ival}	array of {nbkpts} pixel values.
{fval}	array of {nbkpts} floating-point values, corresponding to the values in {ival} (e.g., {fval}[i] is the floating-point value that corresponds to the pixel value {ival}[i].)
{units}	string identifying the units in which the values in {fval} are expressed (e.g., "W mˆ-2 nmˆ-1 srˆ-1".)
{interp}	string identifying the function to used to interpolate values of {fval} for any pixel values not specified in {ival}. Possible values of {interp} are predefined in lqh.h; e.g., LQH_LIN_INTERP for linear interpolation.

RETURN VALUE

success: pointer to new LQ header

failure: NULL

ERRORS

LQ header: bad integer breakpoint: {ival[...]}

The specified breakpoint is outside the possible range of values for {nbits}-bit pixels.

SEE ALSO

IPW: lqhx, mklqh, xxhmake

NOTES

There are not yet any standard values for {units}.

LQH_LIN_INTERP is currently the only defined interpolation function.

NAME

ltof, ltoi, ltou -- long integers conversions

SYNOPSIS

```
double ltof(i)
long i;

int ltoi(i)
long i;

unsigned ltou(i)
long i;
```

DESCRIPTION

These functions normally return the value of {i}. However, if {i} cannot be EXACTLY represented in the return type, then program execution terminates with an appropriate error message.

RETURN VALUE

ltof returns (double) {i}.

ltoi returns (int) {i}.

ltou returns (unsigned) {i}.

ERRORS

```
{i} won’t fit in a "double"
{i} won’t fit in an "int"
{i} won’t fit in an "unsigned"
```

APPLICATION USAGE

These functions are useful for catching possible integer overflow in environments where this would not otherwise trigger an exception. In particular, the use of these functions avoids voluminous spurious "lint" warnings when copying long values to other data types.

SEE ALSO

IPW: dtoa

NOTES

"exactly represented" is tested as follows:

```
long test;
TYPE rtn;

rtn = i;
test = rtn;
```

For exact representation, test == i must be true. Note that in the case of ltof, the test will fail if {i} has more significant bits than a double’s mantissa, even if {i} is nominally within the range of double values.

MNXFP **MNXFP**

NAME
        mnxfp -- minima and maxima of fpixel vectors

SYNOPSIS
```
#include "fpio.h"

void mnxfp(vec, npixv, nbands, mmval)
fpixel_t *vec;
int npixv;
int nbands;
fpixel_t *mmval;
```

DESCRIPTION
        mnxfp find the minimum and maximum value for each of the {nbands} bands in the array of {npixv} pixel vectors pointed to by {vec}. The minima and maxima are interleaved into the array pointed to by {mmval}, such that the minimum value for band {i} is stored in {mmval}[2 * i], and the maximum value for band {i} is stored in {mmval}[2 * i + 1]. The array pointed to by {mmval} must therefore contain at least 2 * {nbands} elements.

SEE ALSO
        IPW: fpvread

NAME

ndig -- number of digits in an integer

SYNOPSIS

```
int ndig(i)
int i;
```

DESCRIPTION

ndig calculates the number of decimal digits needed to represent {i}.

RETURN VALUE

number of decimal digits in {i}

APPLICATION USAGE

ndig is typically used to determine the size of a string needed to hold an ASCII representation of an integer value:

```
int   i;
char *s;
...
s = (char *) ecalloc(ndig(i) + 1, 1);
if (s == NULL) {
        ...
}
s = itoa(i);
```

SEE ALSO

IPW: dtoa

NOTES

For negative values of {i}, the minus sign is counted as 1 digit.

NO_HISTORY **NO_HISTORY**

NAME

    no_history -- turn off history mechanism in bihwrite

SYNOPSIS

```
void no_history(fd)
int fd;
```

DESCRIPTION

    no_history prevents the inclusion of a history record for the current program in any basic image header subsequently written to file descriptor {fd}.

GLOBALS ACCESSED

```
_no_hist[fd]    flag associated with file descriptor {fd}
```

APPLICATION USAGE

    no_history may be called by programs that would otherwise produce redundant history records (e.g., mux).

SEE ALSO

    IPW: xxhwrite

NOTES

    no_history must be called BEFORE bihwrite, or else it will have no effect.

NAME

no_tty -- exit if file descriptor is a terminal

SYNOPSIS

```
void no_tty(fd)
int fd;
```

DESCRIPTION

If the file descriptor {fd} is connected to a terminal device, then no_tty terminates program execution, and writes one of the error messages described below. Otherwise, no_tty simply returns.

ERRORS

(All of the following assume {fd} is connected to a terminal device.)

{program’s usage message}

> {fd} is the standard input.

can’t write image data to a terminal

> {fd} is the standard output.

can’t do image I/O on a terminal device

> {fd} is neither the standard input nor the standard output.

APPLICATION USAGE

IPW users often forget to redirect the standard input and/or output of IPW programs, leaving these I/O channels attached to their terminals. IPW application programs should check all image file descriptors with no_tty, to ensure that image data are not accidentally written to a user’s terminal screen; or, in the case of the standard input, to remind the user that the program expects image, not keyboard, input:

```
int   fd;
...
fd = ustdin();
no_tty();
```

SEE ALSO

IPW: ipwenter, uropen

UNIX: ttyname

NAME

opt_check -- check for conflicting command-line options

SYNOPSIS

```
#include "getargs.h"

void opt_check(n_min, n_max, n_opts, opt[, ...])
int n_min, n_max, n_opts;
OPTION_T *opt;
```

DESCRIPTION

opt_check checks that at least {n_min} and at most {n_max} of the following {n_opts} option descriptors point to options that were specified on the command line. If the check fails, an error message is printed and execution terminates.

ERRORS

```
must specify at least {n_min} of: {option, ...}
may specify no more than {n_max} of: {option, ...}
```

These messages are printed on the standard error output if less than {n_min} or more than {n_max} of the options were specified.

APPLICATION USAGE

opt_check is typically called immediately after ipwenter to ensure that conflicting options were not specified. For example:

```
        opt_check(1, 2, 2, &opt_c, &opt_i);
```

will check that at least 1 of the options described by {opt_c} and {opt_i} was specified.

SEE ALSO

IPW: ipwenter

NOTES

opt_check causes program termination if an error occurs.

NAME

orhmake -- make an orientation (OR) header

SYNOPSIS

```
#include "orh.h"

ORH_T *orhmake(orient, origin)
char *orient, *origin;
```

DESCRIPTION

orhmake allocates a single-band orientation (OR) header. The header is initialized from orhmake's arguments:

{orient}	image orientation. May be either ROW, for row-major, or COLUMN, for column-major.
{origin}	corner of image origin. May be one of:

```
ORIG_1  (upper left)
ORIG_2  (upper right)
ORIG_3  (lower right)
ORIG_4  (lower left)
```

RETURN VALUE

success: pointer to new OR header

failure: NULL

SEE ALSO

IPW: xxhmake

NOTES

The symbols ROW, COLUMN, and ORIG_? are defined in orh.h.

The standard orientation for an IPW image is (ROW, ORIG_1). An image with this orientation does not need an OR header.

POW2 **POW2**

NAME
        pow2 -- compute integral powers of 2

SYNOPSIS
```
        int pow2(i)
        int i;
```

DESCRIPTION
        pow2 computes 2 to the power {i}, checking for out-of-range results.

RETURN VALUE
        success: 2**{i}

        failure: 0

ERRORS
        2**{i} is not an integer

                {i} is negative.

        2**{i} won’t fit in an "int"

                {i} is too large.

APPLICATION USAGE
        pow2 is typically called to determine the range of values which may be assumed by an {i}-bit integer:

```
int     npixvals;
BIH_T **bihp;
...
npixvals = pow2(ltoi(bih_nbits(bihpp[0])));
```

```
NAME
        pvread, pvwrite -- pixel-oriented I/O

SYNOPSIS
        #include "pixio.h"

        int pvread(fd, buf, npixv)
        int fd, npixv;
        pixel_t *buf;

        int pvwrite(fd, buf, npixv)
        int fd, npixv;
        pixel_t *buf;

DESCRIPTION
        These routines are analogous to uread and uwrite, but
        operate on "pixel vectors" instead of bytes.  A pixel vector
        consists of all of the pixel values (one for each band)
        associated with a single image sample, converted to the
        unsigned integer type pixel_t.

        pvread reads {npixv} pixel vectors from the image accessed
        by file descriptor {fd} to the buffer pointed to by {buf}.

        pvwrite writes {npixv} pixel vectors from the buffer pointed
        to by {buf} to the image accessed by file descriptor {fd}.

RETURN VALUE
        success:  number of pixel vectors read or written

        failure:  ERROR

ERRORS
        partial pixel vector read

                The {npixv} passed to pvread requested more pixel
                vectors than were available.

GLOBALS ACCESSED
        _piocb[fd]      pixio control block for file descriptor {fd}

APPLICATION USAGE
        These functions provide a simple way for application
        programs to access "raw" image pixel values, without having
        to worry about the underlying pixel representations (varying
        numbers of bytes and significant bits per pixel.)
```

SEE ALSO

IPW: fpvread, uread, uropen

NOTES

It is important to remember that the {npixv} argument to these functions is the number of pixel VECTORS to be read or written, and that the corresponding number of INDIVIDUAL PIXELS is {npixv} * (number of bands per image sample).

Calls to fpio, pixio, and uio I/O functions should not be intermixed on the same file descriptor.

NAME

sathmake -- make a satellite (SAT) header

SYNOPSIS

```
#include "sath.h"

SATH_T *sathmake(platform, sensor, location, gmdate, gmtime)
char *platform, *sensor, *location, *gmdate, *gmtime;
```

DESCRIPTION

sathmake allocates a single-band satellite (SAT) header. The header is initialized from sathmake’s arguments:

{platform}	platform (spacecraft) from which the image acquired (e.g., "Landsat 5", "SPOT 1", etc.)
{sensor}	sensor that acquired the image (e.g., "TM", "HRV", etc.)
{location}	location identifier for this image (e.g., TM path/row, SPOT K,J designator, etc.)
{gmdate}	Greenwich date of image acquisition, expressed as "YYYYMMDD".
{gmtime}	Greenwich Mean (Universal) time of image acquisition, expressed as "HHMMSS.S..." (i.e., arbitrary number of fractional seconds).

RETURN VALUE

success: pointer to new SAT header

failure: NULL

SEE ALSO

IPW: mksath, xxhmake

NOTES

There are not yet any standard values for {platform}, {sensor}, or {location}.

NAME

skewhmake -- make a skew (SKEW) header

SYNOPSIS

```
#include "skewh.h"

SKEWH_T *skewhmake(angle)
double angle;
```

DESCRIPTION

skewhmake allocates a single-band skew (SKEW) header. The header is initialized from skewhmake’s arguments:

{angle}	skew angle, indicating that successive image lines are offset such that the left edge of the image forms an angle of {angle} degrees from the vertical, measured clockwise.

RETURN VALUE

success: pointer to new SKEW header

failure: NULL

SEE ALSO

IPW: skew, xxhmake

NOTES

The skew angle is measure in DEGREES, not radians.

NAME
        skiphdrs, copyhdrs, fphdrs -- process groups of image headers

SYNOPSIS
```
        #include "gethdrs.h"

        void skiphdrs(fdi)
        int fd;

        void copyhdrs(fdi, nbands, fdo)
        int fdi, nbands, fdo;

        #include "fpio.h"

        void fphdrs(fdi, nbands, fdo)
        int fdi, nbands, fdo;
```

DESCRIPTION
        These functions read all remaining image headers from the input image accessed by file descriptor {fdi}.

        skiphdrs skips (i.e. does nothing) with the input headers.

        copyhdrs copies any input headers associated with band numbers less than {nbands} to the output image accessed by file descriptor {fdo}. Headers associated with band numbers greater than or equal to {nbands} are skipped.

        fphdrs behaves identically to copyhdrs, except that any input LQ headers are made available to the fpio subsystem.

APPLICATION USAGE
        These functions are used by programs which do not require direct access to any of the input image's optional headers. The image data streams involved are left positioned at the first pixel in the image.

        Use gethdrs to ingest specific optional input image headers. Use the appropriate xxhwrite routines to write specific optional output image headers.

SEE ALSO
        IPW: fpvread, gethdrs, xxhread, xxhwrite

NOTES

These functions terminate program execution if any errors are encountered.

bihread must have been called on {fdi} before calling any of these functions. bihwrite must have been called on {fdo} before calling copyhdrs or fphdrs.

NAME

```
        sunhmake -- make a sun position (SUN) header
```

SYNOPSIS

```
        #include "sunh.h"

        SUNH_T *sunhmake(cos_zen, azm)
        double cos_zen;
        double azm;
```

DESCRIPTION

```
        sunhmake allocates a single-band sun position (SUN) header.
        The header is initialized from sunhmake’s arguments:

        {cos_zen}       cosine of the solar zenith angle.

        {azm}           solar azimuth, measured in radians
                        counterclockwise from the south.
```

RETURN VALUE

```
        success:  pointer to new SUN header

        failure:  NULL
```

ERRORS

```
        sunhmake: bad cosine {cos_zen}

                {cos_zen} is outside the range [-1..1]

        sunhmake: bad azimuth {azm}

                {azimuth} is outside the range [-PI..PI].
```

SEE ALSO

```
        IPW: mksunh, xxhmake
```

NOTES

```
        A negative {cos_zen} indicates that the sun is below the
        horizon.
```

UCLOSE **UCLOSE**

NAME

    uclose -- close UNIX file descriptor

SYNOPSIS

```
int uclose(fd)
int fd;
```

DESCRIPTION

    uclose forces any buffered uio output pending on file descriptor {fd} to be written, and then closes the file. All uio internal state information pertaining to {fd} is reset (cleared or deallocated).

RETURN VALUE

    success: OK

    failure: ERROR

GLOBALS ACCESSED

_uiocb[fd]	uio control block for file descriptor {fd}

APPLICATION USAGE

    Application programs normally do not need to explicitly call uclose, since it is called automatically for all open files by ipwexit. However, it may be necessary to explicitly close a file that has been opened for writing, if you subsequently want to reopen it for reading, since uio does not allow simultaneous read/write access.

SEE ALSO

    IPW: ipwexit, ugets, uread, uropen

    UNIX: close

NOTES

    Any buffered INPUT pending on {fd} is discarded.

    Passing a uclose’d file descriptor to any uio routine will cause an assertion violation.

NAME

ucopy -- copy data between files

SYNOPSIS

```
long ucopy(fdi, fdo, ncopy)
int fdi, fdo;
long ncopy;
```

DESCRIPTION

ucopy copies {ncopy} bytes from file descriptor {fdi} to file descriptor {fdo}.

RETURN VALUE

success: number of bytes copied

failure: ERROR

APPLICATION USAGE

ucopy is useful chiefly to programs that are copying all of the pixel data from an input IPW image to an output non-IPW image (see the documentation for the imgsize function.)

For copying pixel data between IPW image files, use the imgcopy function.

SEE ALSO

IPW: imgcopy, imgsize, uropen

UEOF **UEOF**

NAME

```
        ueof -- check for uio end-of-file
```

SYNOPSIS

```
        bool_t ueof(fd)
        int fd;
```

DESCRIPTION

```
        ueof tests whether there any data remaining to be read from
        file descriptor {fd}, which must have been opened for
        reading by uropen.
```

RETURN VALUE

```
        TRUE if end-of-file has been reached; else FALSE
```

GLOBALS ACCESSED

```
        _uiocb[fd]      uio control block for file descriptor fd
```

SEE ALSO

```
        IPW: uread
```

UGETS **UGETS**

NAME
        ugets, uputs -- text-oriented I/O

SYNOPSIS
```
        char *ugets(fd, buf, nbytes)
        int fd, nbytes;
        char *buf;

        int uputs(fd, buf)
        int fd;
        char *buf;
```

DESCRIPTION
        ugets reads bytes from file descriptor {fd} into the buffer pointed to by {buf}, until any of the following are true:

- {nbytes}-1 bytes have been read and transferred to {buf}; or
- a newline character ('0) has been read and transferred to {buf}; or
- an end-of-file condition is encountered on {fd}.

An EOS character (' ') is placed after the last byte transferred to {buf} (i.e., on return, {buf} will point to a null-terminated C string).

uputs writes the EOS-terminated string pointed to by {buf} to file descriptor {fd}. The terminating EOS is NOT written.

RETURN VALUE
        success: {buf} (ugets); number of bytes written (uputs)

        failure: NULL (ugets); ERROR (uputs)

GLOBALS ACCESSED

_uiocb[fd]	uio control block for file descriptor fd

APPLICATION USAGE
        Use ugets and uputs for text-oriented input and output on uio files.

SEE ALSO

IPW: uropen, uread

UNIX: fgets, fputs

NOTES

ugets and uputs are analogous to the UNIX stdio routines fgets and fputs, respectively.

Calls to ugets and uread, or uputs and uwrite, may be freely intermixed.

NAME

```
        uread, uwrite -- byte-oriented I/O
```

SYNOPSIS

```
        int uread(fd, buf, nbytes)
        int fd, nbytes;
        addr_t buf;

        int uwrite(fd, buf, nbytes)
        int fd, nbytes;
        addr_t buf;
```

DESCRIPTION

uread is a reliable interface to the UNIX read function. It reads {nbytes} bytes from file descriptor {fd} into the buffer pointed to by {buf}, unless end-of-file is reached, or an I/O error occurs.

uwrite is a reliable interface to the UNIX write function. It writes {nbytes} bytes from the buffer pointed to by {buf} to file descriptor {fd}. The various types of write errors are detected and folded into a single error return.

These functions provide transparent internal buffering, which conceals the effects of I/O through UNIX pipes.

RETURN VALUE

success: number of bytes read/written

failure: ERROR

ERRORS

incomplete write

Some number of bytes less than the number requested were written. This is not necessarily due to a hard I/O error.

GLOBALS ACCESSED

_uiocb[fd]	uio control block for file descriptor {fd}

APPLICATION USAGE

uread and uwrite are intended to replace the direct use of the UNIX read and write system calls in IPW application code.

SEE ALSO

IPW: uropen

UNIX: read, write

NOTES

The {fd} passed to uread must have originally been obtained from either uropen or ustdin.

The {fd} passed to uwrite must have originally been obtained from either uwopen or ustdout.

Calls to fpio, pixio, and uio I/O functions should not be intermixed on the same file descriptor.

NAME

uremove -- delete a file

SYNOPSIS

```
int uremove(filename)
char *filename;
```

DESCRIPTION

remove causes the file {filename} to be deleted.

RETURN VALUE

success: OK

failure: ERROR

APPLICATION USAGE

uremove is typically used by application programs to delete scratch files. It should be always be called instead of the UNIX system call unlink, or the ANSI library function remove.

SEE ALSO

IPW: uclose, uropen

UNIX: remove, unlink

NOTES

If the file {filename} is currently open (i.e., is connected to a file descriptor), then the behavior of remove is implementation-defined.

NAME

uropen, ustdin, uwopen, ustdout -- access files

SYNOPSIS

```
int uropen(name)
char *name;

int ustdin()

int uwopen(name)
char *name;

int ustdout()
```

DESCRIPTION

These functions provide low-level (uio) access to named UNIX files, or to the standard input and output. They are the ONLY means by which IPW application programs may obtain file descriptors.

uropen returns a readable file descriptor connected to the UNIX file {name}. The name "-" means the standard input.

ustdin (equivalent to uropen("-")) returns a file descriptor for the standard input.

uwopen returns a writable file descriptor connected to the UNIX file {name}. The file is created if it does not already exist, or truncated if it does.

ustdout returns a file descriptor for the standard output.

RETURN VALUE

success: uio file descriptor

failure: ERROR

ERRORS

can’t initialize standard input

> ustdin was unable to allocate a uio buffer for the standard input.

can’t initialize standard output

> ustdout was unable to allocate a uio buffer for the standard output.

APPLICATION USAGE

Use uropen wherever you would normally use the UNIX system call open() to open a file for reading.

ustdin should be called by an IPW main() if the program will be reading image data from the standard input. ustdin should NOT be called if the program uses UNIX stdio input functions (e.g., scanf) on the standard input.

Use uwopen wherever you would normally use the UNIX system call creat() to open a file for writing.

ustdout should be called by an IPW main() if the program will be writing image data to the standard output. ustdout should NOT be called if the program uses UNIX stdio output functions (e.g., printf) on the standard output.

SEE ALSO

IPW: ugets, uclose, ucopy, ueof, uread, urskip

UNIX: creat, open

NOTES

ustdin and ustdout cause program termination if any errors are encountered.

A default protection mode (usually -rw-r--r--) is applied to files created by uwopen.

NAME

urskip -- skip input on UNIX file descriptor

SYNOPSIS

```
long urskip(fd, nbytes)
int fd;
long nbytes;
```

DESCRIPTION

urskip reads and discards {nbytes} bytes from file descriptor {fd}.

RETURN VALUE

```
success:  number of bytes skipped

failure:  ERROR
```

GLOBALS ACCESSED

```
_uiocb[fd]      uio control block for file descriptor {fd}
```

APPLICATION USAGE

Use urskip to effect a "forward seek" on {fd}. Note that you cannot call lseek directly from an IPW program since

- it would confuse uio’s internal buffering;

- IPW input "files" are often really pipes, which do not support seeking (urskip will call lseek if {fd} supports it.)

SEE ALSO

IPW: uread

UNIX: lseek

NOTES

urskip will cause an assertion violation if {nbytes} is less than 1 (i.e., only forward skips are allowed.)

NAME

usrerr, uferr, syserr -- deferred error handling

SYNOPSIS

```
void usrerr(format, ...)
char *format;

void uferr(fd)
int fd;

void syserr()
```

DESCRIPTION

These functions save error information in global variables, for later incorporation in error messages generated by the error or warn functions.

usrerr saves the error message described by the printf-style {format} and optional additional arguments.

uferr saves the file descriptor {fd}, which may be used by error or warn to obtain the associated file name.

syserr causes any current UNIX system error condition to be saved. The associated error message will be incorporated in the output generated by error or warn.

GLOBALS ACCESSED

_usrerr	usrerr stores its error message in this string.
_fderr	uferr sets this to {fd}.
_errno	syserr sets this to the current UNIX "errno" value.

APPLICATION USAGE

usrerr is called to save topical information that would otherwise be passed directly to error or warn; for example, if error or warn may not be called until the current function returns.

uferr should be called before error or warn if the name of the file accessed by {fd} should appear in the error message. Note that any IPW library functions that deal with file descriptors will call uferr themselves whenever appropriate.

syserr is seldom invoked by application programs, since they do not directly invoke UNIX system calls.

SEE ALSO

IPW: error, uropen

UNIX: errno, perror, vsprintf

WINHMAKE **WINHMAKE**

NAME

```
        winhmake -- make a window (WIN) header
```

SYNOPSIS

```
        #include "winh.h"

        WINH_T *winhmake(bline, bsamp, dline, dsamp)
        double bline, bsamp, dline, dsamp;
```

DESCRIPTION

```
        winhmake allocates a single-band window (WIN) header.  The
        header is initialized from winhmake’s arguments:

        {bline}         window coordinate of image line 0

        {bsamp}         window coordinate of image sample 0

        {dline}         window distance between image lines

        {dsamp}         window distance between image samples
```

RETURN VALUE

```
        success:  pointer to new WIN header

        failure:  NULL
```

SEE ALSO

```
        IPW: mkwinh, window, xxhmake
```

NOTES

XXHDUP **XXHDUP**

NAME

xxhdup -- duplicate an XX header

SYNOPSIS

```
#include "xxh.h"

XXH_T **xxhdup(old, nbands)
xxh_t **old;
int nbands;
```

DESCRIPTION

xxhdup is the general form of a specific function that is provided for each type of IPW image header.

xxhdup allocates a new XX header that duplicates the first {nbands} bands of the XX header pointed to by {old}.

RETURN VALUE

success: a pointer to the newly-allocated duplicate header

failure: NULL

APPLICATION USAGE

To duplicate a GEO header:

```
        GEOH_T **i_geohpp;
        GEOH_T **o_geohpp;
        ...
        o_geohpp = geohdup(i_geohpp, nbands);
        if (o_geohpp == NULL) {
                error("can’t duplicate GEO header");
        }
        ...
```

SEE ALSO

IPW: gethdrs, xxhmake, xxhread, xxhwrite

NOTES

The header to be duplicated must have at least {nbands} bands.

The function bihdup does not take an {nbands} argument, instead obtaining the number of bands directly from the {old} header. bihdup always duplicates all of the bands of the {old} header.

XXHMAKE XXHMAKE

NAME

```
        xxhmake -- procedure for creating a new XX header
```

SYNOPSIS

```
        #include "xxh.h"

        XXH_T *xxhmake(...)
```

DESCRIPTION

```
        xxhmake is the general form of a specific function that is
        provided for each type of IPW image header.

        xxhmake allocates a SINGLE BAND of an XX header and
        initializes it with the header-specific argument values.
```

RETURN VALUE

```
        success:  a pointer to the newly-created single-band
                  component of an XX header

        failure:  NULL
```

APPLICATION USAGE

(The following example assumes that the header being created will eventually be written to the {nbands}-band image accessed by file descriptor {fd}.)

To allocate a new image header, you must first allocate the array of pointers to each band’s header:

```
        int             band;   /* image band #          */
        int             fd;     /* image file descriptor */
        int             nbands; /* # bands / sample      */
        XXH_T         **xxhp;   /* -> XX headers         */

        ...

        xxhp = (XXH_T **) hdralloc(nbands, sizeof(XXH_T *),
                                   fd, XXH_HNAME);
```

then, you must create each band’s header with xxhmake:

```
        for (band = 0; band < nbands; ++band) {
                xxhp[band] = xxhmake(...);
                if (xxhp[band] == NULL) {
                        uferr(fd);
                        error("band %d: %s header",
                              band, XXH_HNAME);
                }
        }
```

SEE ALSO

IPW: hdralloc, xxhdup, xxhread, xxhwrite

plus the documentation for each header-specific hmake function.

NAME

xxhread -- read an XX image header

SYNOPSIS

```
#include "xxh.h"

XXH_T **xxhread(fd)
int fd;
```

DESCRIPTION

xxhread is the general form of a specific function that is provided for each type of IPW image header.

xxhread reads an XX image header from file descriptor {fd}. An array of per-band XXH_T pointers is allocated. If a band has an XX header, then an XXH_T header is allocated and its address is placed in the corresponding array element; otherwise, the corresponding array element is NULL.

RETURN VALUE

success: pointer to array of XXH_T pointers

failure: NULL

ERRORS

```
        can’t allocate "xx" header
        can’t allocate array of "xx" header pointers

                The header won’t fit in memory.

        "xx" header: bad band "{band}"

                An XX per-band header contains an invalid band
                number.

        The following errors pertain ONLY to bihread:

        invalid IPW image (no "basic_image_i" header)

                An image does not begin with a BI header.

        "basic_image_i" header: nbands < 1

                The BI header contains an invalid number of bands.

        no "basic_image" header for band {band}

                The image has no per-band BI component for the
                specified band.  The BI header is the only IPW image
                header that must be present for all image bands.
```

GLOBALS ACCESSED

Pointers to certain image headers are saved in global data structures, so that the information in the header is accessible via the file descriptor of the image from which the header was read:

```
        _bih[fd]        (set by bihread)

        _lqh[fd]        (set by lqhread)
```

APPLICATION USAGE

Except for bihread, which must always be called explicitly, these functions are usually not called directly by IPW application programs. Instead, pointers to the appropriate xxhread functions for the headers a program wishes to ingest are passed to the gethdrs function.

SEE ALSO

IPW: gethdrs, xxhwrite

NOTES

"failure" includes encountering a premature end-of-file.

It is an error to call xxhread if the next item to be read from {fd} is not an XX header.

NAME

xxhwrite -- write an XX image header

SYNOPSIS

```
#include "xxh.h"

int xxhwrite(fd, xxhpp)
int fd;
XXH_T **xxhpp;
```

DESCRIPTION

xxhwrite is the general form of a specific function that is provided for each type of IPW image header.

xxhwrite writes the XX headers pointed to by {xxhpp} to file descriptor {fd}.

RETURN VALUE

success: OK

failure: ERROR

GLOBALS ACCESSED

Pointers to certain image headers are saved in global data structures, so that the information in the header is accessible via the file descriptor of the image to which the header was written:

```
_bih[fd]        (set by bihwrite)

_lqh[fd]        (set by lqhwrite)
```

APPLICATION USAGE

Output image headers may be simply copied from an input image, using the gethdrs or copyhdrs functions, or they may be written explicitly, using the appropriate xxhwrite function. The latter method is used by programs which create new headers (e.g., transpose, which creates an OR header), or modify existing ones (e.g., window, which may modify input WIN and GEO headers.)

SEE ALSO

IPW: boimage, gethdrs, xxhread

NOTES

bihwrite must always be called explicitly to write a BI header to an output image, before any other headers or pixel data are written to that image.

7.3. SHELL SCRIPT SUPPORT ROUTINES

This section contains copies of the on-line documentation for the IPW commands that exist solely to support IPW shell scripts. The format of this documentation is described in §5.2 and §A.1.1.1.

NAME

isposint -- test for positive nonzero integer argument

SYNOPSIS

isposint argument

DESCRIPTION

isposint tests whether {argument} is a positive nonzero decimal integer.

DIAGNOSTICS

The exit status is 0 is {argument} is a positive nonzero integer; nonzero if it is not.

EXAMPLES

A command-line processing loop in a shell script might contain the following:

```
        while :; do
                case $1 in
                ...
                -i)     nbits=$2
                        isposint $nbits ||
                            exec sherror $0 \
                            "$nbits: not a positive integer"
                        shift
                        ;;
                ...
                esac
                shift
        done
```

SEE ALSO

IPW: sherror

NOTES

isposint is currently implemented as a shell script.

NAME

sherror -- standard IPW error message for shell scripts

SYNOPSIS

```
sherror pgm message [file]
```

DESCRIPTION

sherror is an error message generator that may be called by IPW shell scripts. The error messages are printed in a similar format to the IPW error message generated by the library function error.

The {pgm} argument is almost always $0 (i.e., the name by which the shell script was invoked). The {message} argument must be a single string (i.e., it must be quoted if it contains white space). The optional {filename} argument, if present, will appear on a separate line in the error message and will be identified as a file name.

DIAGNOSTICS

ERROR: wrong # args!

EXAMPLES

sherror is usually exec'd, so that it will terminate execution of the shell script. For example:

```
...
case $1 in
...
-*)     exec sherror $0 "$1: unsupported option"
        ;;
...
```

or:

```
[ -r $filename ] ||
    exec sherror $0 "can't access file" $filename
```

SEE ALSO

IPW: error, usage

NOTES

sherror always exits with a nonzero status

sherror is currently implemented as a shell script.

NAME
usage -- standard IPW usage message for shell scripts

SYNOPSIS
usage pgm synopsis description

DESCRIPTION
usage is a usage message generator that may be called by IPW shell scripts. The usage messages are printed in a similar format to the usage message generated by the library function ipwenter.

The {pgm} argument is almost always $0 (i.e., the name by which the shell script was invoked). The {synopsis} and {description} arguments must be single strings (i.e., must be quoted if they contain white space).

DIAGNOSTICS
USAGE: wrong # args!

EXAMPLES
The following is a typical shell script command-line parsing sequence, taken from the mklut command:

```
...
optstring='i:o:k:'
synopsis='[-i in_nbits] [-o out_nbits] [-k bkgd]'
description='make look-up table'

# get command-line arguments

set - `getopt "$optstring" $* 2>/dev/null` ||
        exec usage $0 "$synopsis" "$description"
...
```

SEE ALSO
IPW: ipwenter, sherror

NOTES
usage always exits with a nonzero status

usage is currently implemented as a shell script.

CHAPTER 8: INSTALLATION AND MAINTENANCE

This chapter describes how to install the IPW software. IPW is currently known to operate on the following systems:

hardware	**operating system**
Sun-3	SunOS 3.4, 3.5, 4.0.3
Sun-4	SunOS 4.0.3
IBM PS/2 Model 80	AIX 1.1
IBM RT Model 135	AOS 4.3

Installing IPW on any of these systems should be straightforward. This chapter also provides guidance for porting IPW to a new environment.

8.1. CREATE IPW ACCOUNT

Your IPW host system should have a unique `ipw` user and a unique `ipw` group. Create them now if they don't already exist (you must be the super-user to do this). Here is the relevant entry from our `/etc/passwd` file:

```
ipw:*:22:21:(IPW subsystem):/usr/home/ipw:/bin/csh
```

The `*` in the password field will keep the account disabled until you explicitly set a password. The user- and group-IDs, and the home directory, will have to be customized for your system.

Here is the relevant line from our `/etc/group` file:

```
ipw::21:dozier,frew
```

You should edit the user list to include only those users (besides `ipw`) to whom you wish to grant write access to IPW source files, libraries, etc.

Neither the `ipw` user nor the `ipw` group are required for IPW to function, but they simplify administration. Users can access the IPW root directory as `~ipw` (from shells which support tilde expansion), and the `ipw` password can be given to an IPW administrator without compromising the security of non-IPW files. The `ipw` user can further control access by members of the `ipw` group by selectively enabling or disabling group write permissions on IPW directories. **As distributed, IPW has group write permission enabled on all directories,** so it is a good idea to have the `ipw` user in a group of its own, even if no other users are to be admitted to the group.

Next, make a home directory for `ipw`:

```
# mkdir ~ipw
# chown ipw ~ipw
# chgrp ipw ~ipw
# chmod 775 ~ipw
```

Finally, set a password for the `ipw` account:

```
# passwd ipw
```

All the remaining installation instructions assume you are logged in as (or `su`'d to) `ipw`.

8.2. LOAD THE DISTRIBUTION TAPE

IPW is distributed on magnetic tape[36], as a single UNIX `tar` archive. To extract the tape, type:

```
    % cd ~ipw
    % tar xvpbf blocking-factor tape-device
```

where *blocking-factor* is 20 for 1/2-inch tapes, or 200 for 1/4-inch cartridge tapes. *tape-device* is the appropriate UNIX tape device name (e.g., `/dev/rmt8`, `/dev/rst0`, etc.) The `p` flag is important; it guarantees that the original file permissions will be restored.

8.2.1. Possible ownership problems

Do an `ls -lg` (or `ls -l` for USG-derived UNIXes) on `~ipw`. If the extracted files and directories are owned by user `ipw` and group `ipw`, then skip the remainder of this section.

If your UNIX system allows non-privileged users to execute the chown system call, then the `tar` command shown above may set the user- and group-IDs of the IPW files to whatever they were on the system where the distribution tape was created. These will almost certainly NOT be the correct user- and group-IDs for your system. If this occurs, you will have to become root and run `chown` and `chgrp` on the IPW hierarchy.

If your system has the `xargs` command, you can type[37]:

```
    # find ~ipw -print | xargs chown ipw
    # find ~ipw -print | xargs chgrp ipw
```

Otherwise, type

```
    # find ~ipw -exec chown ipw {} \; -exec chgrp ipw {} \;
```

8.3. DIRECTORY HIERARCHY

The files comprising the IPW distribution are organized into a standard hierarchy of directories. This yields several benefits:

- Once the structure of the hierarchy is learned, an IPW programmer or administrator can quickly locate any particular file.
- IPW programs can locate data files as long as the root of the hierarchy is known.
- Additions to IPW have logical homes in the directory hierarchy.

The IPW directory hierarchy is based on the traditional `/usr` hierarchy found on most UNIX systems, and is thus easily learned by most UNIX users. In particular, the following top-level directories are always present:

```
    bin  doc  etc  h  lib  pub  skel  src
```

Other top-level directories are present in some IPW implementations, but are not part

[36] If you have obtained IPW via `ftp`, then substitute the name of the archive file for *tape-device* in the following instructions.

[37] On BSD-derived systems, the separate `chown` and `chgrp` commands may be combined into a single `chown` *user.group* command.

of the standard distribution. Figure 8.1 illustrates the standard IPW directory hierarchy.

```
      |-bin
      |
      |-doc
      |
      |-etc
      |      |-ansi--|-conf.float
      |      |       |-conf.limits
      |-h----|-conf
      |      |
      |      |-posix-|-conf
      |
      |-lib--|-make--|-conf
      |
      |-pub
      |      |-hdr
      |      |-lib
      |-skel-|-misc
      |      |-pgm
      |      |-sh
      |              |-bitcom
      |              |-cmpimg
      |      |-bin---| ...
$IPW--|      |       |-window
      |      |       |-zoom
      |      |
      |      |       |-atob
      |      |       |-btoa
      |      |-etc---| ...
      |      |       |-rastool
      |-src--|       |-xim
             |
             |       |-isposint
             |       |             |-args
             |       |             |-bih
             |       |-libipw------| ...
             |       |             |-sath
             |       |             |-winh
             |-lib---|
                     |-libunix-----|-conf
                     |             |-src
                     |-sherror
                     |
                     |-syslint
                     |
                     |-usage
```

Figure 8.1: standard IPW directory hierarchy

`bin` contains the executable files (binaries and shell scripts) comprising most IPW commands.

`doc` contains any machine-readable documentation distributed with IPW. If a documentation file contains embedded formatting commands (e.g., for `troff`), then a formatted version will also be present.

`etc` contains the executable files comprising any IPW commands not kept in `bin`. There are two kinds of commands in `etc`:

- maintenance commands (e.g., `ipwmake`)
- non-IPW commands distributed along with IPW. These include various public-domain utilities often invoked by IPW shell scripts.

`etc` contains only `ipwmake` and `install` at the beginning of the IPW installation. to simplify building the rest of IPW.

`h` contains all IPW source header files (i.e., files that are `#included` by IPW C source files.) The file `h/ipw.h` is the "master" IPW header file, and must be `#included` by all IPW source files.

`lib` contains support files and directories; i.e., files that are not accessed directly by an IPW user, but rather indirectly by other components of IPW.

- Files named `lib`*library*`.a` are UNIX archives of compiled IPW functions, accessed by `ipwmake`.
- Files named `llib-`*library*`.ln` are lint libraries, accessed by `ipwlint`.[38]
- Files in the `lib/make` directory contain standard definitions and rules used by `ipwmake`.

`skel` contains template ("skeleton") files that may be used as a starting point for creating new IPW functions and commands.

`src` contains all source code (C and shell) for IPW. The directory hierarchy under `src` mirrors the top-level IPW hierarchy:

- `src/bin` contains the sources for all `bin` commands. The source for each command occupies a its own directory under `src/bin`; e.g., `src/bin/lutx` contains the source for the `lutx` command.
- `src/etc` contains the sources for all `etc` commands. As with `src/bin`, each command has its own directory.
- src/lib contains the sources for all libraries and commands in `lib`. For each library `lib`*library*`.a` there is a directory `src/lib/`*library*.

Files that can be created entirely from other IPW files are not normally included on the IPW distribution media. In particular, `bin` will be empty, and `etc` and `lib` will contain only enough files to "bootstrap" the installation of IPW.

There are some additional standard file and directory names that are used throughout the IPW directory hierarchy:

- `Makedefs` files describe how `ipwmake` is to process a particular component of IPW (see §8.5.)
- `README` files contain design notes, suggestions for further development, and other miscellaneous information.

---

[38] These files are redundant if an ANSI-conforming C compiler is available.

- RCS directories contain the master files used by the RCS revision control system (these files are not usually distributed with IPW).
- TEST directories contain test drivers and input data for the IPW components in the parent directory.
- conf directories contain system-specific versions of files that should be installed (either as-is or with suitable modifications) in the parent directory of the conf directory. These files are described in more detail below.

8.4. CONFIGURING THE SHELL(S)

From here on, we make 2 important assumptions about the IPW environment:

- An IPW user's interactive command interpreter is csh (the Berkeley "C shell")
- command files ("shell scripts") whose first character is a colon (:) will be executed by sh (the "Bourne shell"). Your sh must understand the sharp-sign (#) comment convention; no other extensions to the original Version 7 [BTL 1983] sh are used.

See **pub/cshrc** below for suggestions on dealing with departures from these assumptions.

Also, all path names given below are relative to ˜ipw.

8.4.1. pub/cshrc

The file pub/cshrc should be sourced by all IPW users before they issue any IPW commands. Usually, a line like the following is placed in an IPW user's .cshrc file:

```
source ˜ipw/pub/cshrc
```

This file should not need to be changed, but take a look at it now. The environment variable IPW must be set to the name of the IPW root directory. The environment variable TMPDIR must be set to the name of a directory on which scratch images may be written. Finally, the directories ˜ipw/bin and ˜ipw/etc must be in an IPW user's search path.

If shell scripts beginning with a colon aren't automatically run by sh, then you may need to add the following line to pub/cshrc:

```
setenv SHELL /bin/sh    # or wherever sh lives
```

(This is necessary on some XENIX systems, for instance.)

If your interactive command interpreter isn't csh, then you'll need to prepare an equivalent of pub/cshrc. For example, a pub/profile for sh might look like this:

```
IPW=/usr/home/ipw          # or whatever corresponds to ˜ipw
TMPDIR=/usr/tmp
export IPW TMPDIR
PATH=$PATH:$IPW/bin:$IPW/etc
```

and then sh users would put the following in their .profile file:

```
. /usr/home/ipw/pub/profile
```

8.4.2. `lib/ipwenv`

The file `lib/ipwenv` contains local definitions for nonportable UNIX commands used by IPW shell scripts. Check the definitions in this file and fix any that are inappropriate for your host:

`AWK` An `awk` command that conforms to the 1985 definition [Aho 1988] of the `awk` language. As of this writing, all IPW scripts will work with older versions of `awk`, but this may change with future releases of IPW.

`LC_WD`
Produce a multi-column listing of the current directory. On some UNIX systems, the `ls` command does this by default.

`LLG` Produce a ‘‘long’’ directory listing with both the owner and the group of each file. Some UNIX systems use the command `ls -lg`, while others use `ls -l`.

8.5. CONFIGURING MAKE

There are about as many varieties of the `make` command as there are varieties of UNIX. For the sake of portability, we assume a lowest common denominator of capability (i.e., Version 7 `make`), and provide an `ipwmake` command to automatically generate an appropriate `makefile`[39] for each IPW command and library. `ipwmake` uses some nonportable ‘‘boilerplate’’ that you will probably have to customize.

8.5.1. `lib/make/local`

The `lib/make` directory contains the boilerplate used by the `ipwmake` command (as opposed to program-specific instructions, which are kept in a `Makedefs` file in each source directory). You can take a look at `etc/ipwmake` to see how it all fits together. The only file that should need changing should be `lib/make/local`. The directory `lib/make/local/conf` contains versions of `lib/make/local` for various systems. You should copy or link the most appropriate one to `lib/make/local`, editing it if necessary.

A detailed description of the contents of `lib/make/local` is beyond the scope of this chapter; see the comments therein. However, here are some things to watch out for:

`AR_XOPTS`
`ipwmake`’s library-update procedure relies on the ability to extract files from an archive with their original modification times. BSD-derived `ar`’s use the `o` qualifier to do this, while (some) USG `ar`’s do this by default.

If your `ar` command **cannot** be persuaded to restore the original modification times of extracted files, then you will have to modify `lib/make/rules` so that object files are **not** removed from the library source directory after they are inserted into the library. This will consume additional disk space, but will prevent redundant compilations if you plan to modify or extend the IPW libraries.

8.6. CONFIGURING THE C COMPILER

Now the header files used by IPW C code must be modified. This basically involves elaborating the specific ways in which your system differs from an ‘‘ideal’’

[39] The `makefile` generated by `ipwmake` is piped directly to `make`.

combination of ANSI C [ANSI 1989] and POSIX [IEEE 1988].

8.6.1. `h/config.h`

The file `h/config.h` contains many `#define`'s that customize your C environment. The directory `h/conf` contains versions of `h/config.h` for various systems. You should copy or link the most appropriate one to `h/config.h`. Unless your system EXACTLY matches one of the configurations in `h/conf`, you should check each `#define` against its description in `h/conf/README`. Here are some things to pay special attention to:

`CC_NOBCOPY`, `CC_NOBZERO`
: Note that these options read as a double-negative; i.e. `CC_NOBCOPY` should be 1 if `bcopy()` ISN'T provided in your host's C library.

`CC_BIGENDIAN`
: This option refers to the order of bytes in an integer; it should be 1 on machines where the memory address of an integer is also the address of its high-order byte (e.g., MC680x0).

`OPT_BUF`
: This is the buffer size IPW uses with `read()` and `write()`. It **MUST NOT** be larger than the maximum number of bytes that may be written to a pipe in a single `write()`. 4096 is the largest portable value. On memory-limited systems, you may want to use a smaller value, in which case 1024 is suggested.

`FREG_?`
: These options are only appropriate for machines with separate floating-point registers (e.g. MC68881) that are known to the C compiler. Do **NOT** use for machines that store floating-point values in general registers (e.g. VAX).

8.6.2. `h/ansi`

If your system has ANSI-conforming C header files, then you must copy or link `<float.h>` and `<limits.h>` to the directory `h/ansi`. Otherwise, you must create substitutes for these header files, as described below.

8.6.2.1. `h/ansi/float.h`

The file `h/ansi/float.h` is a substitute for the ANSI header file `<float.h>`. The directory `h/ansi/conf.float` contains versions of `h/ansi/float.h` for various systems. If you're lucky, you'll be able to copy or link one of these files unchanged into `h/ansi/float.h`. If you're a bit less lucky, i.e. if your particular machine isn't mentioned explicitly, BUT it supports IEEE 754 floating point, you can copy or link the generic file `h/ansi/conf.float/ieee754`.

If, however, you're not so lucky, you'll have to generate a `h/ansi/float.h` from scratch. First, try going to the directory `src/etc/machine` and typing `make`. If the resulting program `machine`[40] compiles and runs successfully, it will generate a `float.h` case for your machine - you just have to catch the output of `machine -float` in `h/ansi/float.h`.

In the unlikely event that `machine` won't compile or run successfully, you'll have to do some detective work. Your system may have a file `<values.h>` that contains

[40] `machine` incorporates a C version of the Fortran subroutine `machar` [Cody 1988].

some of the parameters you'll need; otherwise, consult a hardware manual or a local guru.

8.6.2.2. h/ansi/limits.h

The file h/ansi/limits.h is a substitute for the ANSI header file <limits.h>. The directory h/ansi/conf.limits contains versions of h/ansi/limits.h for various systems. You should copy or link the most appropriate one to h/ansi/limits.h, editing it if necessary. If none of the existing versions are usable, you can generate one with the machine command described in the previous section, invoked as machine -limits.

8.6.3. h/posix

If your system conforms to the IEEE 1003 (“POSIX”) UNIX specification, then you must copy or link <limits.h> to the directory h/posix. Otherwise, you must create a substitute for this header file, as described below.

8.6.3.1. h/posix/limits.h

The file h/posix/limits.h is a substitute for the POSIX header file <limits.h>. Actually, this file only contains POSIX material NOT present in the ANSI <limits.h> (the real POSIX <limits.h> is a superset of the ANSI <limits.h>). The directory h/posix/conf contains versions of h/posix/limits.h for various systems. You should copy or link the most appropriate one to h/posix/limits.h, editing it if necessary (see h/posix/conf/README.)

8.7. SRC/LIB/LIBUNIX

IPW provides substitutes for some ANSI library functions that may not be available on all systems. The file src/lib/libunix/Makedefs controls the building of this compatibility library. The directory src/lib/libunix/conf contains versions of src/lib/libunix/Makedefs for various systems. You should copy or link the most appropriate one to src/lib/libunix/Makedefs, editing it if necessary. Then, you must copy or link the appropriate source files from src/lib/libunix/src into src/lib/libunix.

8.8. BUILDING IPW

At this point, you should be able to run ipwmake in the src directory and build the entire system:

```
% cd ˜ipw/src
% ipwmake install >& make.out &
```

Any errors in make.out (other than those explicitly marked ignored) indicate a problem, and should be tracked down before using IPW.

8.9. POST-INSTALLATION ISSUES

Certain IPW commands cannot (yet) be installed automatically, since they require non-portable system header files and/or function libraries. At this writing, the following commands must be installed by explicitly running ipwmake in the appropriate source directory:

command	**requires**
etc/mc	-ltermcap
etc/rastool	<suntool/canvas.h> <suntool/scrollbar.h> <suntool/sunview.h> -lsuntool -lsunwindow -lpixrect
etc/xim	<X11/Xos.h> <X11/Xlib.h> <X11/Xutil.h> <X11/cursorfont.h> -lX11

Note that the distributed version of `lib/ipwenv` assumes that the `mc` command exists. If you cannot compile `mc`, then you should change the definition of `LC_WD` in `lib/ipwenv`.

Most shell scripts supplied with IPW require the `getopt` command, which is standard on many UNIX systems. If your system doesn't already have a `getopt` command, you should compile and install the public-domain version provided in `src/etc/getopt`.

If you will be developing your own IPW C programs, then you will probably want to compile profiling and debugging versions of the IPW libraries, and `lint` libraries for use with `ipwmake lint`. Run `ipwmake lintlib`, `ipwmake -D`, and `ipwmake -P` in the appropriate library directories.

CHAPTER 9: SHORTCOMINGS

In this chapter we will explore some of IPW's notable shortcomings. It should be noted that some of these are artifacts of the current implementation, while others are inherent in the design of IPW.

9.1. LACK OF A DATA DICTIONARY

IPW has no central data dictionary. While a data dictionary is normally associated with a database system, there are several aspects of IPW that, in retrospect, would have benefited from greater centralization, including:

- image header keywords;
- command-line options;
- units of measure.

The current implementation of IPW allows each image header type to maintain its own keyword name space. This has already led to duplication of some common keywords (e.g., ‘‘units’’) across header types. As headers become more varied, the odds of duplicate keywords having contradictory meanings can only increase.

Except for the use of `-H` to invoke a command's usage message, there is no enforced standardization of IPW command line options. For example, the `-i` option variously introduces:

- an input file name;
- the number of bits per pixel;
- the name of a linear-quantization interpolation function.

This is partly an inevitable consequence of using single-character options; however, there is little excuse for not standardizing on certain option letters for the most ubiquitous options (e.g., input file name).

Throughout IPW, different units of measure are used for the same phenomena. For example, angles are specified in various places in:

- decimal degrees
- degrees, minutes, seconds
- radians
- trigonometric functions

It is not always obvious from context which representation should be used. In retrospect, common units of measure should have been defined, and standard functions provided for converting between the standard and alternate representations.

All of these problems can be partially attributed to a desire to allow decentralized development. It was intended from the start that any IPW site could extend IPW, by adding primitives, creating new image header types, etc. However, it is clear that a mechanism is needed for ‘‘buying back’’ these scattered additions into a master copy of IPW, and resolving any conflicts thus created.

9.2. PROBLEMS WITH COMMAND-LINE ARGUMENTS

Several problems have become evident with the IPW command-line argument scheme. Type conversion of optargs is currently limited to a few built-in types (integer, floating-point, string, etc.). An alternative method would be to let optarg type conversion be handled by a user-supplied function [Allman 1989a], whose address would be included in each option descriptor. This would allow arbitrarily complex optarg types (subrange types, non-decimal integers, automatically-opened input files, etc.).

Limiting options to single characters adheres to existing UNIX practice and a proposed standard [Hemenway 1984] , but, as mentioned in the previous section, leads inevitably to overloading. At best, commands with several options will probably have at least one extremely strained mnemonic (e.g., `-b latitude`). Possible solutions to this include simply allowing multicharacter options, as does the IM Toolkit software [Paeth 1986a] , or using an alternate option syntax that does not conflict with the UNIX standard [Fenlason 1990].

A larger issue involves consistency in specifying multiple input files. There are several ways to do this, all of which are used by IPW:

- multiple operands: *command file1 file2 ...*
- multiple bands: `mux` *file1 file2 ...* | *command*
- multiple options: *command -opt file1 -opt file2 ...*

While it would appear that any one of these models could supplant the others, there are in fact good reasons for using all of them. The multiple-operands model is normally preferred, both because delivering all input data on a single stream meshes most naturally with the pipeline paradigm, and because it requires fewer open files than the multiple-operands or multiple-optarg models (file descriptors can be a scarce resource). The disadvantage of the multiple-operands model with respect to the other two models is that it requires an additional `mux` operation if the desired inputs are separate files[41].

For some operations, the multiple-operand model is arguably a more "natural" notation. For example, the current `bitcom` operates on multiple input bands. Yet a frequent application of bitcom is to apply a mask to an image, and the mask is almost certain to be stored separately, so an additional `mux` step is almost always required:

```
mux image mask | bitcom -a
```

versus

```
bitcom image mask
```

Having two versions of programs like `bitcom` seems an unsatisfactory solution to this problem.

The multiple-optarg model is appropriate in situations where there are different **types** of input. The easiest ways to distinguish these is by associating them with different options; e.g.:

```
convolve -i image -c kernel
```

However, it may also be appropriate to infer the data type from the input band order; e.g., `shade` assumes that its input band 0 is slope and band 1 is azimuth, since that is the output from `gradient`. It is not evident that this convenience should be sacrificed for consistency.

---

[41] `mux` itself is an example of an operation that **requires** the multiple-operand model.

9.3. SHORTCOMINGS OF THE DATA MODEL

Forcing the external pixel data to be byte-aligned buys processing efficiency at the expense of increased external storage. This may prove to have been a bad decision as both the average size and pixel precision of remotely-sensed imagery increases. To illustrate this with an example we have already encountered, AVIRIS images have 10-bit pixels, each of which IPW would store in 2 bytes, yielding 6 bits per pixel of non-data padding. In a standard AVIRIS half-scene of 256 lines, 614 samples, and 210 bands, these non-data bits would account for 24,756,480 bytes of external storage, and about 30 seconds of transfer time over an Ethernet with 800 KByte per second throughput (an optimistic value).

This storage overhead can be partially alleviated by use of file compression algorithms; for example, the UNIX `compress` utility [Welch 1984] deals particularly well with ‘‘dead space’’ in files. However, this is an extra processing step that can consume a great deal of processor time, especially during the compression (as opposed to decompression) phase. It may become necessary to modify IPW’s external storage scheme to use bit packing, although the tradeoffs in processing time and portability will have to be carefully weighed.

The quantized storage of floating-point pixels as unsigned integers has yielded one difficulty that appears unresolvable short of allowing floating-point pixels to be stored externally. Some operations cannot predict the range of output pixel values, and thus select the output quantization parameters before all of the output values have been generated. It is thus necessary for these operations to divert the output values to a scratch file, checking for minima and maxima as the output values are generated, and then read the output data back from the scratch file and write it to the output file. `lincom` is such an operation, since it cannot predict in advance the various combinations of pixel values that will occur in any input sample. Assumptions could be made using the most extreme values possible, but this would likely yield an unacceptably conservative quantization (i.e., loss of output precision). Thus far, this ‘‘two-pass’’ quantization has been deemed an acceptable price to pay for a single, portable external data format.

Every IPW primitive that has both input and output images must deal with the disposition of the input image headers. A problem arises when a header is copied to the output image when the primitive has modified the pixel data in a way that renders the header invalid. For example, the current implementation of `lutx` copies all of the input headers to the output image. It is easy to see that an input `lq` header will very likely be invalidated by the pixel value modifications performed by `lutx`. Yet, the only way to deal with this problem would be to build knowledge of `lq` headers into `lutx`. Given that new header types may be added to IPW at any time, it seems that building knowledge of specific headers into programs like `lutx` is a bad idea. The current implementation of IPW offers no easy solution to this problem, but it could be solved with an extension to IPW discussed in Chapter 10.

9.4. SHORTCOMINGS OF THE PROCESS MODEL

The strictly sequential processing model imposed by pipelines has some obvious disadvantages. The necessity to divert image data to a scratch file for multiple-pass operations has already been mentioned. Similarly, some apparently trivial operations, such as modifying an image header, require copying the entire image. We believe that neither of these are sufficient justification for discarding the pipeline model, but they are ‘‘hidden costs’’ imposed by the model that an application programmer or script writer should be aware of.

The exclusive reliance on lookup tables to implement univariate point operations can be clumsy at times. The doubling in size of a lookup table with each additional bit of input pixel precision is not always obvious, until a program like `lutx` fails due to memory exhaustion. Similarly, the use of text-based programs (e.g., `interp`) to generate lookup tables is practical for 8-bit values but prohibitively slow for 16-bit values; this problem will become more acute as the average pixel precision of remotely-sensed imagery increases [Esaias 1986, Goetz 1986]. Also, the primitives that manipulate lookup tables deal exclusively with quantized pixel values, whereas it is often desired to manipulate the de-quantized floating-point values.

Many IPW primitives use masks, but not completely consistently. Some primitives, such as `bitcom`, use an option to indicate that the last band is a mask, while other, such as `hist`, allow a mask to be specified as a separate file. This is largely due to masks being added late in the design cycle, but it should be standardized, both at the command-line level, and with library support for masking operations.

IPW neighborhood operations do not deal with image boundaries in a standard way. For example, `convolve` simply passes boundary pixels through unaltered to the output image, while `gradient` uses special variants of the slope and azimuth algorithms to calculate boundary values. Neither of these is completely satisfactory.

9.5. SHORTCOMINGS OF THE PROGRAMMING MODEL

The principal shortcoming of the IPW programmer's model has been excessive duplication of code. For example, there are substantial similarities between the routines that manipulate the various types of headers. A great deal of "boilerplate" code in IPW primitives could probably be generated automatically.

An obvious candidate for simplification is the IPW error handling facility. Currently almost all low level errors are passed back to the application programmer as failure codes, yet almost always the programmer's response is to call `error`, which terminates execution, with an error message that could usually have been generated automatically. A slightly more draconic treatment of errors by the library routines, by simply calling `error` directly when an error known to be unrecoverable occurs, would drastically simplify the programmer's interface to IPW. The loss of control could be somewhat mitigated by allowing the programmer to specify a customized low-level error handler, as an alternative to the standard IPW error handler.

CHAPTER 10: PRINCIPAL CONTRIBUTIONS

In spite of the criticisms of the previous chapter, we believe that IPW has largely proven to be a successful system. The underlying design principles, while not without shortcomings, have provided great flexibility in adapting the system to specific investigations, even in IPW's current experimental state. In this chapter, we highlight what we believe are IPW's salient contributions to the field of image processing for remote sensing and Earth science.

10.1. EXPLOITATION OF UNIX

Certainly one of the most successful aspects of IPW is its exploitation of the UNIX operating system. The UNIX command language, file system, and broad assortment of programming and text processing tools, have all been pressed into service. Several IPW command are UNIX shell scripts. Many of these scripts use the text processing language `awk` to perform complicated data transformations that would otherwise require compiled programs to implement. IPW is maintained with the UNIX tools `make` and `rcs`, and is portable largely because of the standardization of the UNIX programming environment. The decision to rely heavily on those parts of UNIX known to be portable has repaid us handsomely in the avoidance of duplicate, and in keeping the overall size of IPW down to an easily managed level.

10.2. PORTABILITY

Considerable effort has been expended to ensure that the IPW source code is as portable as possible across a variety of UNIX environments. We have been paid back by the relative ease of porting IPW to completely new architectures. This gives us considerable freedom in purchasing new image processing hardware, since we are virtually certain of being able to run IPW on it.

In addition to portable source code, IPW has a portable image data format, with ancillary data stored as printable ASCII text, and pixel data stored as unsigned binary integers. This has two benefits. It allows IPW images to be moved transparently between different hosts (subject to possible byte-swapping of multibyte pixel data, handled automatically by the `pixio` functions). Since the portable data format is a simple one, it also makes it fairly easy for IPW programs to communicate with programs outside IPW. It has thus proven fairly simple to ‘‘step sideways’’ from an IPW processing sequence, to take advantage of such non-IPW software as graphics and statistical packages.

10.3. PIPELINES AND PRIMITIVES

The central paradigm of IPW processing is a pipeline of low-level generic commands, or ‘‘primitives’’. This paradigm has proven robust and versatile. It allows new processing sequences to be constructed and tested quite rapidly. Components of a pipeline can easily be replaced if a better method becomes available. In a networked environment, computationally intensive components can be targeted to the most capable remote processors[42].

[42] We look forward to evaluating IPW on a tightly-coupled multiprocessor system.

The pursuit of fundamentality in the assembly of the basic set of primitives has led to some interesting idioms. The lookup table transform operator `lutx` has emerged as the basic univariate point operation; implementing a new operation now involves merely constructing the appropriate lookup table. The common operations of selecting a contiguous subset of an image for processing, in the spatial or spectral dimensions, has been implemented in the separate primitives `window` and `demux`, rather than in the IPW I/O system. The necessity of supporting the pipeline paradigm has led to implementing multivariate point operations as combinations of image bands, rather than as combinations of multiple images.

The implementation of the pipeline components as standalone UNIX programs has been a largely positive experience. Although it does lead to some duplication of startup and error-handling code, the benefits of ease of replacement, and of similarity to the way the rest of UNIX works, have far outweighed these minor shortcomings.

10.4. IMAGE DATA FORMAT

The adherence to a single image data format in IPW has greatly simplified the overall architecture of the system. In addition to not having to provide software support for a variety of formats, the presence of a single image format encourages the use of images for a variety of non-obvious applications. For example, IPW histograms and lookup tables are both stored as single-line images, which allows them to be edited, displayed, etc. by all of the IPW primitives.

BIP interleaving has paid off as we have begun to process imaging spectrometer data, which has hundreds of bands. Other interleaving schemes would require a program to maintain a prohibitive fraction of an image in memory in order to construct a spectrum for a single sample. Simple benchmarks have convinced us that BIP incurs no measurable additional overhead when compared with more popular interleaving schemes.

10.5. LINEAR QUANTIZATION

One of the most significant contributions of IPW is the promulgation of the linear quantization method for encoding floating-point values into binary integer pixels. Many image processing operations simply cannot be carried out entirely in the integer domain, yet external storage of binary floating-point data would cause insurmountable portability problems. The linear quantization method has allowed us to keep a portable (and easily displayed) image data format, and at the same time gives us greater control over the range and precision of the floating-point pixel values than we would have if we used binary floating-point data directly.

10.6. DISPLAY NOT REQUIRED

The independence of IPW from any particular display hardware has benefited us in two ways. First of all, it contributes to the freedom we have to select the most cost-effective image display hardware. Secondly, since IPW does not rely on the special processing capabilities of any particular display system, we need not display an image in order to process it. This may seem like a phantom benefit but in fact most of the image processing we do does not involve displaying the final output image. It would be a great imposition if we had to monopolize display hardware for a processing sequence that could easily take place in a ‘‘batch’’ mode.

10.7. USE OF NON-IMAGE DATA

IPW's ability to integrate point, vector, and polygon data with existing images has proved critical to several investigations. For example, IPW has been used to extrapolate snowpack properties over a digital elevation image of an alpine watershed, from a sparse set of points representing field snowpack measurements [Elder 1989]. This would have been impossible without some way to insert the field measurements into the image datasets.

The definition of the basic point data type as an ASCII (*location*, *value*) tuple simplifies the preparation of the point data for ingestion by IPW (much field data is already in this format). Point data are converted to line segments and polygons by a single primitive which interpolates image locations as needed. Beyond this there are no explicit point data manipulation tools in IPW, since these data may be easily processed with the standard UNIX text utilities.

CHAPTER 11: FUTURE DIRECTIONS

In this chapter we describe possible further work on IPW. Some of the actions suggested herein are direct responses to shortcomings noted in Chapter 9, while others are more general directions that seem to be indicated by the evolution of the environments in which IPW is used.

11.1. NEW PRIMITIVES

There are several needed primitives missing from the current implementation of IPW. Some of these are under active development, while others will hopefully be provided by users of IPW who are inspired by this document, or by their own needs.

The most obvious omissions from IPW are generic geometric primitives for image warping. At this site (UCSB) we have previously implemented a geometric processing suite that will be included in a future version of IPW. The basic design premise is that **mapping**, or the determination of correspondences between locations in two images, should be completely separate from **resampling**, or the computation of pixel values at non-integral locations. We have designed a primitive `resamp` whose inputs are a source image, and a 2-band **resampling map**, whose geometry is that of the desired output image, and whose pixel values are the line (band 0) and sample (band 1) coordinates of the corresponding sample in the source image. `resamp` interpolates the source image to obtain a pixel value at this (possibly non-integral) location, and places this value in the output image. The beauty of this scheme is that the generation of the resampling map is a completely separate problem, which may be attacked by a variety of methods (global functions, triangulation, etc.). For completeness, we have designed a `polymap` primitive which generates a resampling map based on a BIH and set of polynomial coefficients.

Additional utility geometric routines are needed. The problem of border effects in neighborhood operators could be solved by a primitive which padded an image with either extrapolated or constant borders equal to one-half the neighborhood size. Primitives like `dither` could be generalized by supplying a primitive which repeatedly tiled a small image (such as a dither matrix) up to the size of a target one; the two could then be combined by a multivariate point operation. Both of these ideas are borrowed from [Paeth 1986a].

IPW needs a simple way to generate constant images, both as test patterns, and as backgrounds onto which data can later be scribed with `edimg`.

IPW currently contains no frequency-domain routines. A minimal primitive would be a one-dimensional FFT, which could be combined with `transpose` to yield two-dimensional Fourier transforms.

The set of multivariate point operations needs a primitive to perform boolean combinations of input bands; i.e., the greater or lesser of all input values. An even more general primitive would be one that accepted an arbitrary algebraic expression to specify the input band combinations, but it is unclear how this could be efficiently implemented; in the HIPS image processing software system [Landy 1984] this was accomplished by actually recompiling the primitive for each new expression.

Certain non-image support primitives would be quite useful. For example, an early instructional version of IPW [Paddon 1988] included a program which read the variance-covariance matrix output by `mstats` and wrote normalized eigenvectors which could be used as coefficients with `lincom` to generate principal component images.

In addition to such ''bridge'' programs (i.e., programs that facilitate the connection of existing IPW programs), there are a variety of non-image-oriented operations that are essential to the analysis of remotely-sensed data. IPW has already been used to implement some of these, such as radiative transfer models for deriving atmospheric corrections [Li 1987]. We anticipate that generic specifications for such operations, analogous to the specifications of the existing IPW primitives, will have to be developed.

11.2. USER INTERFACE

IPW has deliberately avoided the issue of a modern user interface. The IPW primitives are logical targets for manipulation for high-level, window-based interface, but it seems pointless to build such functionality into every IPW primitive. Recently, however, there have appeared image processing systems which exploit the graphical construction of processing pipelines [Rasure 1987]. Another promising development is availability of generic window interfaces that are programmable at a level analogous to shell scripts [Perkins 1988, Musciano 1988], and could thus be bound to IPW primitives without a complete redesign of IPW. These appear to be logical directions for growth in the IPW user interface.

In the short term, some simple tools for locating existing IPW facilities, analogous the UNIX `apropos` and VMS `help` commands, would be extremely useful. These would accept less precise specifications than `ipwman`, allowing a user to locate a command or library routine by keyword or functional grouping.

The notion of masks needs to be much more fully integrated into IPW than it is now. Masks are currently supported on a program-by-program basis. What is needed is a unified model of masking operations, including multiband masks (i.e., an *N*-band mask applied bandwise to an *N*-band image), presented to the user in such a way that the masks become a natural means of expressing irregular regions in IPW. This will go a long way towards bridging the gap between IPW and vector-based GIS systems.

11.3. PROGRAMMER INTERFACE

Perhaps the most radical change in the IPW programmer interface will be effected by the widespread availability of the C++ language [Stroustrup 1986]. A migration to an object-oriented structure should help solve many of the problems related to code replication. We believe that future versions of IPW will migrate to C++, initially simply as an ANSI C environment.

IPW currently makes no use of compiler construction technology, yet there are some obvious candidates for it at the programmer level. A generalized algebraic expression parser could be put to use in an algebra primitive. Even more useful would be the development of a ''little language'' to describe external data formats, both IPW and foreign. This would greatly simplify the development of both new IPW headers, and of ingest programs for foreign data types.

11.4. DATA REPRESENTATION

For reasons outlined in §9.3, IPW will probably have to incorporate bit-packing of pixel data in its external data format. Alternatively, a more general means of reducing image storage overhead would be an entropy-preserving compression scheme optimized for image data [Rice 1983]. Studies of multispectral remote-sensing data have shown that compression ratios of up to 3:1 may be achieved by real-time algorithms using only a 2-pixel neighborhood [Chen 1987]. Both bit-packing and generic noiseless compression could be implemented in the `pixio` layer of the IPW I/O subsystem, and would not require any changes to existing application programs (save those which call the `uio` I/O layer directly); however, these compression schemes will be difficult to implement both portably and efficiently.

Full portability of IPW pixel data between networked heterogeneous machines needs to be implemented. At a minimum, this requires automatic byte-swapping of pixel data where necessary (the BIH already contains a `byteorder` field which can be used to implement this). In the long term, alternate data representations more suitable for a networked environment should be investigated [Sun 1987, Rew 1989].

APPENDIX A: CODING STANDARD

This standard contains rules and guidelines to which all IPW source code should conform. A working knowledge of the C language and the IPW and UNIX programming environments is needed to make effective use of this standard.

The standard is divided into three sections

- **style**
- **portability**
- **performance**

The style section is the most comprehensive, since it addresses the twin goals of **readability** and **maintainability**. IPW functions and programs are intended to be general-purpose tools, and as such must be understood, and often maintained (debugged and modified), by persons other than the author; thus, it is essential that code be designed from the beginning with the goal of easing the task of the human reader. The underlying purpose of the code must be evident in its structure, nomenclature, and formatting.

The ubiquity of the C language, and the uniformity of the UNIX interface, create the possibility of running identical IPW source code in a variety of environments. To realize this goal, however, **portability** standards are required, both to acquaint the programmer with legal constructs that are nonetheless empirically non-portable, and to specify how best to deal with non-portable code when its use is unavoidable.

Finally, since IPW programs are likely to be used with large image data sets, there must be **performance** standards that specify acceptable efficiency techniques that do not conflict with the more important style and portability standards.

The following textual conventions are used in this standard:

C language text that should be reproduced literally is shown in `constant-width` type.

C language text that should be replaced by the programmer is shown in *italic* type.

indent level refers to the tab stop at which a source line begins.

white space means any combination of the ASCII characters space, newline, or horizontal tab.

A.1. STYLE

The primary goal of the IPW style standards is **readable** code. Several techniques are employed to achieve this goal:

- straightforward implementation
- internal documentation
- uniform style

Simplicity is perhaps the most important readability technique: the less complicated a piece of code is, the easier it will be to understand, and the more confidently it will be used and modified. In describing the original implementation of UNIX, its chief designer noted that:

> "Throughout, simplicity has been substituted for efficiency. Complex algorithms are used only if their complexity can be localized." [Thompson 1978]

The style standards encourage simplicity by choosing the most straightforward path through the varieties of possible C implementations of common coding situations.

IPW source code should be self-documenting. The style standards encourage this in a variety of ways: through standard comment formats; through guidelines for name selection; and through guidelines for modularization, so that the overall program structure reflects the underlying problem.

Every effort is made to avoid variations in the appearance of the source code that do not directly correspond to some aspect of the underlying problem:

> "Every time we gave someone responsibility for a new module he or she would rewrite it according to his or her standards (allegedly to clean up the other person's bad habits). This process never converges and I feel that it is similar to the dog or wolf who stakes out his "turf" by urinating on each bush on its perimeter.
>
> Only coding conventions stop this process." [Stonebraker 1986]

For example, C programmers have traditionally been vociferous advocates of numerous styles of placing the `{}` around compound statements. Some common styles include:

```
control {                 control                   control
        statements        {                                 {
}                                 statements                statements
                          }                                 }
```

[Kernighan 1978, Minow 1984, Plum 1984]. It has yet to be demonstrated that any brace-placement style is clearly superior. However, most advocates of one particular style would agree that use of a **single** style (even if not their favorite) is important; otherwise, the reader of the code must decide whether or not a sudden change in brace-placement style is indicative of some logical change in the code. In such cases, the style standards arbitrarily mandate the use of one out of several variant renderings. Of course, in situations where one of several otherwise equivalent constructs is clearly the most readable, use of that construct is mandated.

Many of the more trivial style standards (format of comments, use of white space, etc.) can be implemented mechanically. An initialization file for the C source code formatting utility `indent`[43] is provided in `$IPW/pub/indent.pro`; use of this file causes `indent` to adhere to most of the IPW style standards.

A.1.1. Comments

Comments are intended to aid in the understanding of the code. You may assume that the person reading the comments knows the C **language** about as well as you do; however, you may NOT assume that he/she understands anything about the particular **problem** that your code is trying to solve. This leads to the general rule, "comment the purpose of the code, not the implementation" [Lapin 1987]. Extraneous comments about obvious features of C (e.g., `/* increment i */`) will cause the reader of your code to "tune out" the comments altogether.

There are four distinct styles of comments used in IPW source code:

- **End-of-line comments** are allowed ONLY on declarations and `#defines`. The `/*` of an end-of-line comment begins at tab stop 5 (column 41), or the next

[43] free software from UC Berkeley; optionally distributed with IPW

available tab stop. The `*/` of an end-of-line comment begins at tab stop 9 (column 73), or the next available column. End-of-line comments may NOT span the end of the line; this can cause misleading line numbers in `lint` and `cc` error messages.

- **One-line comments** are used to annotate a specific property (e.g., a spurious `lint` error) of the immediately following statement (no intervening blank lines). One-line comments are preceded by at least one blank line, and always begin in column 2.
- **Block comments** separate “paragraphs” of code. There should be no blank lines preceding or following a block comment, that would not otherwise be required by these standards. The `/*` of a block comment is on its own line and begins in column 2. Each line of a block comment has a `*` before any text. The `*`s are vertically aligned with the first `*` in the first line of the block comment. The last line of a block comment is the closing `*/`.
- **Header comments** appear before any global definitions. Header comments begin in column 1.[44] The leading `/*` and trailing `*/` are on their own lines. Each line in the body of the comment begins with a `**`. A header comment is preceded and followed by at least one blank line.

Examples:

```
#define ERROR           (-1)            /* description of "ERROR"       */

/*
** NAME
**      foo -- a nonsense function
**
...
*/

int
foo(bar)
        int             bar;            /* description of "bar"         */
{
        ... C code ...
 /*
  * a block comment describing the purpose
  * (as opposed to the implementation)
  * of the next several lines
  */
        ... C code ...

 /* a one-line comment describing the next statement */
        ... C code ...
}
```

[44] `indent`, when initialized by `$IPW/pub/indent.pro`, will not reformat comments that begin in column 1.

The top-level unit of documentation is a globally-scoped (i.e., defined outside any block) function or variable. Each **module** (compiled source file) should contain exactly one global object.[45] Preserving a one-to-one correspondence between modules and linkable objects minimizes the amount of unused code that gets linked into an executable program.

All global objects in the code must be commented in the style of a UNIX manual page, using the IPW header comment style. As a rule of thumb, these comments should be sufficiently detailed to allow the reader to use the object without reading the rest of the code. In fact, IPW manual pages are generated automatically from these comments by the `ipwman` command.

Procedural code should be thought of as consisting of "paragraphs" of C statements. The purpose of each paragraph of statements should be described in a preceding block comment. Block comments should be terse, and should amplify, NOT simply re-state, the following code. In particular, comments should not contain pseudo-code: the actual code should be as readable as any pseudo-code description.

If a paragraph of code implements a published algorithm, then the preceding block comment should contain a full reference to the publication, including page and equation numbers.

One-line comments should not be used where a block comment would achieve the same purpose, since the change in comment styles is inherently distracting. Instead, one-line comments should be used chiefly to call attention to "gray areas" in the code, which might require maintenance. The most frequent use of one-line comments in IPW is as directives to the `lint` utility (e.g., `/* VARARGS */` before the definition of a function with a variable number of arguments.) The need for more general documentation at the one-line comment level should be obviated by clear code and mnemonic identifiers.

Variable definitions always contain an end-of-line comment. If the permissible values of the variable are restricted to a subrange of its type, then the subrange should be specified in the comment. Similarly, any physical units associated with values of the variable (meters, degrees, etc.) should be documented in the end-of-line comment, unless they are obvious from the variable name.

Sample header comments for `main`s, functions, global variables and header files are given below. Non-mandatory sections of the comments should be omitted if they would otherwise be empty. All header comment text other than the section headings is indented from the leading asterisks by at least 1 tab stop. Follow these formats carefully: `ipwman` depends on them to function properly.

A.1.1.1. Program header comments

The following section headings may appear in the header comments for a `main` function, in the order listed:

[45] A module may contain more than one function if those functions share one or more top-level `static` objects; see §A.1.8.

```
/*
** NAME
** SYNOPSIS
** DESCRIPTION
** OPTIONS
** DIAGNOSTICS
** FILES
** EXAMPLES
** SEE ALSO
** NOTES
*/
```

The `NAME` section contains a single line containing the name by which the program SHOULD be known (which may not be the name of the actual installed binary version of the program), followed by `--`, followed by a "verb+object(s)" description of the program:

```
** NAME
**      hist -- compute image histogram
**
```

The `SYNOPSIS` section contains a minimal schematic description of the program's command line; listing all possible options and operands. Options and operands which are not mandatory are enclosed by `[]`s:

```
** SYNOPSIS
**      hist [-m mask] [image]
**
```

Note the use of symbolic values for option arguments and operands. Subsequent references to these symbolic values are delimited by `{}`s, to indicate that the symbolic values will be replaced by actual, user-specified values when the command is run.

The `DESCRIPTION` section contains a brief narrative description of the program:

```
** DESCRIPTION
**      Hist reads {image} (default: standard input) and computes
**      its histogram.  The histogram is written to the standard
**      output as a single-line IPW image, whose sample offsets
**      represent the pixel values in the input image, and whose
**      pixel values are frequency counts.
**
```

The optional `OPTIONS` section contains a list of each option the program accepts. Each option is described by an indented paragraph with the option as a hanging tag:

```
** OPTIONS
**      -m      histogram only those pixels masked by nonzero values
**              in the image {mask}.
**
```

The optional `DIAGNOSTICS` section contains a description of any "non-obvious" error messages that the program may generate, especially those related to otherwise unstated restrictions on the program's options or input data. The actual text of the error message is reproduced, followed by an explanation indented 1 tab stop. Variable portions of the error message text are delimited by {}s.

```
** DIAGNOSTICS
**      "band {number} has too many bits per pixel"
**
**              The precision of the specified band is such that a
**              histogram of that band would not fit in memory.
**
```

The optional FILES section contains a list of any "hidden" files that the program will access (i.e., not specified explicitly on the command line). Each filename or filename pattern is followed by explanatory text, indented 1 tab stop. Variable portions of a filename (e.g., such as would be replaced by a process-ID) are delimited by {}s. Environment variables are indicated by a leading $.

```
** FILES
**      $TMPDIR/hist{NNNNN}     scratch file
**
```

The optional EXAMPLES section contains one or more examples of UNIX command lines invoking the program. Especially useful are examples that demonstrate often-used pipelines combining this program with other IPW programs:

```
** EXAMPLES
**      The following pipeline will plot the input image's histogram
**      on the standard output:
**
**              hist image | grhist | plot
**
```

The optional SEE ALSO section contains a comma-separated list of related IPW modules whose header comments may prove useful to someone attempting to understand this program. Following this list, any relevant references to the published literature should appear (e.g., a paper describing the algorithm the program employs):

```
** SEE ALSO
**      grhist, histeq
**
**      Frew, D.H. (1989) "Thaumaturgic acceleration of image
**      histogram computations,", COG Tech. Report #7, Berkeley, CA
**
```

The optional NOTES section contains any additional information of which a user of the program should be aware, including, but not limited to, known bugs or deficiencies in the current implementation, comments on algorithms or heuristics employed, or plans or suggestions for future enhancements. The NOTES section may also contain information about restrictions on the use of the program that are not discussed in the DIAGNOSTICS section.

```
** NOTES
**      The number of bits per output pixel is the minimum
**      necessary to accommodate the largest value in the
**      histogram.
**
```

As a rule, any information that is maintained by the RCS version control software (modification dates, authors' names, etc.) should NOT appear in the header comments. This information will appear elsewhere in the source code, and can be extracted along with the header comments for further formatting.

A.1.1.2. Function header comments

The following section headings may appear in the header comments for a library function, in the order listed:

```
/*
** NAME
** SYNOPSIS
** DESCRIPTION
** RETURN VALUE
** ERRORS
** GLOBALS ACCESSED
** APPLICATION USAGE
** SEE ALSO
** NOTES
*/
```

(Sections whose comments are the same as previously described are omitted from the following discussion.)

The `SYNOPSIS` section contains an old-style C prototype for the function.[46] The appropriate `#include` statements should also appear here if any header files (besides `ipw.h`) must be included by the function's caller:

```
** SYNOPSIS
**      #include "bih.h"
**
**      BIH_T **bihread(fd)
**      int fd;
**
```

The optional `RETURN VALUE` section contains a brief description of the function's possible return values:

```
** RETURN VALUE
**      pointer to array of BIH_T pointers; else NULL if EOF or
**      error
**
```

The optional `ERRORS` section contains a description of any IPW errors that are logged

[46] ANSI prototypes are not (yet) portable.

by the function, when the return value indicates an error. The format is the same as for the DIAGNOSTICS section mentioned previously.

```
** ERRORS
**      "{name}" header: bad band "{band}"
**              The band number is outside the range previously
**              specified in this header
**
```

The optional GLOBALS ACCESSED section contains a list of any global variables accessed by this function:

```
** GLOBALS ACCESSED
**      _bih[fd]        bihread's return value is also stored here
**
```

The APPLICATION USAGE section contains a brief description of the context in which an application program might call the function.

```
** APPLICATION USAGE
**      bihread is called by application programs to ingest BIH
**      headers.
**
```

The NOTES section contains, in addition to the topics already mentioned, a description of any state information that the function preserves between successive calls (i.e. in static variables).

A.1.1.3. Variable and header file header comments

The following section headings may appear in the header comments for a global variable or header file, in the order listed:

```
/*
** NAME
** SYNOPSIS
** DESCRIPTION
** APPLICATION USAGE
** SEE ALSO
** NOTES
*/
```

(Sections whose comments are the same as previously described are omitted from the following discussion.)

The SYNOPSIS section contains either a sample #include directive for the header file:

```
** SYNOPSIS
**      #include "bih.h"
**
```

or a sample declaration for the global variable:

```
** SYNOPSIS
**      #include "_pixio.h"
**
**      extern PIXIO_T *_piocb[];
**
```

A.1.2. Names

Variable names should be self-documenting; e.g., `nsamps` for ‘‘number of samples’’, `i_fd` for ‘‘input file descriptor’’, etc. The meaning of a variable name, and the usage of the corresponding variable, should be invariant throughout the program.

Names of constant macros and ‘‘unsafe’’ function macros (those that cause any of their arguments to be evaluated more than once) should not contain any lower-case alphabetic characters. Other identifiers, including ‘‘safe’’ function macro names, should not contain any upper-case alphabetic characters. Word breaks in names should be indicated by an underscore, not by capitalization.

Example:

```
#define PIPE_BUF        4096

#define DIM(a)          ( sizeof(a) / sizeof(a[0]) )
#define strsize(s)      ( strlen(s) + 1 )

static int      line_number;            /* right                */
static int      lineNumber;             /* wrong                */
```

In general, case alone should not be used to distinguish names — portability considerations aside, the distinction is not strong enough visually.

Global names must be longer that 1 character. However, the following traditional 1-character local names may be used:

`c`	ASCII character
`i,j,k`	array or loop index
`m,n`	count or loop limit
`p,q`	pointer
`s`	pointer-to-string

The following 1-character prefixes are also widely used in IPW source code:

`i_`	input image
`m_`	mask image
`n`	number of
`o_`	output image

The suffix `p` indicates ‘‘pointer to’’. A double-indirect pointer is indicated by `pp`.

Long names should differ from one another by more than 2 characters, or by an obvious prefix (not suffix). Certain characters that are easily confused (e.g., `1`, `l`, and `I`; and `0`, `O`, and `Q`) should not be relied on to distinguish different names. Finally, possible phonetic confusion between a name’s ‘‘sound’’ and its meaning should be avoided; a classic example is `inch` for ‘‘input character’’ [Plum 1984].

Function names should follow the standard pattern of "object + verb"; e.g., `geohmake` constructs a new image geodetic header. Similar functions should have similar names; e.g., *xxx*`hmake` is the common pattern for functions that construct image headers.

A.1.3. White space

Properly used, white space in source code contributes significantly to its readability. However, lack of standard usages for white space can lead to confusing stylistic variations. The following standards are admittedly arbitrary, but adherence to them helps minimize distracting and essentially spurious variations in the appearance of the IPW source code.

White space is MANDATORY in the following circumstances:

- preceding and following C keywords (except at the beginning and end of a cast)
- preceding and following binary operators
- following `;` and `,`

White space is PROHIBITED in the following circumstances:

- preceding `;` and `,`
- between a function name and `(`
- on the concave side of a `(` or `)`
- between a unary operator and its operand

Blank lines are MANDATORY in the following circumstances:

- before each `case` or `default` in a `switch`

Blank lines are PROHIBITED in the following circumstances:

- within an expression or simple statement

Source lines that are too long must be broken by inserting a newline, after either a comma or a binary operator. The continuation line should be indented 1 space more than the immediately preceding `(`, or 2 spaces more than the immediately preceding `=`, as appropriate.[47]

Examples:

```
printf("Usage: %s options ...\n",
       progname);

if (a < b  &&
    a > A_MAX) {
        ...
}

result = f(a) +
         f(a * b);
```

The use of embedded form-feed characters to paginate source code is discouraged: it is to easy to add or delete source lines without remembering to adjust the pagination, and

[47] This level of control over indenting is difficult to achieve with `indent` alone.

readily-available tools exist to paginate C source code automatically.

A.1.4. Indentation and braces

A standardized style of indentation and brace placement enhances readability by making the physical layout of the code reflect its logical structure. The style used in the IPW source code is derived from the style popularized in the original C language textbook [Kernighan 1978].

A { is placed on the same line as the associated control statement, preceded by at least one blank. A } is placed on a line by itself at the same indent level as the associated control statement (except for the `do-while` construct; see example). The indent level between braces is increased one tab stop.

Examples:

```
for (i = 0; i < n; ++i) {
        (void) foo(i);
}

while ((c = getchar()) != EOF) {
        putchar(c);
}

do {
        (void) foo(i);
} while (--i > 0);

if (i < 0) {
        lt0 = TRUE;
}
else if (i == 0) {
        is0 = TRUE;
}

switch (option) {

case ’b’:
        bflag = TRUE;
        break;

default:
        error("%c: bad option", option);
}
```

Braces may NOT be elided for one-line blocks. One often needs to add statements to a block, and having to remember to add the braces as well unnecessarily complicates maintainability.

For function definitions, both braces are placed in column 1, and the formal argument declarations are indented one level. Any storage class or type specifiers are separated from the function name by a single newline.

Example:

```
int
abs(i)
        int             i;
{
        ...C code...
}
```

Braces may NOT be elided in multidimensional aggregate initializers. The innermost } in an aggregate initializer MAY be on the same line as its corresponding {. In general, the layout of an aggregate initializer should reflect the structure of the aggregate.

Examples:

```
fpixel_t        lap_kernel[3][3] = {    /* Laplacian kernel      */
        {0.0, -1.0, 0.0},
        {-1.0, 4.0, -1.0},
        {0.0, -1.0, 0.0},
};

int             bytebits[] = {1, 2, 4, 8, 16, 32, 64, 128};
```

A.1.5. The C preprocessor

The C preprocessor is a powerful tool for enhancing the readability and maintainability of C source code. Use of macros to parameterize literal constants allows those constants to be easily changed should the need arise, and is a useful form of documentation. Collecting oft-used macros and declarations into header files simplifies their maintenance.

Within a source file, preprocessor directives are grouped and ordered as follows:

- system-level `#include`
- IPW `#include`
- local (program or module) `#include`
- `#define` constants
- `#define` macros

Each group is preceded by a blank line.

Preprocessor directives always begin in column 1. There is no whitespace between the # and the directive name.

In `#define` directives, the macro name begins at tab stop 1 (column 9) and the replacement text begins at tab stop 3 (column 25), or at the next available tab stop. In all other directives, a single space separates the directive from the subsequent expression, filename, etc.

Examples:

```
#include <stdio.h>

#include "ipw.h"

#include "lutx.h"
```

... header comments would go here ...

```
#define LUT_NBITS       8
#define LUT_SIZE        (1 << LUT_NBITS )

#define VALID_IDX(i)    ( (i) >= 0  &&  (i) < LUT_SIZE )
```

A.1.5.1. Macros

As a general rule, source code should contain no embedded literal constants. The two principal exceptions to this are strings controlling formatted output (e.g., in `printf` calls), and small integers used in loop indices or relative subscripts. All other constants should be replaced by `#define`d macros. Not only does this make the code vastly easier to modify, but if the macro names are carefully chosen, they can contribute significantly toward making the code self-documenting.

Constants whose values depend on the values of other constants should be so defined.

Example:

```
#define N_PIXEL_BITS    32                              /* wrong */
#define N_PIXEL_BITS    ( sizeof(pixel_t) * CHAR_BIT )  /* right */
```

Constants whose values cannot be changed (i.e. they exist as macros only for readability) should be appropriately commented.

Example:

```
#define FALSE           0               /* boolean F: MUST BE 0 */
```

Use of macros to extend or redefine the syntax of C, or to redefine C keywords,[48] should be avoided. Someone who knows C should not have to learn some strange new dialect in order to understand your code.[49]

Any macro replacement text containing operators should be fully parenthesized, including an outermost set of parentheses around the entire replacement text. Arguments in function macro replacement should also be parenthesized, in case they expand into expressions containing operators.

---

[48] Some redefining of keywords is necessary for portability. All of this should be handled by `ipw.h`.

[49] As did UNIX system programmers confronted with the source for the Version 7 `sh`, written in a bizarre mix of C and Algol 68, the latter mapped into C by macros.

Function macros should in general not contain C keywords or blocks of statements; this hinders readability by concealing the logic of the program.

Although conditional compilation is primarily a portability tool, two applications are important in code maintenance. To temporarily disable a section of code, use the construct

```
#if 0
/* code to be disabled */
#endif
```

instead of `/*...*/`; this avoids the problem of nested comments.

All library functions should contain, in the same source file, a `main` routine that can be conditionally compiled with the function to perform stand-alone tests. IPW uses the symbol `TEST_MAIN` to control such compilation.

Example:

```
int
func()
{
        /* code for func() */
}

#ifdef TEST_MAIN
main()
{
        /* code which tests func() */
}
#endif
```

This code should appear at the end of the source file.

A.1.5.2. Header files

Header files should contain any `#define`s, `typedef`s, or `extern` declarations that would otherwise be duplicated across multiple source files in a program or library.

Header files should be functionally organized; e.g., `geoh.h` contains the declarations, `typedef`s, etc. needed to manipulate geodetic image headers. Header files should not include other header files;[50] this tends to obscure dependencies between different components of IPW, of which the reader of the code ought to be aware. Of course, the header comments in a header file should mention any other header files that the header file depends on (except `ipw.h`, which all header files are assumed to require).

In the `#include` directive, UNIX system header file names should always be enclosed in `<>`s; while IPW header file names should always be enclosed in `""`s. Note that all system header files normally needed by IPW programs are included automatically by `ipw.h`.

[50] `ipw.h` is an exception, since it must conditionally substitute IPW header files for nonportable system header files.

The first and last lines in a header file should comprise a conditional "wrapper", which prevents the file from being `#included` more than once in the same compilation:

```
#ifndef H_xxx
#define H_xxx
...
/* H_xxx */
#endif
```

where *xxx* is the capitalized name of the header file, without the `.h` extension.

`#include` directives should always be near the top of a source file, immediately preceding the header comments. This serves to emphasize the module's dependencies on the "outside world." `#define` directives may precede a `#include` if they would modify the included file's behavior; however, with the exception of the conditional compilation controls in `ipw.h`, this should not be necessary. `ipw.h` is always the first in a sequence of `#include` directives.

A.1.6. Declarations

No more than one name should appear in a declaration. The ease of modification this affords is well worth the extra text consumed.

Local objects should be declared close to their point of first use. In practice, this implies that block-level declarations should be used freely, and that individual blocks should be kept short. Local declarations should never override declaration made at a higher level.[51]

Local function declarations should be avoided, by one of the following techniques:

- Declare all functions at the top level, either in header files or near the beginning of the source file.
- Define `static` functions before global functions in the source file, so there are no forward references.

Declarations are indented at the same level as the block in which they occur. A storage class specifier, if present, is separated from the following type specifier by a single blank. The identifier is indented 2 tab stops from the prevailing indent level. In pointer declarations, identifiers with leading `*`s may be back-indented the appropriate number of spaces so that the first character of the identifier name begins at the tab stop. An initializer, if present, is separated from the identifier by a single space.

[51] `lint` will catch this.

Example:

```
{
        static bool_t   already = FALSE;/* ? already called      */

        pixel_t        *buf;            /* -> image line buffer  */
        int             fd = ERROR;     /* image file descriptor */
        int             line;           /* image line #          */
        int             nlines;         /* # image lines         */

        ... C code ...

        for (line = 0; line < nlines; ++line) {
                char           *bufp = buf;

                ... C code ...
        }
}
```

Long integer constants in initializers should always include the trailing `L`. Global and `static` variables that are not explicitly initialized will default to 0. If 0 is not a sensible value for such a variable, then it should be explicitly initialized.

If the declaration of an aggregate contains an initializer, then the initializer must be fully specified; don't omit trailing `0`s. The only exception to this is a top-level global variable definition in a file by itself, which is being initialized only to force storage allocation (see **Portability** below).

Variable and function declarations are grouped according to the following criteria, and appear in the following order. Each group of declarations is followed by at least one blank line.

- `extern` variable declarations
- `extern` function declarations
- `static` variable definitions
- `static` function definitions
- global variable definitions
- global function definitions

Within a block, definitions are grouped and ordered as follows:

- `static` definitions
- local (including `register`) scalar definitions
- local aggregate definitions

`typedef` names are lower-case, ending in `_t`, for scalar types; or upper-case, ending in `_T`, for aggregate types. `typedef` statements should always be top-level, and are formatted like ordinary declarations.

Examples:

```
typedef int     bool_t;

typedef struct node {
        int             datum;
        struct node    *prev;
        struct node    *next;
} LIST_T;
```

Macros which represent possible values of a structure member should be `#defined` immediately following the structure's `typedef`.

The `auto` keyword is always superfluous and should not be used.

A.1.7. Types

`typedef`s in IPW are used mainly as a portability tool. Where portable uses exist for the standard C data types, those types are used directly: loop counters, file descriptors, and function return codes are all `int`s; character strings are stored in arrays of, or manipulated by pointers to, `char`s; etc. This is intended to help make IPW code more readable to programmers with previous C experience. However, there are some circumstances where `typedef`s contribute significantly to readability.

Boolean variables should always have the type `bool_t`, even though C by definition treats boolean expressions as having type `int`. Generic pointers that will be cast to a specific type before dereferencing should have the type `voidp_t`, even though C guarantees that such pointers may always be safely typed `char *`. These applications of the base type are specialized enough to warrant defined types simply for readability's sake.

If standard C types are being used in an especially restricted context, then the comment at the end of the corresponding declaration should mention the restriction. One standard way of doing this is to indicate the subrange of permissible values [Plum 1984].

Example:

```
float           azimuth;                /* radians: -pi..pi      */
```

Default typing (allowing function values and formal arguments to default to `int`) should NEVER be used; this is one of the most common sources of bugs in C code.[52]

The `void` type should always be used to cast the result of function whose value is ignored; this forces the programmer to acknowledge that a return value is being deliberately discarded.

All structures should be `typedefd`, and thereafter only referred to using the `typedef` type. IPW image headers are typical examples of this; see any `$IPW/h/`*xxx*`h.h` file. Note that in a `typedefd` structure the tag is superfluous and should be omitted, unless the structure is self-referential, in which case the tag will be needed in one or more member declarations.

[52] `lint` will catch these, if you're lucky ...

IPW programmers are encouraged to create their own types if doing so contributes to the readability of the program. They should first familiarize themselves with the default types provided in `$IPW/h/typedef.h`, and in any other specialized header files they are using. In particular, structure types contribute to readability when they replace multiple arrays with common indices.

A.1.8. Use of `static`

The `static` keyword is unfortunately overloaded in C, affecting both the visibility and the extent of a declaration [Harbison 1987]. In IPW, there are three situations in which the `static` storage class may be used.

Top-level definitions may be made `static` to keep them from being accessed outside the module in which they appear.

Local variables may be made static if they are aggregates that must be initialized, or if their addresses are required to initialize other aggregates.[53] This use of `static` is common in IPW; e.g., with the `OPTION_T` type to collect program arguments, and with the `GETHDR_T` type to control input image header processing.

Example:

```
{
        static OPTION_T opt_b = {
                'b', "begin line,sample",
                REAL_OPTARGS, "coord",
                OPTIONAL, 2, 2
        };

        ...

        static OPTION_T operands = {
                OPERAND, "input image file",
                STR_OPERANDS, "image",
                OPTIONAL, 1, 1,
        };

        static OPTION_T *optv[] = {
                &opt_b,
                ...
                &operands,
                0
        };
}
```

Finally, local variables may be made `static` to preserve their values between calls to the function in which they are defined. This method is used chiefly for one-time initialization of dynamic data structures:

[53] ANSI C permits the initialization of automatic aggregates, but only with constants (e.g., `static` addresses).

Example:

```
f()
{
        static bool_t   already = FALSE;

        if (!already) {
                already = TRUE;
 /*
  * initialization code goes here ...
  */
        }
}
```

Local `static` variables should never be used as ‘‘implicit’’ arguments (i.e., to affect the behavior of the function as perceived by its caller.)

`static` definitions should not appear in nested blocks (i.e., anywhere within a block except at the beginning of a function).

A.1.9. Expressions and statements

There should be no more than one statement per line of source text. The preferred place to break up expressions that won’t fit on a single line is before the lowest-precedence operator in the expression.

The conditional operator `?:` should be avoided, unless it results in substantially more readable code than the equivalent `if-else`.

The comma operator should not be used to substitute for a block of statements. The only recommended use of a comma operator is in the initialization or increment clauses of a `for` loop.

Example:

```
/* reverse the order of chars in s */
for (i = 0, j = strlen(s) - 1; i < j; ++i, --j) {
        char            tmp;

        tmp = s[i];
        s[i] = s[j];
        s[j] = tmp;
}
```

Side effects which rely on the evaluation order of && or `||` should be avoided.

Example:

```
/* bad */
if (i_filename != NULL
    && (i_fd = uropen(i_filename)) == ERROR) {
        error("can’t open %s", i_filename);
}

/* good */
if (i_filename != NULL) {
        i_fd = uropen(i_filename);
        if (i_fd == ERROR) {
                error("can’t open %s", i_filename);
        }
}
```

In this example, the recommended form emphasizes that:

- `i_fd` is set only when `i_filename` is non-null.
- The error message is a direct consequence of the call to `uropen`, but only an indirect consequence of whether `i_filename` is `NULL`.

Expressions with side effects should NEVER appear in a function argument list.

Always use explicit logical operations in a logical expression; don’t rely on a logical expression being implicitly testing against `0`. Test against the default macros `TRUE` or `FALSE` if no other test is appropriate.

Logical expressions are usually clearer if they are reformulated to eliminate a leading `!`.

Example:

```
/* bad */
if (!(i < 0 || i >= n)) {
        ...
}

/* good */
if (i >= 0 && i < n) {
        ...
}

/* acceptable */
if (!isascii(c)) {
        ...
}
```

Embedded assignment expressions should be avoided, except where necessary for loop control, in which case it is ESSENTIAL that the assignment expression be fully parenthesized and explicitly tested.

Example:

```
/* BAD */                         /* good */
while (c = nextc()) {             while ((c = nextc()) != EOS) {
        ...                               ...
}                                 }
```

Confusion about operator precedence in C is a rich source of bugs. In general, expressions should be fully parenthesized if they contain:

- both `||` and `&&`
- multiple bitwise operators
- any mixture of logical, bitwise, and arithmetic operators

Use of white space should not suggest a misleading operator precedence.

Example:

```
a+b * c
```

A.1.10. Control structures

The most fundamental C control structure is the **block.** A nested block (i.e., other than the outermost block of a function) should be neither too long to fit on a single printed page,[54] nor so deeply nested most of its statements are multi-line.

Excessive nesting can often be controlled by use of C's "disguised GOTOs" — `continue`, `break`, and `return`. The clarity these provide by minimizing nesting should always be weighed against their disruption of a simple top-to-bottom flow of control. `goto` itself should hardly ever be used,[55] and then only to branch forwards, and never into a block at an equivalent or deeper nesting level.

Sequences of blocks at the same level (e.g., `if-else`) should be arranged so the shortest blocks appear first, if their ordering is otherwise unimportant.

The `switch` is a special case of `if-else`, and should only be used if:

- the `case`s are mutually exclusive
- the order in which the `case`s are considered is unimportant
- a single integer expression, evaluated once, is being tested
- the code is significantly more readable than the equivalent `if-else`.

The `case`s in a `switch` should each be on a separate line. The `default` case should always be last, and should always be present; if there is no sensible default, then the `default` case should contain an appropriate call to `bug`. The code for each case (including `default`) should end with either `break`, `continue`, `return`, the comment `/* FALL THROUGH */`, or a call to a function that never returns (e.g., `exit`, `error`, etc).

---

[54] A reasonable limit is 56 lines, the pagination unit of the UNIX utility `pr`.

[55] There are 5 `goto`s in 22,000 lines of IPW source code.

Loops should be designed so that there is minimal interaction between the control statements and the loop body. Control variables should be modified in either a control statement or the loop body, but NOT in both. Control variables should not be assumed to contain any particular value after the loop terminates, unless computing this value is the sole purpose of the loop.

Counters used to control loops should be able to handle the "off-by-one" value that they will usually contain when the loop terminates.

`do-while` loops should usually be rewritten as `while` or `for` loops, unless:

- the control variable is explicitly set immediately before the loop is entered.
- you really do want the loop to always execute at least once.

Often there is a temptation to push all the work being done by a loop into its control statements, especially in a `for` loop. This should be avoided — the loop control statements should be limited to controlling whether the loop is executed:

Example: sum of all elements of `array`

```
/* BAD */
for (i = 0, sum = 0; i < n; sum += array[i], ++i) {
        continue;
}

/* GOOD */
sum = 0;
for (i = 0; i < n; ++i) {
        sum += array[i];
}
```

If the desired outcome of a loop is still entirely a consequence of the execution of its control statements, then the loop body should contain a single `continue` statement, rather than a `;` by itself:

Example: largest power of 2 less than `n`

```
/* BAD */                           /* GOOD */
for (i = 1; i < n; i *= 2) {        for (i = 1; i < n; i *= 2) {
        ;                                   continue;
}                                   }
```

The way in which a loop terminates should be obvious from its syntax. In almost all cases, a straightforward `while`, `for`, or `do-while` will suffice. Occasionally loop termination is controlled by a condition tested within the loop body; this unusual construct should be clearly commented.

Example (based on [Dromey 1982]):

```
/*
 *             n
 * accumulate x  in product, for integer n > 0
 */
        product = 1.0;

        for (; ; ) {
/*
 * extra multiplication if n is odd,
 * before we divide away the low-order bit
 */
                if (n & 01) {
                        product *= x;
                }
/*
 * reduce number of multiplications
 * by exploiting (x**2)**(n/2) == x**n
 *
 * terminate before unnecessary x = x**2,
 * to avoid possible overflow
 */
                if ((n /= 2) == 0) {
                        break;
                }

                x *= x;
        }
```

A.1.11. Modules

The fundamental modular unit in C is the source file. Compiled source files are the basic units which can be linked together to form executable programs.

IPW tries to maintain a one-to-one relationship between source files and global identifiers (functions or variables). This eliminates the possibility that an executable program will contain unused modules, since only identifiers that are explicitly referenced will be loaded. Of course, source files may also contain an arbitrary number of top-level `static` identifiers, since they are invisible during the linking process.

Source files are named after the most significant global identifier (variable or function) they contain. This makes it easier for maintainers to find the source file associated with a particular identifier.

There are some general rules for deciding how to modularize a particular programming problem. First of all, the overall problem should be divided into levels of abstraction. Functions at one level should ideally call only functions at the same or next lower level. Global data structures should likewise be allotted to particular levels, and not accessed by functions outside those levels.[56] A good example of this model is

[56] As program size becomes less of an issue, the benefits of a 1:1 mapping between globals and

the IPW scheme for image header I/O, in which the various layers (header-specific, hdrio, and uio) each have associated global data structures, and each call only the next lower layer:

layer	functions	data structure
BIH	bihread, bihwrite	BIH_T **_bih[]
hdrio	hrname, hgetrec, hwprmb, hputrec	HDRIO_T _hdriocb[]
uio	ubof, ugets, uputs	UIO_T _uiocb[]

Code should not be duplicated if there is any possibility of sharing it. If you find yourself doing the same thing in two or more functions, it's worth it to generalize that action into a separate function: there will be fewer lines of code to keep track of, fewer duplicated fixes to make if the code proves buggy, and a smaller overall program. The function hstrdup is an example of this: it is essentially identical to strdup, except that if the duplication fails, it constructs a more useful error message, thereby centralizing error-handling code that would otherwise be duplicated throughout the header I/O routines:

```
char            *
hstrdup(s, name, band)
        char            *s;             /* string to duplicate   */
        char            *name;          /* header name           */
        int              band;          /* header band #         */
{
        char            *rtn;           /* duplicate string      */

        REQUIRE(s != NULL);

        rtn = strdup(s);

        if (rtn == NULL) {
                usrerr(band >= 0 ?
                       "'%s' header: can't dup '%s' (band %d)" :
                       "'%s' header: can't dup '%s'",
                       name, s, band);
                return (NULL);
        }

        return (rtn);
}
```

A general principle for deciding what actions should be encapsulated in a single function is that a function should be fully explicable by a simple sentence; i.e., one verb and one object. To the extent that a function deviates from this criterion, it should be considered a candidate for:

- consolidation with another function

source files will have to be weighed against the benefits of shared access to top-level static, which would eliminate the need for many of the global variables currently employed by IPW.

- splitting into two or more functions
- expansion into another level of abstraction

Functions should be kept small enough to be easily comprehended. A source file should be small enough that you would not worry about losing track of the order of the pages in a printed version.[57] Deep nesting of blocks, or many blocks containing block-scoped local variables, are also indications that a function may be too large.

Ideally, all communication between a function and the "outside world" should be via either the formal arguments, or the function's return value. However, it is also desirable to minimize the number of arguments in a function. As a practical matter, beyond about 5 arguments, it becomes much more difficult for the programmer to remember the order of the arguments, and what each argument does. Functional grouping of the arguments can mitigate this somewhat (e.g., input arguments come first, then output arguments), as can other regularities in argument order (e.g., all IPW I/O routines accept arguments in the order: file descriptor, buffer, count). Nonetheless, "if there are 12 arguments, then you forgot one of them." [BLI 1984].

IPW allows functions to communicate via global data structures, in lieu of arguments that would otherwise convey the following:

- information unused in the called function, but required by functions further down the calling hierarchy
- environmental or contextual information that, once set, is used by several functions but never modified.

The resulting simplification of function calling sequences has proven worthwhile; however, this use of globals requires some discipline on the part of the programmer, since there is nothing in C to prevent any function from modifying a global to which it has access. All functions that explicitly access or modify global variables MUST identify those variables in their header comments.

The common C technique of placing related objects in a single structure can also be used to help reduce the number of arguments passed to a function, since only a single argument (the structure's address) need be passed to gain access to all of the objects.

The meaning of an argument should be invariant; i.e., "op-code" arguments which alter the interpretation of other arguments should be avoided. The one permissible exception to this is an argument which signals whether additional arguments follow, in a function which accepts a variable number of arguments.

The type **void** should always be used for functions which return no value.

Functions should not require a special calling sequence to be initialized; instead, they should rely on either compile-time initialization of local **static**, or run-time initialization triggered by a local **static** flag.

A.1.12. Miscellany

If a function explicitly allocates storage, then it should either `free` the storage before it returns, or pass a pointer to the storage back to the caller, in which case the disposition of the storage becomes the caller's responsibility.

[57] There are currently well over 500 source files in IPW, about 10 of which are more than 4 pages long.

Functions should always explicitly allocate any storage they require for their own use — the practice of passing "work vectors" as arguments is strongly discouraged.

If a function is known to set the global variable `errno`, then `errno` should be cleared before, and examined after, every call to the function, unless the function provides some other way to detect whether an error has occurred (e.g., by a unique return value).

Source text lines should not contain more than 78 characters after tabs have been expanded into spaces.

In situations where the `++` and `--` operators would have the same effect in either prefix or postfix form, use the prefix form.

Example:

```
/* wrong */                       /* right */
for (i = 0; i < n; i++) {         for (i = 0; i < n; ++i) {
```

The expression following `return` or `sizeof` should be enclosed in parentheses.

A.2. PORTABILITY

Portability refers to the set of constraints imposed on the coding process so that the resulting code will run in a range of C environments (the combination of C compiler, C library, operating system, and underlying hardware), without significant environment-specific changes. Therefore, a basic principle of portability (after [Lapin 1987]) is: restrict the code to the subset of C defined by the intersection of the capabilities of all desired C environments.

IPW defines this "portable environment" incrementally. The basic environment for which IPW code is written is defined by the original C language specification [Kernighan 1978] and by Version 7 of the UNIX operating system [BTL 1983]. Where extensions to this environment have proved necessary, they have been drawn from the (draft) ANSI C standard [Harbison 1987] and the IEEE POSIX operating system standard [IEEE 1988]. Only extensions which can be easily simulated in the base environment have been used. For example, several C library functions are used that are not defined in the base environment, and portable source for these is provided with IPW for environments that lack them.[58] Thus, use of standard library functions is a significant step towards portability.

The IPW programmer should use `lint` (specifically, `ipwlint` or `ipwmake lint`) religiously; this will catch most nonportable constructs. Some versions of `lint` will object to constructs that the programmer knows are safe; e.g., casting a pointer returned by a memory allocation function. Such constructs should be preceded by the one-line comment `/* NOSTRICT */`, to indicate to future maintainers that the programmer understood the `lint` message generated by the subsequent line.[59]

Some circumstances require nonportable code, which should be isolated in separate files and clearly marked as nonportable (e.g., in the `NOTES` section of the header comment). In many cases, a nonportable construct can be made portable by

[58] Or, they are redefined in terms of some local equivalent; e.g., substituting `index` for `strchr`.

[59] The original `lint` documentation indicated that `/* NOSTRICT */` would disable type checking of the subsequent source line, but this feature was apparently never implemented.

providing a few nonportable alternatives, one of which is selected at compile time by conditional preprocessing.

A.2.1. Legal but nonportable

This section lists C language constructs which are well-defined but nonetheless nonportable, principally because they are relatively recent additions to C and thus not universally implemented, but also because they have proven likely to expose deficiencies in particular C environments.

Features added to the C language since its original definition should in general be avoided; these include:

- hexadecimal character constants
- bit fields
- `enum` and `signed` types
- `const` and `volatile` type qualifiers
- `#elif` preprocessor directive
- #, ##, and `defined` preprocessor operators
- concatenation of string literals

Structures should be used neither as actual arguments in function calls nor as function return values; use structure pointers instead. Structures should not be copied by assignment; instead, use the library function `memcpy` to copy between structure addresses.

One extension to the original C specification that IPW uses is the separation of structure name spaces; i.e., the same member name may appear in different structures. This is supported by all C environments we have encountered, and is essential for readability.

Although explicitly allowed in C, arrays with more than 2 dimensions are poorly supported in some C environments. Therefore, IPW programs requiring higher-dimensioned arrays should dynamically allocate them with the IPW function `allocnd`. Such arrays may also be passed as arguments to functions, which cannot be done with normal multidimensional arrays without the function's knowing in advance all but the leftmost dimension.

While `main` is technically an integer function, many C environments discard its return value. IPW `main`s should always terminate by calling `ipwexit`.

Casting integer types will not always result in the unused high-order bits being set to 0. To access the low-order bits of an integer, use explicit masking.

Example:

```
long            i, j;
...
/* wrong */
i = (short) j;

/* right */
i = j & mask(sizeof(short) * CHAR_BIT);
```

String constants should not be broken across multiple lines with an embedded `\`.

Do not assume any particular evaluation order for complex expressions, except that specified for the following operators: `,` `?:` `&&` `||`.

A.2.2. Gray areas

The C language allows numerous constructs and expressions whose implementation and results are undefined. Needless to say, these "gray areas" of the language definition should be avoided.

The notion of white space is extended in some environments to include form feeds, vertical tabs, etc. In IPW programs, white space should be restricted to spaces, horizontal tabs, and newlines.

Many C environments make strings literals read-only at execution time. To obtain a writable string with a compile-time initial value, use the construct:

```
static char     s[] = "initial value";
```

Negative integers are notorious for exposing weaknesses in C environments. Neither the bit-shift operators `>>` and `<<`, nor the division operator `/`, should be used with negative integral operands. Negative quantities should not be converted between integer and floating-point representations. In all cases, care should be taken to avoid integer overflow, since its consequences are undefined.

Operations on characters should be avoided — the only safe ones are `==`, `!=`, and the boolean functions defined in `<ctype.h>`. For printable characters, the graphic representations should be used instead of octal (e.g., `'A'` instead of `'\101'`), since nondecimal constants may be unexpectedly sign-extended.

Floating-point initializers should be limited to individual constants. Some C **compilers** cannot do floating-point arithmetic when folding constants; others use software-emulated floating-point which may differ from the run-time environment.

Example:

```
/* BAD */                               /* GOOD */
double magic_ratio = 1.0 / 3.0;         double magic_ratio;

                                        magic_ratio = 1.0 / 3.0;
```

Operators with side effects should not be applied to variables that appear on both sides of an `=`.

Example:

```
/* BAD */                               /* GOOD */
sumxy[i++] = x[i] + y[i];               sumxy[i] = x[i] + y[i];
                                        ++i;
```

Command-line argument processing in IPW programs should be handled by `ipwinit`. IPW programs should not otherwise access the `main` arguments `argc` and `argv`.

IPW programs should not directly call the C library memory allocation functions. Generic requests for memory should be handled by `ecalloc`, requests for multidimensional arrays should be handled by `allocnd`, and requests for specific IPW objects should be handled by the allocation functions for those objects; e.g., *xxx*`hmake`.

`NULL` should always be cast to the correct pointer type if it appears in a function argument list.

Many C environments maintain different formats for function and data addresses, so IPW programs should never copy addresses between function and data pointers.

Since function formal arguments may actually be stored in a wider type than declared, a function should never take the address of a formal argument.

The address of an aggregate object, other than an array, should alway be obtained explicitly: don't assume that `&structure` is equivalent to `structure`.

A.2.3. Common blunders

This section lists some constructs that are explicitly forbidden by all C definitions, yet are used frequently enough to merit an explicit warning.

Comments should not be nested. The previously-described preprocessor mechanism should be used to ‘‘comment out’’ a section of code.

The address of a local object should never be used in a context outside the extent of the block where the object is defined. This includes returning the address as the value of a function, or placing the address in a variable whose extent exceeds that of the defining block.

Functions being invoked via a pointer should use explicit pointer notation; e.g., `(*fp)()`, not `fp()`.

The only legal pointer type conversion is to and from an `voidp_t` (or its equivalent, `char *`). All such conversions must be explicitly cast, and no other pointer type conversions are allowed. A pointer should not be dereferenced unless it is of the same type as the object whose address it contains.

Arrays must have at least 1 element.

Don't assume that the underlying character set is ASCII. In particular, printable characters may not collate contiguously.

Don't use multicharacter constants.

Don't `realloc`-ate data structures containing pointers to dynamically-allocated storage.

Don't test floating-point variables for equality.

Don't rely on parentheses to force a particular evaluation order. If the order is numerically significant, then the expression must be broken up into separate statements, and extra variables used to hold the intermediate results.

Remember that the unary minus is an operator. The expression `-x` may be unexpectedly widened if the quantity `-x` is representable in a narrower type than `x` (e.g., `-32768`).

Clear bits by `&`-ing with the complement of the bits to be cleared, to assure correct treatment of high-order bits:

```
i &= ~bits;
```

A.2.4. Preprocessor

Preprocessor control statements should be entirely contained on 1 line. There should be no white space before or after the `#`. Preprocessor directives should not be redefined with `#define`. There should be no comments on preprocessor control lines, except for the end-of-line comment on each `#define`.

Function macro definitions should have no white space between the macro name and the `(`. Function macros should be limited to 4 or fewer arguments. Formal arguments should not appear inside string literals in the replacement text. Function macros will not be called recursively.

The file name in a `#include` directive should not be a pathname; the actual location of header files is handled by `ipwmake`. The file name should be enclosed either in `<>`s (UNIX header files) or `""` (IPW header files). Header files should not end in an unterminated comment or string literal.

`#define`-ing a previously defined macro may cause an error; when in doubt, precede a macro definition with an appropriate `#undef`. Macro definitions may or may not be stacked; don't assume that `#undef` will ‘‘pop’’ a previous definition.

IPW uses `#if` and `#ifdef` liberally to control the compilation of nonportable code. When using `#if`, the conditional expression is constrained as follows:

- any macros must be defined, and must expand to integer constants (undefined macros will not necessarily expand to 0)
- no environmental inquiries (e.g., `sizeof`)
- arithmetic is `long int`
- no casts

Large integer constants in `#if` expressions should be explicitly `long` (i.e., trailing `L`). Character constants should be avoided. Mathematical errors in the evaluation of `#if` expressions cause unpredictable results.

The files under `$IPW/h/conf/` contain examples of environment-specific compilation controls currently recognized by IPW.

A.2.5. Names

Global names must be unique in the first 6 characters. All other names, including macros and labels, must be unique in the first 8 characters. Case may be used as described previously, but names should be selected so that all names would be distinct in a monocase environment.

Names should have neither leading nor trailing underscores.

File names (source code and header files) should be no more than 12 characters total — UNIX guarantees 14-character file names, and RCS requires 2 for the `,v` suffix.

A.2.6. Types

C provides several integer types, whose sizes vary according to the host environment. All that is guaranteed by C is that

```
sizeof(short) <= sizeof(int) && sizeof(int) <= sizeof(long)
```

for any particular implementation. While this implies that `int`, `short`, and `long`, and the corresponding `unsigned` types, could all be represented by the same number of bits, in practice the following assumptions about the domains of the various integer types have proven safe:

type	min. value	max. value
`char`	0	127
`unsigned char`	0	255
`int`	-32767	32767
`short`	-32767	32767
`unsigned`	0	65535
`unsigned short`	0	65535
`long`	$-(2^{31}-1)$	$2^{31}-1$
`unsigned long`	0	$2^{32}-1$

To understand these assumptions, it should be remembered that `int`s may be as small as 16 bits, `char`s may be signed, and negative integers may be expressed in signed-magnitude as well as complement representations.

Several additional restrictions on the use of integer types are observed by IPW:

- `int` should be the default integer type (especially for function arguments and return values, which are automatically widened to `int` from ''narrower'' types.) If conserving memory is important, use `short`. If overflow is likely, use `long`.
- Do **not** use the `char` types as small integers, since their sign behavior is implementation-dependent.
- The derived type `bool_t` should be used for any integer variable that will only assume the values `TRUE` or `FALSE`.
- Unsigned types should not be used directly, since they may not be implemented in a particular environment. Instead, use the IPW derived types and access macros listed below. The access macros are used to extract the value of an unsigned expression; they are no-ops in environments which correctly implement the corresponding type.

type	access macro
`uchar_t`	`UCHAR()`
`ushort_t`	`USHORT()`
`ulong_t`	`ULONG()`

- `long` quantities should not be directly assigned to narrower quantities, since undetectable truncation may result. Instead, use the following IPW functions, each of which accepts a single `long` argument :

function	returns
ltof	double
ltoi	int
ltou	unsigned

These functions cause program termination with an appropriate error message if the long argument could not be exactly represented in the return type. In situations of possible precision loss where such functions are not available, explicit casts should be used.

Example:

```
int             i;
char            c;

c = (char) i;
```

All quantities whose representation could need to be changed if IPW were moved to a new environment should be typedefd. Two conspicuous examples of this are pixel_t and fpixel_t, the types used for integer and floating-point image pixels, respectively. On some machines it may be expedient to make pixel_t equivalent to unsigned char, trading precision for speed; a similar tradeoff may be made between float and double for fpixel_t. Often, such representation decisions can be made at compile time:

Example:

```
 /*
  * put file descriptors in most space-efficient type
  */
#if OPEN_MAX <= SHORT_MAX
typedef short   fd_t;
#else
typedef int     fd_t;
#endif
```

Standard C types should not be redefined; e.g., avoid gimmicks like

```
#define int     long
```

The new ANSI types, such as long double and signed char, should be avoided, at least until ANSI-conforming compilers become ubiquitous. Some essential ANSI constructs are already simulated by IPW; in particular, the generic pointer void * appears in IPW as the type voidp_t.

Casts should always be used when copying from one type to another. Casts are especially necessary in the following circumstances:

- before a constant used as a formal argument:

Example:

```
#include <math.h>
...
double          sqrt;

sqrt_2 = sqrt((double) 2);
```

- before a function call whose return value is ignored:

  Example:

  ```
  (void) printf("%d %g0, i, variance[i]);
  ```

A.2.7. Machine-dependent constants

IPW programs often need to reference hardware-dependent constants; e.g., the smallest or largest floating-point numbers, the machine epsilon, etc. Hard-coding such values into the source code, or even `#define`-ing them on a per-module basis, would seriously restrict portability.

The approach taken by IPW is to determine such machine-dependent constants **dynamically**, and then to store these as a single set of macros that are automatically included by `ipw.h`. The list of constants is generated by the IPW program `machine`, which uses algorithms from [Cody 1988] plus a few of our own. When IPW is installed on a new host machine, the installation procedure requires that the `machine` program be compiled and run before compiling any other code.

This automatic approach seems superior to that of [Fox 1978], in which all known variants of the machine-dependent parameters are hard-coded in specific subroutines, and the implementor must enable the appropriate values for the host system, or determine them independently if they are not already provided.

The output of `machine` is a direct replacement for either of the ANSI C header files `<float.h>` or `<limits.h>` (depending on the options specified when `machine` is run). The IPW implementor may arrange for the `machine`-generated versions of these files to be included by `ipw.h` in environments that do not otherwise provide them.

A.2.8. Environment

This sections documents miscellaneous restrictions imposed by various C environments.

Overly complex declarations should be avoided. A declaration should not contain more than 6 modifiers (e.g., `*`, `[]`, etc.).

Local variables are typically allocated on a stack, whose size may be severely limited in some environments. No more than about 4 KBytes of local storage should be allocated per block, and no local object should be larger than 1 KByte. This restriction does not apply to dynamically allocated space.

Any coding technique which relies on 2's-complement arithmetic, or on the presence or absence of integer sign extension, should be avoided. This includes using `>>` for division by powers of 2, and using `char`s as small signed integers.

The type of `sizeof` varies across environments. It should not be assumed to be `unsigned`, nor should it be used as a `case` constant in a `switch` statement.

Byte ordering within integers obviously varies between different hardware. `unions` or `char` pointers should therefore not be used to extract bytes from integers — use shifting and masking instead.

C environments use a variety of mechanisms for isolating the defining instance of a global variable from multiple global declarations. IPW code should be written so that all global variables are defined exactly once, and initialized where defined; all other references should contain the keyword `extern`. This has led to the idiom of defining each global variable in its own source file.

Aggregate objects may contain unused bits (''holes'') due to the alignment requirements of their components. Binary comparisons (e.g., `memcmp`) of aggregates should therefore be avoided, since the value of the unused bits is undefined.

Environment variables should always be accessed via the `getenv` library function, not via a third `envp` argument to `main`.

A.3. PERFORMANCE

The performance of a piece of code (usually taken to mean how fast it executes, but also often meaning how sparingly it uses machine resources) is of secondary concern to its maintainability, portability, and style:

> ''It's easier to get a working system efficient than it is to get an efficient system working''. [Minow 1984]

General advice on performance tuning is beyond the scope of this section; [Bentley 1982] contains an excellent summary of the standard methodology. However, there are some particular techniques that IPW exploits that will be mentioned in the following subsections.

Most UNIX environments provide execution profiling at either the function or the source statement level. These statistics are essential to focus the performance tuning effort on the portions of code that are most resource-consumptive. The adage that ''ten percent of the code accounts for ninety percent of the running time'' has certainly proven true for IPW. The `-P` option of `ipwmake` can be used to build an IPW program with profiling code included, and to link the program to profiled versions of the IPW libraries.

As the tuning-profiling process iterates, it is essential to keep notes on the relative improvements yielded by various techniques [Collyer 1987]; in IPW, these notes should be kept in the `README` file in the same directory as the source code being tuned. These notes serve the dual purpose of preventing future maintainers of the code from trying ''obvious'' tuning techniques that have already proven ineffective, and of establishing patterns of effective tuning techniques that may be added to the performance standards.

A.3.1. Memory

Efficient use of memory resources means minimizing both the total consumption of memory, and the accessing of ''expensive'' memory objects.

IPW attempts to minimize the memory consumed by program instructions, by limiting the number of functions per object module (usually 1:1). This avoids the loading of code that is never executed.

Image processing is by nature a memory-intensive activity. IPW programs should attempt to minimize this by maintaining only the minimum required image context for the particular operation being performed. For example, point operations require only a single pixel of context, while neighborhood operations no more than the neighborhood of the current pixel.

The hierarchy of functions called by an IPW program should be as shallow as possible. This minimizes the amount of stack space occupied by functions that are waiting for a called function to return. In other words, it's better to design a program as a sequence of function calls at the same level, rather than a few top-level calls that each proceed several levels down.

Temporary storage space should be local to the function. Ideally, such space should be `auto` objects declared within the function. These object are the least expensive to allocate and free; however, such arrays must be sized at compile time and are restricted by stack size limits in some environments. If dynamically-allocated space is therefore necessary, it must be explicitly `free`d before the function returns.

The IPW programmer should be aware that the expense (run-time overhead) of accessing objects varies considerably according to their type, storage class, and alignment. Especially expensive is converting data between types that have radically different internal representations; e.g., between integer and floating-point. Such conversions should be avoided, possibly by redesigning the algorithm to operate entirely in either the integer or the floating-point domain.

Another ‘‘hidden’’ type conversion occurs when types are ‘‘widened’’ in function call and return sequences. Generally, any integer actual argument or return value is converted to `int`, and any floating-point actual argument or return value is converted to `double`, so the use of other types for formal arguments or return values may cause unnecessary conversions.[60]

In most environments, `int` and `double` operations are the fastest integer and floating-point operations. However, many C compilers allow the option of disabling the automatic `float` to `double` conversion in expressions where all floating-point operands have type `float`. If this option is in effect, then `float` operations can be significantly faster than `double` operations. For this reason, IPW uses the defined type `fpixel_t` for the floating-point representation of image pixels. This type is `float` by default, but may be set to `double` in environments where automatic widening cannot be disabled.

Frequently-accessed members of a structure should be located near the beginning of the structure, since many hardware architectures support fast variations of indexed access when the offset is small (of course, the first member may always be accessed without any offset being computed.)

Finally, on some architectures, some benefit may be realized by declaring local scalar variables before local arrays, owing to vagaries in indexed access to stack-based objects [BLI 1984].[61]

---

[60] These conversions can be avoided in an ANSI C environment if an appropriate prototype is in scope when the function is called.

[61] This has not been observed in any current IPW environment.

A.3.2. Registers

Use of the `register` storage class for frequently-accessed variables is probably the single most obvious performance enhancement available to the C programmer, but registers should not be considered a panacea. The tendency to "put everything in a register" can actually be counterproductive: `register` declarations do not always result in faster code; they may be ignored altogether by a good optimizing compiler; and they may mislead a subsequent maintainer of the code into assuming that a great deal of thought went into their assignment. `register` declarations should therefore only be used in cases where they are known to be beneficial. In practice, this means that *register* declarations should be applied only after the code is complete enough that it can be subject to function- or (preferably) statement-level execution profiling.

`register` declarations, assuming the compiler pays attention to them, are demonstrably most effective when applied to pointers (especially pointers to structures) or to integers used as array indices. They are less effective for integers used only in arithmetic calculations.

The number of registers per function available for general use can vary from 2 (PCs) to essentially unlimited (some RISC architectures). On some machines (e.g. VAX), all data types are taken from the same register pool; other machines (e.g. MC680x0) distinguish between data and address registers. Machines with floating point hardware may provide separately assignable floating-point registers. Excess `register` declarations will in all cases be ignored, but the order in which the available registers will be assigned is undefined.

To allow the programmer some control over how registers are assigned, IPW source code contains no unadorned `register` declarations. Instead, the pseudo storage classes `REG_`*n* (for integer and pointer registers) and `FREG_`*n* (for floating-point registers) are provided, where *n* is a number between 1 and (currently) 6.[62] The declarations should be assigned in decreasing order of importance. When IPW is installed, declarations corresponding to the available number of registers are set to `register`, and the remaining declarations are elided.

Example:

```
char *
memcpy(dest, src, nbytes)
        REG_1 char     *dest;           /* -> destination array  */
        REG_2 char     *src;            /* -> source array       */
        REG_3 int       nbytes;         /* # bytes to copy       */
{
        while (--nbytes >= 0) {
                *dest++ = *src++;
        }
}
```

It should be noted that there are certain situations where a `register` declaration adds a fixed amount of execution overhead. Architectures that save and restore registers as part of the function calling sequence will add overhead to the function call for

[62] We assume that environments providing more than 6 assignable registers probably also have compilers smart enough to optimally allocate registers without programmer assistance.

each additional register allocated. Using `register` declarations on formal arguments may also cause an explicit copying of the formal argument into a register, even if it is subsequently unused.

A.3.3. Control structures

The overall flow of control in a function is of course dictated by the particular algorithm it implements. Within this constraint, there are a few simple techniques employed in IPW, involving the selection of the most generally efficient of otherwise equivalent structures.

Loops are the most obvious target for control structure optimization. "Endless" loops (i.e. exited only via `break` or `exit()`) should be written as `for (;;)`— on some IPW hosts the equivalent `while (TRUE)` has been observed to compile into an explicit test of the constant condition.[63] Loops within which the value of the control variable is not consulted should be written in the "count down to 0" form.

Example:

```
/* fast */                      /* slow */
for (i = n; --i >= 0; ) {       for (i = 0; i < n; ++i) {
        ...                             ...
}                               }

        /* usually fastest, if testing against 0 */
        i = n;
        do {
                ...
        } while (--i > 0);
```

The `do-while` form, with the initialization of the loop counter immediately preceding the loop, is an IPW idiom.

The efficiency of a `switch` statement is notoriously unpredictable: on some architectures, it compiles into an efficient jump table, while on others it is functionally equivalent to an `if-else` sequence. Moreover, a `switch` statement allows no control over the order in which the various `case`s are evaluated, whereas an explicit `if-else` sequence can be arranged to test the most frequent cases first. In IPW `if-else` is therefore preferred to the `switch`, unless the `switch` results in significantly more readable code.

A similar caveat applies to the conditional operator `?:`. It often results in significantly worse code than the equivalent `if-else`, and is also usually less readable.

If repeated calls to a particular function are accumulating a significant fraction of the program's runtime, then it may be worthwhile to construct a macro "wrapper" for the function, which tests for common situations in which the function call can be avoided. For example, if strings are being compared, and it is likely that a significant number of them will differ in the first character, then calls to the string-compare function can be avoided by explicitly testing the first characters of the strings [Collyer 1987]:

---

63 Most compilers ARE smarter than this ...

```
#define STREQ(s1,s2) (*(s1) == *(s2) && strcmp((s1)+1,(s2)+1) == 0)
```

References

Adams 1979.
Adams, J. and E. Driscoll, "A low-cost transportable image processing system," in *First ASSP Workshop on Two-Dimensional Signal Processing*, Berkeley, CA, October 3-4, 1979.

Adobe 1985.
Adobe Systems Inc., *PostScript Language Reference Manual,* Addison-Wesley, Reading, MA, 1985.

Aho 1988.
Aho, A., B. Kernighan, and P. Weinberger, *The AWK Programming Language,* Addison-Wesley, Reading, MA, 1988.

Allman 1989a.
Allman, E., "Breaking the code," *UNIX Review*, vol. 7, no. 11, pp. 79-86, 1989.

ANSI 1989.
X3J11 Technical Committee on the C Programming Language, "American National Standard for Information Systems - Programming Language C," X3.159-1989, X3 Secretariat, Computer and Business Equipment Manufacturers Association, New York, 1989.

Bentley 1982.
Bentley, J., *Writing Efficient Programs,* Prentice-Hall Inc., Englewood Cliffs, NJ, 1982.

BLI 1984.
Britton Lee Inc., "Host software coding standards, revision 2.10," 205-1187-003, Britton Lee Inc., 12 November 1984.

Bracken 1983.
Bracken, P., "Remote sensing software systems," in *Manual of Remote Sensing, Second Edition*, ed. R. Colwell, vol. 1, pp. 807-839, American Society of Photogrammetry, Falls Church, VA, 1983.

BTL 1983.
Bell Telephone Laboratories, Inc., *UNIX Time-Sharing System : UNIX Programmer's Manual, Volume 1,* Holt, Rinehart and Winston, New York, 1983.

Castleman 1979.
Castleman, K., *Digital Image Processing*, Prentice-Hall, Englewood Cliffs, NJ, 1979.

Chase 1986.
Chase, R., "Data and Information System Data Panel Report," Technical Memorandum 87777, NASA, 1986.

Chen 1987.
Chen, T., D. Staelin, and R. Arps, "Information content analysis of Landsat image data for compression," *IEEE Transactions on Geoscience and Remote Sensing*, vol. GE-25, no. 4, pp. 499-501, 1987.

Cody 1988.
Cody, W., "Algorithm 665, Machar: A subroutine to dynamically determine machine parameters," *ACM Transactions on Mathematical Software*, vol. 14, pp. 303-309, 1988.

Collyer 1987.
Collyer, G. and H. Spencer, "News need not be slow," in *Winter 1987 USENIX Technical Conference Proceedings*, pp. 181-190, USENIX Association, Washington, DC, 1987.

Dozier 1984.
Dozier, J., "Snow reflectance from Landsat-4 Thematic Mapper," *IEEE Transactions on Geoscience and Remote Sensing*, vol. GE-22, no. 3, pp. 323-328, 1984.

Dozier 1989.
Dozier, J. and J. Frew, "Rapid calculation of terrain parameters for radiation modeling from digital elevation data," *Proceedings IGARSS '89*, 1989.

Dromey 1982.
Dromey, R., *How to Solve it by Computer*, Prentice-Hall Inc., Englewood Cliffs, NJ, 1982.

Dubayah 1990.
Dubayah, R., J. Dozier, and F. Davis, "Topographic distribution of clear-sky radiation over the Konza Prairie, Kansas, USA," *Water Resources Research*, 1990 (in press).

Duda 1973.
Duda, R. and P. Hart, *Pattern Classification and Scene Analysis*, Wiley, New York, 1973.

Elder 1989.
Elder, K., J. Dozier, and J. Michaelsen, "Spatial and temporal variation of net snow accumulation in a small alpine watershed, Emerald Lake basin, Sierra Nevada, California, USA," *Annals of Glaciology*, vol. 13, pp. 56-63, 1989.

EOSAT 1985.
EOSAT, *User's Guide for Thematic Mapper Computer-Compatible Tapes*, Earth Observation Satellite Company, 1985.

Esaias 1986.
Esaias, W., *Moderate-Resolution Imaging Spectrometer Instrument Panel Report*, Earth Observing System Reports, IIb, NASA, 1986.

Feldman 1979.
Feldman, S., "Make : a program for maintaining computer programs," *Software--Practice and Experience*, vol. 9, pp. 255-265, 1979.

Fenlason 1990.
Fenlason, J., "tar : The GNU Tape Archive," online documentation, 28 January 1990.

Fox 1978.
Fox, P., A. Hall, and N. Schryer, "Framework for a portable library," *ACM Transactions on Mathematical Software*, vol. 4, pp. 177-188, 1978.

Frew 1984.
Frew, J. and J. Dozier, "The QDIPS image processing system," (CSL Technical Report), Computer Systems Lab, University of California, Santa Barbara, 1984.

Goetz 1986.
Goetz, A., *High-Resolution Imaging Spectrometer Instrument Panel Report*, Earth Observing System Reports, IIc, NASA, 1986.

Green 1976.
Green, P., *Mathematical Tools for Applied Multivariate Analysis*, Academic Press, New York, 1976.

Harbison 1987.
Harbison, S. and G. Steele Jr., *C: A Reference Manual (Second Edition)*, Prentice-Hall Inc., Englewood Cliffs, NJ, 1987.

Hemenway 1984.
Hemenway, K. and H. Armitage, ''Proposed syntax standard for UNIX system commands,'' *UNIX/world*, vol. 1, no. 3, pp. 54-57, 1984.

Hunt 1982.
Stevens, W. and B. Hunt, ''Software pipelines in image processing,'' *Computer Graphics and Image Processing*, vol. 20, pp. 90-95, 1982.

IEEE 1988.
Technical Committee on Operating Systems, IEEE Computer Society, ''IEEE Standard Portable Operating System Interface for Computer Environments,'' Std 1003.1-1988, The Institute of Electrical and Electronic Engineers, Inc., New York, 1988.

Johnson 1978.
Johnson, S., ''Lint, a C program checker,'' in *UNIX Time-Sharing System : UNIX Programmer's Manual, Volume 2*, pp. 278-290, Holt, Rinehart and Winston, New York, 1983.

Kernighan 1976.
Kernighan, B. and P. Plauger, *Software Tools*, Addison-Wesley, Reading, MA, 1976.

Kernighan 1978.
Kernighan, B. and D. Ritchie, *The C Programming Language*, Prentice-Hall Inc., Englewood Cliffs, NJ, 1978.

Landy 1984.
Landy, M., Y. Cohen, and G. Sperling, ''HIPS : a Unix-based image processing system,'' *Computer Vision, Graphics, and Image Processing*, vol. 25, pp. 331-347, 1984.

Lapin 1987.
Lapin, J., *Portable C and UNIX System Programming*, Prentice-Hall Inc., Englewood Cliffs, NJ, 1987.

LaVoie 1987.
LaVoie, S., C. Avis, H. Mortensen, C. Stanley, and L. Wainio, ''VICAR User's Guide, Version 1,'' D-4186, Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA, 1987.

Li 1987.
Li, S., Z. Wan, and J. Dozier, ''A component decomposition model for evaluating atmospheric effects in remote sensing,'' *Journal of Electromagnetic Waves and Applications*, vol. 1, no. 4, pp. 323-347, 1987.

Martin 1988.
Martin, T., M. Martin, R. Davis, R. Mehlman, M. Braun, and M. Johnson, ''Planetary Data System Standards for the Preparation and Interchange of Data Sets, Version 1.1,'' JPL D-4683, Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA, 03 October 1988.

Meyer 1988.
Meyer, B., *Object-Oriented Software Construction,* Prentice-Hall, New York, 1988.

Minow 1984.
Minow, M., ''A C style sheet,'' *Toolkit - The UNISIG Newsletter*, vol. 3, no. 1, pp. 3-34, DECUS, April 1984.

Musciano 1988.
Musciano, C., ''Tooltool User's Guide, Version 2.1,'' online documentation, Advanced Technology Dept., Harris Corp., Melbourne, FL, 1988.

Paddon 1988.
Paddon, J., ''Image Processing Software for Education in Remote Sensing,'' M.A. thesis, Dept. of Geography, University of California, Santa Barbara, CA, September 1988.

Paeth 1986a.
Paeth, A., ''The IM Raster Toolkit : design, implementation, and use,'' Tech. Rept. CS-86-65, University of Waterloo Computer Science Department, December 1986.

Paeth 1986b.
Paeth, A., ''A fast algorithm for general raster rotation,'' in *Proceedings, Graphics Interface '86,* pp. 77-81, Vancouver, BC, 1986.

Perkins 1988.
Perkins, D., D. Howell, and M. Szczur, ''The Transportable Applications Environment - an interactive design-to-production development system,'' in *Digital Image Processing in Remote Sensing*, ed. J. Muller, pp. 39-64, Taylor and Francis, London, 1988.

Plum 1984.
Plum, T., *C Programming Guidelines,* Prentice-Hall Inc., Englewood Cliffs, NJ, 1984.

Porter 1987.
Porter, W. and H. Enmark, ''A system overview of the Airborne Visible/Infrared Imaging Spectrometer (AVIRIS),'' *Proc. SPIE (Imaging Spectroscopy II)*, vol. 834, pp. 22-31, 1987.

Pratt 1978.
Pratt, W., *Digital Image Processing,* Wiley, New York, 1978.

Press 1988.
Press, W., B. Flannery, S. Teukolsky, and W. Vetterling, *Numerical Recipes in C,* Cambridge University Press, Cambridge, 1988.

Rasure 1987.
Rasure, J., D. Argiro, E. Engquist, S. Hallette, R. Neher, S. Wilson, and M. Young, ''Interactive image processing using X windows,'' *ESD*, pp. 32-33, December 1987.

Rew 1989.
Rew, R., *netCDF User's Guide, Version 1.0,* Unidata Program Center, University Corporation for Atmospheric Research, Boulder, CO, April 1989.

Rice 1983.
Rice, R. and J. Lee, ''Some practical universal noiseless coding techniques, part II,'' JPL Publication 83-17, Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA, 1983.

Richards 1986.
Richards, J., *Remote Sensing Digital Image Analysis,* Springer-Verlag, New York, 1986.

Ritchie 1974.
Ritchie, D. and K. Thompson, ‘‘The UNIX time-sharing system,’’ *CACM*, vol. 17, pp. 365-375, 7, July 1974.

Scheifler 1986.
Scheifler, R. and J. Gettys, ‘‘The X Window System,’’ *ACM Transactions on Graphics,* vol. 5, no. 2, pp. 79-109, April 1986.

Sellers 1965.
Sellers, W., *Physical Climatology,* University of Chicago Press, Chicago, 1965.

Simonett 1978.
Simonett, D., T. Smith, W. Tobler, D. Marks, J. Frew, and J. Dozier, ‘‘Geobase Information System Impacts on Space Image Formats,’’ SBRSU Technical Report 3, Remote Sensing Unit, Department of Geography, University of California, Santa Barbara, CA, April 1978.

Simonett 1983.
Simonett, D., ‘‘The development and principles of remote sensing,’’ in *Manual of Remote Sensing, Second Edition,* ed. R. Colwell, vol. 1, pp. 1-35, American Society of Photogrammetry, Falls Church, VA, 1983.

SPOT 1989.
SPOT Image Corporation, *SPOT User's Handbook,* SPOT Image Corporation, Reston, VA, 1989.

Stonebraker 1986.
Stonebraker, M., ‘‘Retrospection on a database system,’’ in *The INGRES Papers : Anatomy of a Relational Database System*, ed. M. Stonebraker, pp. 46-62, Addison-Wesley, Reading, MA, 1986.

Stroustrup 1986.
Stroustrup, B., *The C++ Programming Language,* Addison-Wesley, Reading, MA, 1986.

Sun 1987.
Sun Microsystems, Inc., ‘‘External Data Representation Standard: Protocol Specification,’’ RFC 1014, Internet Activities Board, June 1987.

Sun 1988.
Sun Microsystems Inc., ‘‘Pixrect Reference Manual,’’ part number 800-1785-10 rev A, Sun Microsystems Inc., Mountain View, CA, May 1988.

Thompson 1978.
Thompson, K., ‘‘UNIX Time-Sharing System: UNIX implementation,’’ *Bell Sys. Tech. J.*, vol. 57, no. 6, pp. 1931-1946, 1978.

Vane 1988.
Vane, G. and A. Goetz, ‘‘Terrestrial imaging spectroscopy,’’ *Remote Sensing of Environment*, vol. 24, no. 1, pp. 1-29, February 1988.

Welch 1984.
Welch, T., ‘‘A technique for high-performance data compression,’’ *IEEE Computer*, vol. 17, no. 6, pp. 8-19, 1984.

Wharton 1987.

Wharton, S. and Y. Lu, ‘‘The Land Analysis System (LAS): a general-purpose system for multispectral image processing,’’ in *Proceedings of IGARSS ’87 Symposium*, pp. 1081-1086, Ann Arbor, MI, 18-21 May 1987.