LOW LEVEL VISION SYSTEM

James H. Burrill

COINS Technical Report 87-14

January 1987

Abstract

This report is a presentation of the general design and design philosophy for the new Visions system. It is the result of reading of the documentation of the current system and numerous discussion with people in the VISIONS System (LLVS) because it is designed to facilitate operations on image data only and to exist as a sub-part of larger systems constructed to handle all three levels of the vision processing problem.

Low Level Vision System

James H. Burrill

March 6, 1987

Abstract

This report is a presentation of the general design and design philosophy for the new Visions system. It is the result of reading of the documentation of the current system and numerous discussions with people in the VISIONS research group. I refer to this new system as the Low Level Visions System (LLVS) because it is designed to facilitate operations on image data only and to exist as a sub-part of larger systems constructed to handle all three levels of the vision processing problem.

Contents

List of Figures

Preface

This system should be viewed as a tool for the construction of further systems much as a compiler is viewed as a tool for constructing applications. Therefore, there are three parties involved that must be kept separate when reading this document. There is the "user" — that anonymous person who uses not this system but the *end* application built with it. There is the writer of image operators — a *user* of this system in their own right who constructs the *end* application system. Finally, there is this system. References to the "user" in this document refer to the writer of the image operators.

Any proposal for a new system should start with the justification of why the time and expense should be spent to do it. The justification is usually in the form of requirements that old facilities do not satisfy. If these requirements are not sufficient reason then we can be content that we need not spend additional resources on a new system.

This document refers to an "old Vision System". This system is known as the *VISIONS Image Operating System* and was designed by Ralf Kohler as part of his PhD dissertation[33]. It is this system from which the requirements were generated. The reader familiar with that system may assume that details left unspecified will in all probability work in the same way.

As with any documentation of a new system this document should be viewed as being incomplete, incorrect, and subject to change.

Many people contributed greatly to the ideas expressed by this system; sometimes suggesting important ideas that were subsequently incorporated, or, strenously objecting to proposals and implemented features. The differences from the old system resulted mainly from the ideas of Les Kitchen. The brakes were supplied mostly by Joey Griffith and Charlie Kohl.

Many people worked on the implementation. Robert Heller was responsible for implementing the *per-pixel* logic and the display operators. Nick Afshartous, Jonathan Malin, and Dave Franklin contributed most of the standard image operators. Ken Whitehouse and I contributed the remaining Lisp code.

[33]Kohler, R.R., "Integrating Non-semantic Knowledge into Image Segmentation Processes", COINS Technical Report 84-04, University of Massachusetts, Amherst, Ma 01003, 1981

1 Requirements

The following are the reasons why a new system has been constructed even though the old system was still very productive.

- The system should be coded in CommonLisp and C.

  In order to interface more easily with other groups doing VISIONS research and to port the system to as many other places as possible, it is necessary to use standard languages -- languages that can be ported easily. A form of Lisp is necessary to allow easy construction of the higher level programs. At the lower level, a language that allows very efficient code to be constructed is necessary because of the huge amounts of computation that low level vision requires. CommonLisp satisfies the Lisp requirement as it is fast becoming a standard and is being used by other groups working on shared projects. The lower level language requirement is satisfied by C for the same reasons.

- The system should be at least as efficient as the old system.

  The old system is very inefficient in doing standard operations on pixel data. It is, however, very flexible in the operations that it allows users to do. The inefficiency results from the numerous procedure calls that are made to process each pixel of data (minimum of three). A good design should preserve the power of the old while allowing standard operations to be done faster.

- The system structure should follow well understood structured design principles.

  The old system has numerous problems that allow novice users to easily introduce bugs in their systems. These problems are primarily the result of the way parameters are passed and stored in the system. There is no association of parameters with the specific objects that they describe. Operators are allowed to pass parameters between themselves. This bypasses the higher level control logic and, thereby, increases the cohesion[1] of the system as a whole. It also introduces order dependency that is not readily apparent to the reader nor always necessary in the operation of the system.

- The new system should be easier to learn and use.

  The old system made an effort in this direction. Its structure allowed image operators to be constructed more easily because they did not have

[1] The more cohesion there is between two modules, the harder it is to modify one module without affecting the other.

to include looping and indexing control as well as other standard sub-operations. The implementation of the *help* feature was another important effort. The new system should improve on both of these aspects if possible. In addition, the interface to the *end* user for obtaining the values of parameters should be put at the top level so that they can respond using facilities that exist at that level (Lisp instead of FORTRAN I/O).

- Various new features that are accepted, useful extensions should be added.

  Who can design a successor product without wanting to add something of their own? But, it is important to realize that anything of this nature should not be added unless: it is clearly accepted as a needed feature that will improve the flexibility or efficiency of the system, or its negative impact on the cost of running, learning, and maintaining the new system will be minimal.

- General requirements

  The following things are listed so that the reader knows they have not been overlooked.

  - Ergonometric[2] factors should be considered in all design choices.
  - The system should be able to handle all exceptions in a user friendly way.
  - The system should be independent of particular graphics devices.
  - The internal structures used should be readily understandable by the user of the system.

2 Non-requirements

Things the design of the new system will not worry about.

- The *end* user interface of the old system need not preserved.

  The general opinion is that the user interface to the image operators need not be preserved as to do so may interfere with some of the requirements.

[2] i.e. a measure of the impact of technology on biological systems.

3 Representation of Image Data

3.1 Planes

The basic "object" supported by the LLVS is the *plane*. A *plane* is a rectangular array of pixel data. The *plane* has a *type*[3], a *size* (the row and column maximum for the array), a *level* (some integer), a *location* (two integers), a *background value*, and an *associations* list.

A *plane* is defined as a structure using the *defstruct* in Figure 1. The *type* of *plane* is the type of pixel data stored in the *pixel* slot of the *plane*. The following types of pixels are defined:

1. :BIT — 0 or 1
2. :BYTE — 0 to 255
3. :SHORT — a 16 bit signed integer
4. :INT — a 32 bit signed integer
5. :FLOAT — a 32 bit floating point number

The size of the *plane* is the dimensions of the array. The array always has two and only two dimensions. Access to the slots of the structures are allowed using the access functions created by the standard CommonLisp `defstruct` facility. However, access to the actual pixel data should only be done via functions provided by LLVS as actual pixel representations will differ on different machines.

The *level* of a *plane* is an integer that is similar to the *level* of the old system. It is a power of two describing the resolution of the image. This integer may be any value from 0 to *n*. It is used when *planes* at different resolutions are processed together by one operator. There is a primary *level* used in any operation. This value is obtained from the user or from one of the *planes* involved. The relation of the primary *level* to the *level* of any *plane* is used to determine the stride used in that *plane* during an operation [4].

The *background value* is the value supplied by an *access function* for references outside the extent of a *plane* when the `:background-value` access method is used[5].

The *associations* list is provided as a way in which certain parameters that are associated with the *plane* such as `:MINVAL` and `:MAXVAL` may be stored and is represented as a CommonLisp association list.

[3]The type is basically the form of the pixel data. This data is in Lisp arrays on some machines and in special structures on others. While it is possible to construct *planes* containing any valid Lisp object, only *planes* restricted to a set of special numeric pixel formats may be used with most of the image operators.

[4]If the primary *level* is 6 and the *level* of a *plane* in an operation is 7, then the stride used will be 2 — every other pixel in the row direction and every other pixel in the column direction will be accessed from that plane.

[5]The *out-of-bounds* action is determined by the image operator which *may* obtain it from the *end* user.

```
(defstruct
  plane "a rectangular array of pixel data"
   (level 0 :type fixnum)
   (pixels nil :type pixel-data)
   (row-location 0 :type fixnum)
   (column-location 0 :type fixnum)
   (background-value 0)
   (associations nil :type list))
```

Figure 1: Structure of a Plane

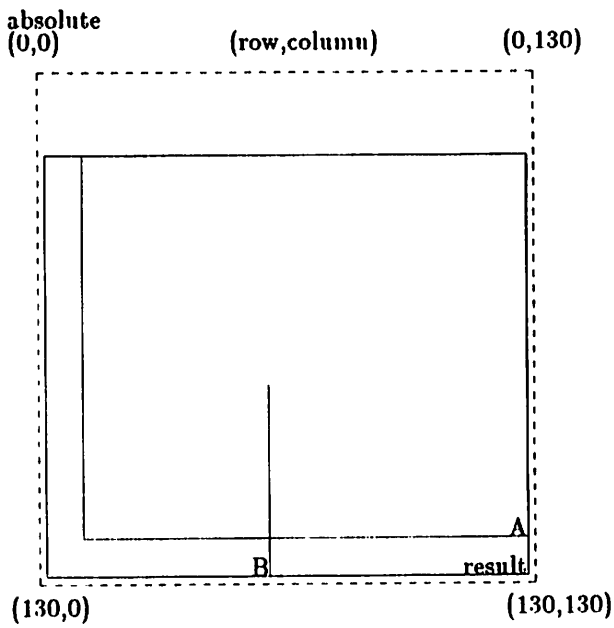

Figure 2: Non-congruent planes example.

LLVS uses the concept of a *conceptual plane* to control the processing of image operators. The *conceptual plane* is the largest extent at the primary *level* of an operation that encompasses all the *planes* used in the operation. This definition is necessary because each *plane* in an operation may be at a different *level*, be a different *size*, or be at a different *location*. The extent of the *conceptual plane* of an operation can be modified by the *end* user by specifying *limits* to be used in the operation.

The *location* of a *plane* is used to line up two *planes* for processing when involved in the same operation[6]. As an example, consider two *planes* **A** and **B** of the same *level*[7] as shown in Figure 2. Plane **A** is a 100 by 120 image and plane **B** is a 50 by 60 image. Plane **A**'s *location* is (20,10) and plane **B**'s *location* is (80,0). The operation is to add the values of the two *planes*[8] to produce a third *plane*. The third *plane*'s *location* will be (20,0) and its *size* will be (110,130). The operation will be done over an area of 110 by 130. Therefore, the *conceptual plane* is at location (20,0) and is *size* (110,130).

The old Vision System does not have the concept of *location*. It considers every *plane* to be lined up in the upper left hand corner. The reason for introducing the concept of *location* is so that pieces of *planes* may be extracted for more intensive processing[9]. The *location* allows various extracted *planes* to be related in further operations. This concept can be completely ignored by the user as every *plane* will have a *location* of (0,0) unless the user changes it or forms a new *plane* by extracting it from another one.

The coordinate system used is *row* - *column* where columns increase from left to right and rows increase from top to bottom. For any pixel in the plane whose coordinate is (row, col) the smallest value for (row, col) is in the upper left-hand corner. The reasons for choosing a row - column coordinate system over some other system are:

- The old system uses row - column.
- Raster scan displays and inputs use row - column.
- Row - column can be mapped readily into the monotonically increasing, contiguous memory of a digital computer.
- A simple transformation results in cartisian coordinates.

[6] All *plane locations* are relative to the absolute origin (0,0).

[7] When *planes* at different *levels* are used in an operation, the *locations* are converted to correspond to a location in a *conceptual plane* at the primary *level* of the operation. Thus, if the primary level is 9, the plane's level is 8 and its location is (10,15), the plane's conceptual location for the operation is (20,30). If the primary level were 7 then the plane's conceptual location would be (5,7.5) in the primary level.

[8] For values missing from either plane, the user may specify that the *background value*, the value obtained from the nearest neighbor, or some other method be used.

[9] By extracting the piece first a lower paging overhead may be obtained when processing it.

$$\begin{array}{lcl} row & \leftrightarrow & x \\ column & \leftrightarrow & y \end{array}$$ ... a rotation 90° counter-clockwise on displays.

3.2 Mask Planes

As with the old Vision System, LLVS processes planes under the control of a mask. A mask is a *plane* at some *level*. The mask plane is indexed as any other plane in an operation[10]. Each pixel value is compared to a CommonLisp *sequence* of values[11]. If the pixel from the mask plane is equal to a member of the vector, the P section of the image operator is called for that position. Otherwise, no value is stored in that position in all output *planes* of the operation. The *mask-plane* and *mask-value* are supplied as arguments to the per-pixel *drivers*.

It is up to the image operator to obtain the *mask-plane* and *mask-value*. The operator can use the standard parameter access facilities for this. While the *per-pixel drivers* will automatically utilize masks, it is the responsibility of the writer of the operator to pass the masks to the *driver*. And, if the operator is written using the *per-plane* method, it must be written explicitly to handle masks.

3.3 Cones

A *cone* is a set of planes organized as a Lisp association list. *Cones* can be used to manipulate planes as a set. Some primitive operators to save to or restore from disk are provided. Other operators will apply an image operator to all the *planes* in a *cone* generating a new *cone*. An association list structure is used so that a "name" may be associated with a *plane* in a standard way. This also allows all CommonLisp functions that deal with association lists to be used with cones.

3.4 External Storage of Planes and Cones

LLVS provides access to the external files of image data in use by the old Vision System. However, since it is not be feasible for *planes* generated by the new system to be placed into these files, an equivalent set of I/O functions in Lisp are provided to get and save these new *planes* into new external structures.

---

[10]This is includes the way missing values are treated.

[11]This comparison is done using integer comparison on integer values. Any floating point values in the *mask-plane* or *mask-value* are first truncated to integers.

4 Writing Image Operators

Image operators may be written in several ways depending on the efficiency required and the degree of effort the writer wishes to provide. To the *end* user each image operator looks like a normal Lisp function that operates on special data called *planes*. LLVS provides many Lisp functions to help in the effort of producing these operators and to make the user interface as uniform as possible. The Examples section of this reference annotates some examples of image operators written in the different ways.

4.1 Types of Image Operators

LLVS provides two ways of writing image operators. The first way (*per-pixel* method) is very similar to the way used in the old Vision System - the user writes their operation in terms of the effect on one pixel.

The second way (*per-plane* method) requires operations to be written in terms of a complete *plane*. Consequently, the *per-plane* operators are much more complex to write[12]. But, because these operators do their own looping through the *plane*, they require no procedure calls per pixel of the result. Standard macros, templates, and examples are provided to make writing these operators as easy as possible.

Whether an image operator is a *per-pixel* or *per-plane* operator makes no difference in the way it is called at the top level.

At the top level the image operator is written in Lisp. This allows the full power of CommonLisp to be used to write and debug this part. As much as possible has been moved to this level[13]. This code includes the logic for obtaining the parameters needed, allocating result *planes*, calling the proper *driver*, and processing any result parameters. Since the developer of the image operator is required to write this Lisp function, they can write it in any way they choose.

4.2 The *Per-pixel* Method

The *per-pixel driver* is called with all the planes required in the operation and supplied with the pointer to the function that processes each pixel. The driver calls that function for every pixel in the *conceptual plane* of the operation. An image operator using the *per-pixel* method will actually be composed of at least one, to as many as needed, C functions. Some of the functions may be used for any initialization and termination logic needed for the function called for each pixel[14]. The *per-pixel driver* may be called using different functions. This allows

[12]It is intended that these operations be written by the systems personnel and include the standard image operations.

[13]For this reason everything that is included in the E section in the old Vision System is in this top level.

[14]These functions correspond to the I, P, and T sections of the old VISION System.

the image operator to make as many passes as needed over the data. Functions are typically categorized as follows:

- I function

  This function is called directly by the image operator Lisp code with any value needed in the P function of the operator as its arguments. It is the job of the I function to place these values in the appropriate `static` variables and do any other initialization required. As the writer of the image operator writes both the I function and the top level Lisp function, the writer is free to pass any values to the I function in whatever form desired.

- P function

  This function is called once per pixel of the *conceptual plane* of the operation and has no arguments. It is called by the *per-pixel driver*.

- T function

  This function is called after all the calls to the P function. The T function must do any cleanup required and pass back any resultant values required. It passes these values back by using the arguments it was called with as addresses of the variables to be changed. As the writer of the image operator writes both the T function and the lop level Lisp function, the writer is free to pass any values back from the T function in whatever form desired.

Since the writer of the image operator writes the calls to the I and T functions, the writer need not include calls to them if they are not needed. The writer can also have as many P, I, or T functions as needed.

When the *per-pixel* method is used to write an image operator, the P function must obtain the pixels that it will process and set the pixels that are the result. *Access functions* are used to obtain and set pixels in a *plane* and convert from the data format of the *plane* to the format desired by the P function function. Most access functions return or set one pixel but some access functions return a *window* of pixels.

4.3 The *Per-plane* Method

The writer of a *per-plane* image operator can write the operator in any way they prefer. The *per-plane* method is another way of saying do it yourself. LLVS does provide some help in the form of Lisp functions for setting up structures for passing to C, C macros for common coding sequences, existing image operators that can be modified to suit individual needs, and the best wishes of the designer of this system.

In the *per-plane* method the writer of the image operator must do all of the indexing over the *planes*. This is not as easy as it may seem because the actual

planes may not conform the *level*, *size*, and *location* of the *conceptual plane* of the operation which can be specified by the caller of the image operator. The writer ends up duplicating a *per-pixel* driver.

The more that the writer can constrain the generality of their operator, the easier it will be to write and the faster it will execute. The following things may be considered:

- :BOUNDS-ACTION — if one can be picked it eliminates a `switch` statement when an *out-of-bounds* reference occurs.
- *level* and *location* — If all the *planes* are at the same *level* and *location* then conversion of row and column indexes for each *plane* will not be needed.
- *plane type* — if each *plane* can be constrained to a particular type, its type need not be checked at each pixel.

The Lisp Forms section of this manual documents some Lisp functions that will be helpful in passing the standard things to the C routine. Some C macros and functions are discussed in the C Support section.

4.4 Image Operator Parameters

In the LLVS, parameters controlling how an operation is to be done are always associated with *something*.

- Parameters may be associated with a *plane* in which case they will be on the *plane's* association list. These parameters should only describe aspects of that particular *plane*.
- Parameters may be associated with a particular operation. These parameters are given as arguments to the image operator.
- Parameters may be associated with the *end* user. These parameters generally control overall aspects of the processing such as the type of error checking to be done.
- Standard parameters often have a default defined.

To standardize the way these parameters are obtained, several functions and facilities are supplied as part of LLVS. When an image operator requests a parameter using these facilities, they use a specific search order to find the value of that parameter. This order is:

1. Use the value given by an argument in the call to the image operator, else
2. Use the value associated with the appropriate *plane* involved in the operation, else

3. use the value kept in a special variable in the package specified by the *LLVS-USE-PACKAGE* special variable, else

   - use the default specified, or
   - ask the user for the value from the terminal.

The writer of the image operator has full control of the exact way in which these values are obtained but is encouraged to use the standard mechanisms.

4.5 Standard Arguments

These parameters are provided by most image operators.

- limits

  The *limits* determines the *conceptual plane* over which the operator processes pixels. This argument must be NIL or be a CommonLisp association list which contains any or all of the following value pairs:

  - :LEVEL

    This is the *level* of the *conceptual plane*. If it is not given, the highest *level* of all of the input *planes* is generally used. Any result *plane* allocated will be at this *level*.

  - :START-ROW

    This specifies the first row in the *conceptual plane* at which the operation is to be done.

  - :END-ROW

    This specifies the last row in the *conceptual plane* at which the operation is to be done.

  - :DELTA-ROW

    This specifies the stride — the value added to the row index each time around the row indexing loop.

  - :START-COL

    This specifies the first column in the *conceptual plane* at which the operation is to be done.

  - :END-COL

    This specifies the last column in the *conceptual plane* at which the operation is to be done.

  - :DELTA-COL

    This specifies the column stride — the value added to the column index each time around the column indexing loop.

- mask-plane

  This argument specifies a *mask-plane* that is used to control what pixels are visited in the conceptual plane. If the pixel of the *mask-plane* at the *current reference* is equal to one of the *mask-values*, the operation is performed at the corresponding pixels of the input *planes*, producing a result in the corresponding pixels of the output *planes*. Otherwise, no operation is performed at that pixel and no output pixel is stored.

- mask-value

  This is a sequence of values against which the *mask-plane* is compared. The comparison is done with integer values, truncating where necessary.

- bounds-action

  This argument specifies the operation to be performed if no pixel exists at the *current reference* in an input *plane*. The following values are allowed:

  - :ERROR — generate an error report.

    Only the first out-of-bounds pixel will be reported. Any further out-of-bounds references will be return the *background-value*. The total number of out-of-bounds references will be reported.

  - :BACKGROUND-VALUE — use the *background-value* of the *plane*.

  - :NEAREST-VALUE — use the nearest existing value in the *plane*.

4.6 Allocating Result Planes

Normally, each image operator allocates a new *plane* to contain the result of the operation. The caller of the image operator then binds this result to some symbol. This standard usage can be modified by using the **use-plane** or **static-plane** forms.

When different regions of an input *plane* are being processed separately, it is often useful to place each result in the same *plane*. This can be accomplished by using the *use-plane* form as in

```
(use-plane plane (image-operator ... ))
```

If this form is used, the result is placed in the specified *plane*. If this form is not used, the image operators will allocate a result *plane* of *type*, *size*, etc as deemed appropriate by the operator. This generally means a *plane* corresponding to the *conceptual plane* and of the same *type* as an input *plane*. The result of the *use-plane* form is the result *plane*.

Because of the way many CommonLisp systems work, allocating many large *planes* in the course of a computation will result in many painful waits for garbage collecting. To overcome this, some CommonLisp systems supply a way

of allocating *static* objects. These objects are not garbage collected. If a *plane* will be used for many computations or over a long period of time, making it *static* will improve performance. This can be accomplished by using the *static-plane* form as in

```
(static-plane name (image-operator ... ))
```

If this form is used, the result *plane* is allocated statically and bound to the *name* supplied as if a *setq* had been used. LLVS keeps a list (*LLVS-STATIC-PLANE-LIST*) of the symbols to which these *planes* are bound. Once allocated, the storage areas used can not be reclaimed.

The writer of an image operator must use the **result-plane** form instead of **construct-plane** or **new-plane** for allocating the result *plane* in order for **use-plane** and **static-plane** to have an effect.

The **find-all-planes** function will return a list of all variables bound to *planes*.

4.7 Using DEFINE-OPERATOR

This macro defines an image operator. The macro may be used to generate part or all of the Lisp code for an image operator. DEFINE-OPERATOR will generate the Lisp forms for obtaining the parameters, the interface forms for calling the C level functions, and even the entire body of the Lisp function. Whether or not DEFINE-OPERATOR generates any of this code depends on how the DEFINE-OPERATOR call is written.

The DEFINE-OPERATOR macro generates some initial code in the image operator. Part of this code binds the variable *OPERATOR-NAME* to the name of the image operator. This is the name used in error messages communicated to the user. The error processing logic references this *special* variable to obtain access to the description of the operator and any help information available. It is important that every image operator set up this special variable immediately upon entry so that the proper operator information is displayed if an error occurs. The description of the operator may obtained using
(documentation *operator-name* 'function)
and the help is obtained by
(documentation *operator-name* 'llvs-help)
The Lisp *form* is obtained using
(documentation *operator-name* 'llvs-form)
The CommonLisp DESCRIBE function will also display this information. The *OPERATOR-NAME* variable is declared as a special variable by using the CommonLisp DEFVAR facility when LLVS is created.

DEFINE-OPERATOR then generates the code to obtain each parameter *if* the parameter was defined with the needed information or is one of the standard parameters.

DEFINE-OPERATOR replaces every occurrance of CALL-P-SECTION, CALL-I-SECTION, or CALL-T-SECTION form with either a call to C-PER-PIXEL-DRIVER or a a form to call an *external* function. DEFINE-OPERATOR also generates any required linkage definitions such as the DEFINE-P-SECTION form and other forms required by the particular implementation of CommonLisp[15].

```
(define-operator name arguments description help body)
```

- name

  The operator name is used for help and other things and as the name of the generated Lisp function.

- arguments

  This argument is a list of all of the arguments that the image-operator function will accept. The argument list is specified in the same way as for any Lisp function with the following exceptions:

  1. If one of the arguments is one of the following symbols:
     - LIMITS,
     - MASK-PLANE
     - MASK-VALUE
     - BOUNDS-ACTION

     the symbol is also used to obtain the *iop-parameter* structure that defines the argument. In this case the DEFINE-OPERATOR macro generates a `setq` of the form:

     ```
     (setq symbol (get-parameter 'symbol
                                 llvs-symbol-def
                                 :arg-value symbol))
     ```

     Each of the special arguments is described using standard *iop-parameter* structures so that the help facilities of this system can describe the arguments for each operator to the user of the operator.

  2. If one of the arguments is preceded by *:special*, no special code is generated for getting the argument. As this is the default action except for the symbols listed above, use of *:special* is only necessary for those symbols when the user of DEFINE-OPERATOR wishes to write their own argument processing logic for those arguments.

[15] CALL-OUT and DEFINE-EXTERNAL-ROUTINE are the VaxLisp way of calling C routines from Lisp.

3. If the argument is given as a list, DEFINE-OPERATOR assumes that the first element on the list is the image operator argument name and that the remainder of the elements are to be supplied as arguments to a generated GET-PARAMETER call. The second element of the list is also checked. If it is one of the following (PLANE, FLOAT, INTEGER), a standard IOP-PARAMETER value is used in the generated GET-PARAMETER form.

Writers of image operators are expected to use the GET-PARAMETER function to obtain the values to be used for all the other arguments as well. This standardizes the *end* user interface. If, in addition, the definitions of the parameters are included in the DEFINE-OPERATOR form, this makes this information available to whomever requests help with the image operator.

- description

  This is the standard *short* CommonLisp type of description.

- help

  In addition to the standard documentation string, a *help* string is required. This *help* string is attached to the name of the image operator as a documentation string of type *llvs-help*. The *help* is displayed to users who request help using the image operator. It is meant to be significantly more detailed than the *description*. Generally, this help is obtained by using the CommonLisp DESCRIBE function.

- body

  This is the high level Lisp code for the image operator.

  If the body is not supplied, DEFINE-OPERATOR will generate a body that allocates one result *plane* and calls the P function with all the input *planes* and that result *plane*. The input *planes* must be specified using the PLANE designation for that argument to the image operator. For example:

```
(define-operator add-planes
                 ((p1 plane) (p2 plane))
                 "Description"
                 "Help")
```

The result *plane* is returned as the result of the image operator. The Sharable Image name is the same as the name of the image operator

(with "_" substituted for "-") and the P function entry point name is that name appended with "_P_SECTION".

DEFINE-OPERATOR searches each form of the *body* looking for the following forms:

- `(CALL-P-SECTION "Image Name"`
  `"Entry point name"`
  `input-planes`
  `output-planes`
  `... )`

  This is converted to a call to the C-PER-PIXEL-DRIVER function and the image name and entry point name are used to generate the DEFINE-P-SECTION form needed to locate and load the C function referenced. The `input-planes` and `output-planes` arguments must be forms that evaluate to a list of *planes*. Any other arguments supplied are assumed to be the standard keyword arguments required by C-PER-PIXEL-DRIVER. Any of the standard arguments not given are automatically generated here by DEFINE-OPERATOR (which also insures that the lexically scoped variables `limits`, `mask-plane`, `mask-value` and `bounds-action` are defined in every image operator).

- `(CALL-I-SECTION "Image name"`
  `"Entry point name"`
  `"variable definition"`
  `("variable definition" value-expression)`
  `... )`

  This form is converted to the non-CommonLisp forms needed to call the *external* function[16]. The interface definitions for the variables are generated from the *variable definition* strings which are in the form of C variable definition syntax. For example:

```
(CALL-I-SECTION "IMAGEOP"
                "IMAGEOP_I_SECTION"
                "int iterations"
                "float *initial-sum")
```

  defines two arguments to the C function IMAGEOP_I_SECTION. The first one is passed as a C `int`. The second one is passed as the address of a C `float`[17]. The variables in the Lisp code of the image

[16] When using VaxLisp, the CALL-OUT form and the required DEFINE-EXTERNAL-ROUTINE form are generated.

[17] Because C passes floating point values as type `double`, all Lisp floating point values must be passed as addresses.

operator should be *declared* as type `fixnum` and `single-float`.
An initial value for a variable may be specified if the list form is used. For example,

```
(CALL-I-SECTION "XXX" "YYY"
  ("int i" (1+ k)))
```

would evaluate (1+ k) and pass the value as i everytime YYY were called. *The initial value form may not be used with arguments that are passed by reference (i.e. an address is passed to the function).*

- `(CALL-T-SECTION ... )`
  CALL-T-SECTION and CALL-I-SECTION are processed identically

When DEFINE-OPERATOR generates any forms in addition to the Lisp function for the image operator[18], it places them in a *special* variable called *LLVS-SECTIONS* for your edification.

4.8 Using DEFIOP

This macro provides an easy way to define an image operator. The macro generates a DEFINE-OPERATOR form using many default arguments.

`(defiop name arguments description help body)`

The arguments to this macro are the same as for DEFINE-OPERATOR with the exception of *arguments*. Each symbol in this list is taken to be an argument to the operator specifying a *plane*. Thus, the form

```
(defiop intensity
  (red green blue)
  "sums the pixels from three planes"
  "To use this operator you must ..."
  (...))
```

expands to

```
(define-operator intensity
  ((red plane) (green plane) (blue plane)
   &key limits mask-plane mask-value bounds-action)
```

[18]For VaxLisp, DEFINE-P-SECTION and DEFINE-EXTERNAL-ROUTINE may be generated

```
"sums the pixels from three planes"
"To use this operator you must ..."
(...))
```

The *limits, mask-plane, mask-value,* and *bounds-action* arguments are considered standard, required arguments.

4.9 Using Access Functions with the *Per-pixel* Method

The pixels in the *planes* used by an image operator are obtained by calling *access functions.* Use of the access functions frees the image operator code from many details concerning representation of the pixel data, error checking, etc.

The type of result returned or value being set is specified by the writer of the image operator. The *driver* sets up the address of the proper *access function* that will convert the value in the *plane* to the format required by the C code. This allows the P function to be written without regard to the different pixel data formats allowed by the system.

The *access function* will convert from the one type to the other for the caller. When converting values from one type to another the possibility of truncation (float ⇒ int) and overflow (int ⇒ bit) must be considered. |*Not yet implemented — The first time an overflow occurs a warning message will be printed specifying where in the conceptual plane the overflow occurred. The value will be truncated. Subsequent overflow values will be truncated without notice. At the end of the operation, the number of overflows will be displayed.*| All the *access functions* will truncate without notice when converting from floating to integer.

These access functions are chosen by the *per-pixel driver* to convert from the *plane's* pixel type to the type required by the image operator and to do the proper out-of-bounds operation. The P function can be written, to a large extent, independent of the exact type of pixels in the *planes* that are being used. It does this by indirectly calling the *access functions* through external variables containing addresses of these functions. Each variable is an array where each element of the array corresponds to one *plane* of the operation. The *driver* sets up these variables using the types of the *planes* passed in as arguments.

Access functions are called indirectly through external variables set up by the *per-pixel driver.* The *driver* puts the address of the access function that should be used in C extern variables. The particular access function to be used is determined by the type of *plane* being accessed, the name of the variable referenced in the C code and what processing is desired for *out-of-bounds* references to the *plane.*

The names of the external variables are composed from the following options:

$$llvs\_\left\{\begin{array}{l} gp \\ sp \\ wp \end{array}\right\}\left\{\begin{array}{l} d \\ o \\ a \end{array}\right\}\left\{\begin{array}{l} i \\ f \end{array}\right\}|n|$$

where *gp* or *sp* specifies whether the pixel is retrieved or set and *wp* specifies that a set or *window* of pixels is returned. The *i* and *f* in the names specifies the type of value returned by the get functions or expected by the set functions. There is one set of these variables associated with each possible *plane*. The set is denoted by *n* where *n* is 0 for the first set, 1 for the second set, and so on. *n* is used as an index into the array of *access function* addresses and must match the order of the *planes* as passed to the *per-pixel driver*.

While the *gp* access functions can be used to obtain individual pixels in a *plane*, there are times when the image operator will want to access a group of pixels around some pixel. The *wp*[19] functions are designed to obtain this group with one function call. These functions fill a vector of elements that is passed as an argument[20].

In frequent cases the image operator is only interested in accessing pixels at the *current reference*. In this case special *direct* access functions are provided that do not have the row and column arguments. These functions are the *d* functions.

For the *o* functions, pixels are referenced offset from the current pixel (*current reference*) of the *plane*. The offset is considered to be at the *level* of the actual *plane* and not at the *level* of the *conceptual plane*.

For the *a* functions, absolute referencing is used where the reference is offset from the absolute *origin* (0,0). That is — absolute (0,0) is considered the *current reference*[21]. The offset is considered to be at the *level* of the actual *plane* and not at the *level* of the *conceptual plane*.

If the referenced pixel does not actually exist, these functions will generate an error to inform the user. This normal operation can be modified by the caller of the image operator specifying some value for the :BOUNDS-ACTION standard parameter. When this is done, the *per-pixel driver* puts the addresses of other access functions in the external variables to handle these options. For the *sp* functions, no value is stored. For the *gp* and *wp* functions a value is obtained. The :BACKGROUND-VALUE versions return the *background-value* from the *plane's* descriptor. The :NEAREST-VALUE versions return the nearest value in the *plane* as shown in Figure 4.

The following arrays of *access function* addresses are set up by the `c-per-pixel-driver` where MAXPLANE is the maximum number of input and output *planes* allowed. The array entries for which there is no corresponding *plane* are set with the address of an error function.

```
#define  MAXPLANE 9

globalref int    (*llvs_gpdi[MAXPLANE])();
```

[19] No *wpo* versions are defined.

[20] The user may pass in a matrix or vector.

[21] This means that the row and column arguments passed to these functions are relative to absolute (0,0).

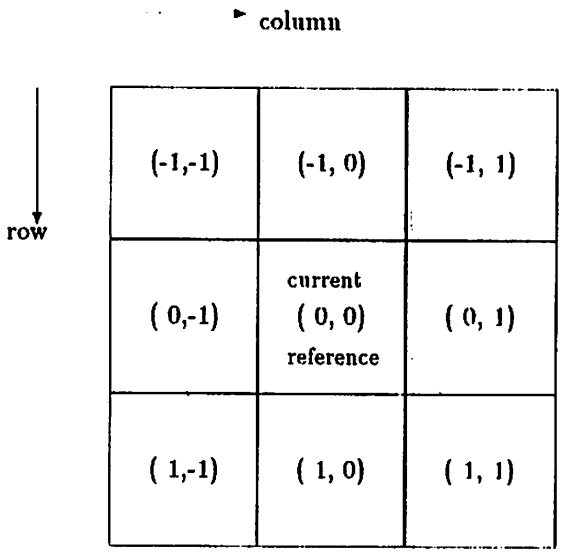

Figure 3: Relative addressing for a 3 x 3 window.

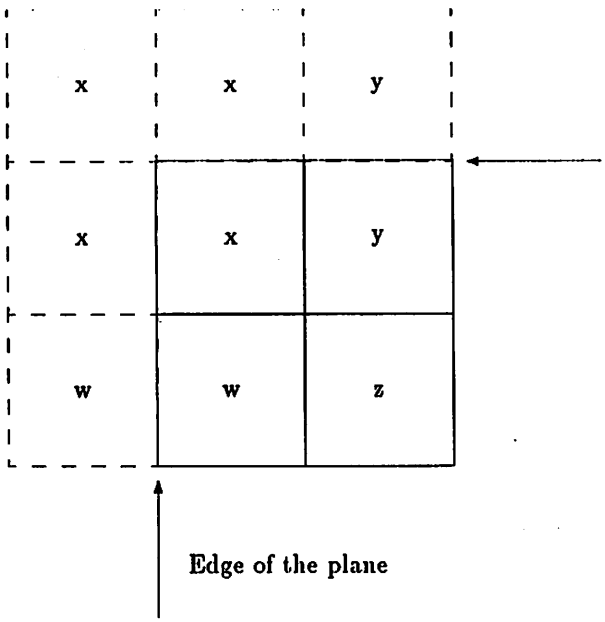

Figure 4: Nearest Value Example.

0	1	2
3	current 4 reference	5
6	7	8

Figure 5: Pixel order from a 3 x 3 window at offset (-1,-1).

```
globalref float (*llvs_gpdf[MAXPLANE])();

globalref int   (*llvs_gpoi[MAXPLANE])();
globalref float (*llvs_gpof[MAXPLANE])();

globalref int   (*llvs_gpai[MAXPLANE])();
globalref float (*llvs_gpaf[MAXPLANE])();

globalref void  (*llvs_spdi[MAXPLANE])();
globalref void  (*llvs_spdf[MAXPLANE])();

globalref void  (*llvs_spoi[MAXPLANE])();
globalref void  (*llvs_spof[MAXPLANE])();

globalref void  (*llvs_spai[MAXPLANE])();
globalref void  (*llvs_spaf[MAXPLANE])();

globalref void  (*llvs_wpdi[MAXPLANE])();
globalref void  (*llvs_wpdf[MAXPLANE])();

globalref void  (*llvs_wpai[MAXPLANE])();
globalref void  (*llvs_wpaf[MAXPLANE])();
```

The *gpd* functions take only one argument:

```
(*llvs_gpdi[plane])(plane)
```

- plane

  This is the number of the plane from which the pixel is to be obtained. The numbering is relative to the order the planes are specified in the call to the *per-pixel driver* and must match the number in the array index. The numbering starts at 0.

The *spd* functions have a second argument:

```
(*llvs_spdf[plane])(value, plane)
```

- value

  This is the value to be set in the *plane* at the position specified.

The *gpo* functions take three arguments:

```
(*llvs_gpof[plane])(plane, rowoffset, coloffset)
```

- plane

  This is the number of the plane from which the pixel is to be obtained. The numbering is relative to the order the planes are specified in the call to the *per-pixel driver* and must match the number in the array index. The numbering starts at 0.

- rowoffset

  This is the row relative to the current reference in *this plane* and is not affected by the *level* of the *conceptual plane* over which the image operator is being applied. Valid values are 0 and positive or negative integers.

- coloffset

  This is the column relative to the current reference in *this plane* and is not affected by the *level* of the *conceptual plane* over which the image operator is being applied. Valid values are 0 and positive or negative integers.

The *spo* functions have a fourth argument:

```
(*llvs_spoi[plane])(value, plane, rowoffset, coloffset)
```

- value

  This is the value to be set in the *plane* at the position specified.

The *wpd* functions take six arguments:

```
(*llvs_wpdf[plane])(plane, rowoffset, coloffset,
                    numrows, numcols, vector)
```

- plane

  This is the number of the plane from which the pixels are to be obtained. The numbering is relative to the order the planes are specified in the call to the *per-pixel driver* and must match the number in the array index. The numbering starts at 0.

- rowoffset

  This is the row relative to the current reference in *this plane* and is not affected by the *level* of the *conceptual plane* over which the image operator is being applied. Valid values are 0 and positive or negative integers. The row is the top edge of the window.

- coloffset

  This is the column relative to the current reference in *this plane* and is not affected by the *level* of the *conceptual plane* over which the image operator is being applied. Valid values are 0 and positive or negative integers. The column is the left edge of the window.

- numrows

  This is the number of rows in the window.

- numcols

  This is the number of columns in the window.

- vector

  This is the array that the values will be placed in. The calling function must allocate this array and it must be of sufficient size to hold all the values requested. This array is not returned as the function result.

Examples:

```
/* get a 3 x 3 window around the current reference in plane 2 */

float vec[9];

  (*llvs_wpdf[2])( 2, -1, -1, 3, 3, vec);

/* get a 3 x 4 window around the current reference in plane 2 */

float vec[3][4];

  (*llvs_wpdf[2])(2, -1 -2, 3, 4, vec);
```

5 Example Image Operators

Annotated examples of image operators are shown. The first image operator is the INTENSITY operation that produces a new *plane* that is the average of three input *planes* (i.e. red green blue). The two examples of intensity illustrate the two different ways in which an image operator may be written. The last example is of STATS-PLANE written using the *per-pixel* method. This example illustrates how arguments may be passed back and forth to the C code.

5.1 INTENSITY: Using the *Per-pixel* Method

Writing an image operator using the *per-plane* method is the easiest way to write one. You trade efficiency for this ease.

Notes

1. The DEFIOP macro is like the CommonLisp DEFUN form. It requires the name of the resultant function and the list of arguments. The intensity operator has three required arguments — the three *planes*.

   Since it is written using DEFIOP all of the specified arguments are assumed to be *planes* and the standard arguments are provided as **&key** arguments by default. Calls to GET-PARAMETER are generated for all the arguments. The DEFINE-OPERATOR form could have been used as well. DEFINE-OPERATOR provides more flexibility and is, correspondingly, more difficult to use.

   Unlike DEFUN, the CommonLisp *documentation-string* is required. In addition, a second string is also required. This string should contain information useful to someone wanting to know what the image operator does and how to call it. This information is displayed by the CommonLisp DESCRIBE function. DEFIOP causes DESCRIBE to automatically include the CommonLisp form for calling the image operator.

2. The result *plane* must be allocated. **Result-plane** is used so that the **use-plane** and **static-plane** may be used by the caller of the image operator.

   As the documentation stated the result *plane* will be of type :BYTE. The *level* and *size* of this *plane* is calculated by **union-plane-limits*** to be essentially as large as the largest input *plane* and at the highest *level* of the three *planes* unless modified by the caller specifying some *limits*. The **limits** argument supplied here is a **keyword** argument automatically placed in the **intensity** function's argument list by **defiop**.

3. Most image operators return a *plane* — the one generated by the image operator. C-PER-PIXEL-DRIVER returns the list of output planes.

4. C-PER-PIXEL-DRIVER is used because this example is written using the *per-pixel* method. In this method, your image operator need only worry about what to do at each pixel. Getting to each pixel is the responsibility of the *per-pixel driver.*

   The *per-pixel driver* must be told what C routine to use and what type of data format the routine expects each pixel to be in. This is provided by the *p-section* given as the first argument. Setting up the *p-section* is discussed in Note 6.

   The second argument must be a list of all the input *planes* used. The third argument must be a list of all the output *planes* used. It is not possible to accidently set a pixel in an input *plane*; if a *plane* is needed for both input and output, it must be passed in both lists.

5. These are the standard arguments. You must include them exactly as shown for the use of the *per-pixel driver.*

6. The DEFINE-P-SECTION form is much like the CommonLisp DEFVAR form but builds the information needed by the C-PER-PIXEL-DRIVER so that it can find the code for the P section. This is provided by the Sharable Image name (i.e. INTENSITY) and the entry point name (i.e. INTENSITY_P_SECT). Note that the name *intensity-p-section* is the same as specified when C-PER-PIXEL-DRIVER was called.

7. DEFINE-OPERATOR and DEFIOP will generate the P section if the CALL-P-SECTION form is used in place of C-PER-PIXEL-DRIVER. The equivalent call is shown here.

8. DEFINE-OPERATOR and DEFIOP will generate the body of the image operator if it is left out. This example generates the equivalent of the first two examples of INTENSITY.

9. Doing this `#include` brings in the necessary definitions to allow the C code to use the things set up by the *per-pixel driver.* Part of this is the external specifications for the access functions.

10. Macros are defined for the *access function* calls needed. These macros make the code more readable, allow changes to be made more readily, and with fewer errors.

    One of the things the C code needs is some way to obtain and set each pixel. This is provided by a set of *access functions* set up by the *per-pixel driver.* The *per-pixel driver* sets up a set of *access functions* for each *plane.* These *access functions* are called by your code to get or set individual pixels. The external variables actually contain the address of the *access function.* In this example the simplest *access functions* are used. They

only get or set one pixel. You may use other *access functions* on a *plane* separately or at the same time.

The *access functions* convert the pixel to the type declared in the name of the variable. Therefore, to get pixels represented as integers, use the variables ending in *i*. Use the variables ending in *f* for pixels represented as C `floats`.

11. This is the entry point of the C code routine — it must match the name given in your DEFINE-P-SECTION declaration. This entry point is called by the *per-pixel driver* once for each pixel in the *conceptual plane*[22]. It is not possible to pass any arguments to this routine (see the STATS-PLANE example) and the *per-pixel driver* does not expect the routine to return any value.

12. The intensity operator is defined as averaging the pixels of three input *planes* to produce one pixel of the result *plane*. Note how each pixel is obtained using an indirect call to the *access function* set up by the *per-pixel driver*. As *access functions* are written with no knowledge of any particular *plane* or *plane* number, they must be passed the *plane* number so that they can obtain the address of the proper *plane*. This numbering MUST match the order in which the *planes* are given in the list to C-PER-PIXEL-DRIVER. Failure to do this will result in unexpected and, possibly, undetected errors!

    Note that the actual averaging is done using floating point.

13. The result pixel is stored into the result *plane*. The *access function* does the conversion to the type of the actual result *plane*.

Lisp Code for Intensity

```
(defiop intensity  (red green blue)              ; Note 1

"This operator produces an intensity plane from three input planes."

"The result is a BYTE plane whose each pixel is the average of the
pixels
from each of the three input planes."

  (let
    ((result                                     ; Note 2
       (result-plane
         :byte
```

[22]This is true for every pixel selected by the *mask-plane*.

```
        (union-plane-limits* limits red green blue))))

    (first                                          ; Note 3
      (c-per-pixel-driver                           ; Note 4
        intensity-p-section
        (list red green blue)
        (list result)
                                                    ; Note 5
        :mask-plane mask-plane
        :mask-value mask-value
        :bounds-action bounds-action
        :limits limits)))


(define-p-section intensity-p-section               ; Note 6
                  "INTENSITY"
                  "INTENSITY_P_SECT")

(defiop intensity  (red green blue)                 ; Note 7

"..."

"..."

(first
  (call-p-section
    "INTENSITY"
    "INTENSITY_P_SECT"
    (list red green blue)
    (list (result-plane
            :byte
            (union-plane-limits* limits red green blue))))))

(defiop intensity  (red green blue)                 ; Note 8

"..."

"..."
)
```

C Code for Intensity

```
                                                    /* Note 9   */
#include "llvs:llvs_per_pixel.h"
                                                    /* Note 10  */
```

```
#define intensity(val) (*llvs_spdf[0])(val,0)
#define red (*llvs_gpdf[0])(0)
#define green (*llvs_gpdf[1])(1)
#define blue (*llvs_gpdf[2])(2)

/* intensity p section */
                                                /* Note 11  */
void intensity_p_sect ()
{
float rval;
                                                /* Note 12  */
rval = (red + green + blue)/3.0;
                                                /* Note 13  */
intensity(rval);
}
```

5.2 INTENSITY: Using the *Per-plane* Method

Using the *per-plane* method is much more difficult than using the *per-pixel* method because you have to supply all of the logic normally provided by the *per-pixel drivers*. The payoff is faster execution.

Notes

1. The DEFIOP macro is like the CommonLisp DEFUN form. It requires the name of the resultant function and the list of arguments. The intensity operator has three required arguments — the three *planes*.

   Since it is written using DEFIOP all of the specified arguments are assumed to be *planes* and the standard arguments are provided as **&key** arguments by default. Calls to GET-PARAMETER are generated for all the arguments. The DEFINE-OPERATOR form could have been used as well. DEFINE-OPERATOR provides more flexibility and is correspondingly more difficult to use.

   Unlike DEFUN, the CommonLisp *documentation-string* is required. In addition, a second string is also required. This string should contain information useful to someone wanting to know what the image operator does and how to call it. This information is displayed by the CommonLisp DESCRIBE function. DEFIOP causes DESCRIBE to automatically include the CommonLisp form for calling the image operator.

2. The *level* and *limits* of the *conceptual plane* of the operation are calculated by `union-plane-limits*` to be essentially as large as the largest input *plane* and at the highest *level* of the three *planes* unless modified by the caller specifying some *limits*.

3. The structure of the *conceptual plane* must be passed to the C code for the image operator so that it knows what rows and columns to use. BUILD-LIMITS-C constructs a C `struct` structure for communicating these values.

4. The result *plane* must be allocated. `Result-plane` is used so that the `use-plane` and `static-plane` may be used by the caller of the image operator.

   As the documentation stated the result *plane* will be of type :BYTE. It is as large as the *conceptual plane*.

5. Since the *per-plane* method is being used to write the image operator instead of the other methods available, a vaild assumption is that the speed of the operator is important. For this reason, the code is written to handle two cases — with a *mask-plane* and without one. This eliminates the testing for the match with a *mask-value* at every pixel in the *conceptual plane* when no *mask-plane* is supplied. The `mask-plane` argument is a `keyword` argument that is automatically put in the `intensity` function's argument list by `defiop`.

6. When there is a *mask-plane*, there must be *mask-values* that must be passed to the C code.

7. The C code also needs access to all the things that were handled by the C-PER-PIXEL-DRIVER for the *per-pixel* method. These include the *location, level* and *background-value* of each *plane*. All *planes* used including the *mask-plane* must be included. This information is placed in a statically defined area referenced vis the *special* global variable *LLVS-PLANE-INFO-VECTOR*.

   It is also possible to pass this information individually for each plane.

8. The VaxLisp CALL-OUT facility is used to call the C routine. The first argument must be the name used in the VaxLisp DEFINE-EXTERNAL-ROUTINE declaration to describe the C routine so that it may be found and called. The number, type, and order of the arguments in the C code MUST match the specification in DEFINE-EXTERNAL-ROUTINE.

   In order that all image operators written using the *per-plane* method be as consistent as possible the following rules should be followed:

   - The *limits* should be the first argument.
   - If a *mask-plane* is being used, the *mask-values* structure should be the second argument.
   - The information about the *planes* should follow the *mask-values* argument.

- Any other values needed should follow the plane information.
- Finally, the actual *planes* should be last. And, they should be in the order: *mask-plane*, result *planes*, input *planes*.

9. Each plane is passed using PASS-PLANE-PIXELS or PLANE-PIXELS-C. PLANE-PIXELS-C will report an error if the *plane* is NIL while PASS-PLANE-PIXELS will use a dummy *plane* in that case. You must use one of these rotuines to pass a *plane* to the C code.

10. This is the call for the case where there is no *mask-plane*. Note the different name used for the C routine and the lack of any mention of *mask-plane* or *mask-values*. A different DEFINE-EXTERNAL-ROUTINE is also required.

11. The DEFINE-EXTERNAL-ROUTINE declaration is used to inform VaxLisp of what the C rotuine looks like. The Sharable Image name and entry point must be specified. Each argument is also specified. You should copy the arguments given here exactly for your image operator written using the *per-plane* method. You may include additional ones as well. If the additional ones are not *planes*, refer to the VaxLisp Reference Manual for a description of how to define them.

    The DEFINE-EXTERNAL-ROUTINE must be evaluated before the corresponding CALL-OUT is evaluated.

12. See Note 11 and simply note that the entry point name is different and that the *mask-plane* is not being used.

13. These `#include` statements must be included in order for your program to access the structures being passed in correctly.

14. The order and number of arguments used by the C routine must match that specified in the DEFINE-EXTERNAL-ROUTINE declaration. The type of arguments must also match. Use the type specifiers as given here.

15. Since the image operator written this way does its own indexing over the argument *planes*, it needs a lot of local variables.

16. The argument *planes* may each be at a different *level* from the *level* of the *conceptual plane* of the operation. The difference must be known for each *plane* so that it may be indexed properly.

17. This example image operator ignores the :BOUNDS-ACTION standard argument. It uses the :BACKGROUND-VALUE method only. This logic obtains the *background-value* for each *plane*. If you wish to use the :BOUNDS-ACTION argument, you must pass it in as an additional argument and use a `switch` statement when pixel accessing is done.

18. This `for` loop indexes over the rows in the *conceptual plane*.

19. These statements convert the row index from the *conceptual plane* to a row index at the *level* and *location* of each actual *plane*.

20. This `for` loop indexes over the columns in the *conceptual plane*.

21. These statements convert the column index from the *conceptual plane* to a column index at the *level* and *location* of the *mask-plane*, obtain the pixel at that location, and check to see if its value is in the *mask-values* vector. If it is, the code to do the intensity calculation is called. Note that the `break` terminates the `for` loop when a value is matched.

22. These statements convert the column index from the *conceptual plane* to a column index at the *level* and *location* of each actual input *plane* and the result *plane*.

23. The pixels from each input *plane* are obtained. If :BOUNDS-ACTION were being used, a `switch` statement would be required here to select the appropriate GET_PIXEL and SET_PIXEL macro.

24. The calculation is made and the result pixel stored.

25. This is the routine for the case when no *mask-plane* is used. Note the different name and compare it to the previous routine.

Lisp Code for INTENSITY

```
(defiop intensity (red green blue)              ; Note 1

"This operator produces an intensity plane from three input planes."

"The result is a BYTE plane whose each pixel is the average of the
pixels
from each of the three input planes."

(let                                            ; Note 2
  ((union-lims (union-plane-limits* limits red green blue)))

  (let
    ((c-limits (build-limits-c                  ; Note 3
  union-lims))
     (result                                    ; Note 4
       (result-plane :unsigned-byte union-lims))

    (if mask-plane                              ; Note 5
```

```
      ; a mask plane exists

      (let
                                              ; Note 6
        ((c-mask-values
            (build-mask-values-c mask-value)))
                                              ; Note 7
         (build-plane-info-vector-c
           (list mask-plane result red green blue))
         (call-out                            ; Note 8
           intensity-p-sect
           c-limits
           c-mask-values
           *llvs-plane-info-vector*
           (plane-pixels-c mask-plane)        ; Note 9
           (pass-plane-data result)
           (pass-plane-data red)
           (pass-plane-data blue)
           (pass-plane-data green)
           ))

        ; there is no mask plane specified

        (progn                                ; Note 10
          (build-plane-info-vector-c (list result red green blue))
          (call-out
            intensitynm-p-sect
            c-limits
            *llvs-plane-info-vector*
            (pass-plane-data result)
            (pass-plane-data red)
            (pass-plane-data blue)
            (pass-plane-data green)
            )))
    result))) ; Return the result plane

(define-external-routine                      ; Note 11
(intensity-p-sect :image-name "INTENSITY"
                  :entry-point "INTENSITY_P_SECT")
  (limits :access :in
          :lisp-type alien-structure
          :mechanism :reference)
  (mask-values :access :in
               :lisp-type alien-structure
```

```
                :mechanism :reference)
  (plane-info :access :in
              :lisp-type alien-structure
              :mechanism :reference)
  (mask-plane :access :in
              :lisp-type alien-structure
              :mechanism :reference)
  (result :access :in
          :lisp-type alien-structure
          :mechanism :reference)
  (red :access :in
       :lisp-type alien-structure
       :mechanism :reference)
  (green :access :in
         :lisp-type alien-structure
         :mechanism :reference)
  (blue :access :in
        :lisp-type alien-structure
        :mechanism :reference)
  )

(define-external-routine                          ; Note 12
  (intensitynm-p-sect :image-name "INTENSITY"
                      :entry-point "INTENSITYNM_P_SECT")
  (limits :access :in
          :lisp-type alien-structure
          :mechanism :reference)
  (plane-info :access :in
              :lisp-type alien-structure
              :mechanism :reference)
  (result :access :in
          :lisp-type alien-structure
          :mechanism :reference)
  (red :access :in
       :lisp-type alien-structure
       :mechanism :reference)
  (green :access :in
         :lisp-type alien-structure
         :mechanism :reference)
  (blue :access :in
        :lisp-type alien-structure
        :mechanism :reference)
  )
```

C Code for INTENSITY

```
                                                    /* Note 13 */
#include "llvs:llvs_per_pixel.h"
#include "llvs:llvs_per_plane.h"
                                                    /* Note 14 */
intensity_p_sect(limits,mask_values,plane_info,
                 mask_plane,result,red,green,blue)
LIMITS *limits;
MASK_VALUES *mask_values;
PLANE_INFO *plane_info[MAXPLANES];
PLANE *mask_plane;
PLANE *result;
PLANE *red;
PLANE *green;
PLANE *blue;
{                                                   /* Note 15 */
  int currow,curcol;
  int mroff,xroff,rroff,groff,broff;
  int mcoff,xcoff,rcoff,gcoff,bcoff;
  int mdelta,xdelta,rdelta,gdelta,bdelta;
  int m_pix,imaskv,mbgv;
  float rbgv,gbgv,bbgv;
  float intensity,r_pix,g_pix,b_pix;
                                                    /* Note 16 */
  mdelta = limits->level - plane_info[0]->level;
  xdelta = limits->level - plane_info[1]->level;
  rdelta = limits->level - plane_info[2]->level;
  gdelta = limits->level - plane_info[3]->level;
  bdelta = limits->level - plane_info[4]->level;
                                                    /* Note 17 */
  if (mask_plane->datatype == FLOAT)
    mbgv = plane_info[0]->background.flonum;
  else
    mbgv = plane_info[0]->background.fixnum;

  if (red->datatype == FLOAT)
    rbgv = plane_info[2]->background.flonum;
  else
    rbgv = plane_info[2]->background.fixnum;

  if (green->datatype == FLOAT)
    gbgv = plane_info[3]->background.flonum;
  else
    gbgv = plane_info[3]->background.fixnum;
```

```
  if (blue->datatype == FLOAT)
    bbgv = plane_info[4]->background.flonum;
  else
    bbgv = plane_info[4]->background.fixnum;
                                                /* Note 18 */
  for (currow = limits->startrow;
       currow <= limits->endrow;
       currow += limits->deltarow) {
                                                /* Note 19 */
    TRANSLEVEL(mroff,currow,mdelta,plane_info[0]->row_location);
    TRANSLEVEL(xroff,currow,xdelta,plane_info[1]->row_location);
    TRANSLEVEL(rroff,currow,rdelta,plane_info[2]->row_location);
    TRANSLEVEL(groff,currow,gdelta,plane_info[3]->row_location);
    TRANSLEVEL(broff,currow,bdelta,plane_info[4]->row_location);
                                                /* Note 20 */
    for (curcol = limits->startcol;
         curcol <= limits->endcol;
         curcol += limits->deltacol) {
                                                /* Note 21 */
      TRANSLEVEL(mcoff,curcol,mdelta,plane_info[0]->column_location);

      GET_PIXEL(m_pix,mbgv,mask_plane,mroff,mcoff,plane_info[0]);

      for (imaskv = 0; imaskv < mask_values->num_values; imaskv++)
        if (m_pix == mask_values->value_base[imaskv]) {
                                                /* Note 22 */
          TRANSLEVEL(xcoff,curcol,xdelta,plane_info[1]->column_location);
          TRANSLEVEL(rcoff,curcol,rdelta,plane_info[2]->column_location);
          TRANSLEVEL(gcoff,curcol,gdelta,plane_info[3]->column_location);
          TRANSLEVEL(bcoff,curcol,bdelta,plane_info[4]->column_location);
                                                /* Note 23 */
          GET_PIXEL(r_pix,rbgv,red,rroff,rcoff,plane_info[2]);
          GET_PIXEL(g_pix,gbgv,green,groff,gcoff,plane_info[3]);
          GET_PIXEL(b_pix,bbgv,blue,broff,bcoff,plane_info[4]);
                                                /* Note 24 */
          intensity = (r_pix + g_pix + b_pix)/3.0;

          SET_PIXEL(intensity,result,xroff,xcoff,plane_info[1]);

          break;
        }
    }
  }
}
```

```
                                                    /* Note 25 */
intensitynm_p_sect(limits,plane_info,
                   result,red,green,blue)
LIMITS *limits;
PLANE_INFO *plane_info[MAXPLANES];
PLANE *result;
PLANE *red;
PLANE *green;
PLANE *blue;
{
  int currow,curcol;
  int xroff,rroff,groff,broff;
  int xcoff,rcoff,gcoff,bcoff;
  int xdelta,rdelta,gdelta,bdelta;
  int imaskv,mbgv;
  float rbgv,gbgv,bbgv;
  float intensity,r_pix,g_pix,b_pix;

  xdelta = limits->level - plane_info[0]->level;
  rdelta = limits->level - plane_info[1]->level;
  gdelta = limits->level - plane_info[2]->level;
  bdelta = limits->level - plane_info[3]->level;

  if (red->datatype == FLOAT)
    rbgv = plane_info[1]->background.flonum;
  else
    rbgv = plane_info[1]->background.fixnum;

  if (green->datatype == FLOAT)
    gbgv = plane_info[2]->background.flonum;
  else
    gbgv = plane_info[2]->background.fixnum;

  if (blue->datatype == FLOAT)
    bbgv = plane_info[3]->background.flonum;
  else
    bbgv = plane_info[3]->background.fixnum;

  for (currow = limits->startrow;
       currow <= limits->endrow;
       currow += limits->deltarow) {

    TRANSLEVEL(xroff,currow,xdelta,plane_info[0]->row_location);
    TRANSLEVEL(rroff,currow,rdelta,plane_info[1]->row_location);
```

```
    TRANSLEVEL(groff,currow,gdelta,plane_info[2]->row_location);
    TRANSLEVEL(broff,currow,bdelta,plane_info[3]->row_location);


    for (curcol = limits->startcol;
         curcol <= limits->endcol;
         curcol += limits->deltacol) {

      TRANSLEVEL(xcoff,curcol,xdelta,plane_info[0]->column_location);
      TRANSLEVEL(rcoff,curcol,rdelta,plane_info[1]->column_location);
      TRANSLEVEL(gcoff,curcol,gdelta,plane_info[2]->column_location);
      TRANSLEVEL(bcoff,curcol,bdelta,plane_info[3]->column_location);


      GET_PIXEL(r_pix,rbgv,red,rroff,rcoff,plane_info[1]);
      GET_PIXEL(g_pix,gbgv,green,groff,gcoff,plane_info[2]);
      GET_PIXEL(b_pix,bbgv,blue,broff,bcoff,plane_info[3]);

      intensity = (r_pix + g_pix + b_pix)/3.0;

      SET_PIXEL(intensity,result,xroff,xcoff,plane_info[0]);

    }
  }
}
```

5.3 STATS-PLANE: Passing values between C and Lisp

This example shows how you may use an I section and a T section to pass values between the Lisp and C levels of your image operator. The I section is used for initialization of any values other than *planes* required by the P section. The T section is used to pass back to the Lisp level any values computed by the P section[23].

STATS-PLANE generates some statistics about a *plane*. To do this it must count the number of pixels and sum the values of all the pixels. Since the P section is called once per pixel, it is inconvenient to put logic in it to test if it is the first time it is called so that it can zero the accumulators. Instead, we use an I section that is called once so that it may do the initialization.

As the P section can not have any arguments, it can not pass back any values to the Lisp level. Instead, we call a T section to obtain these values and pass them back. For STATS-PLANE, the T section passes back the accumulators initialized by the I section and utilized by the P section.

[23] You are not restricted to just these two sections. You may call as many C routines are you want. However, most problems can be handled with these two sections or less.

Notes

1. A DEFINE-EXTERNAL-ROUTINE is needed for each routine used[24]. This declaration specifies the Sharable Image containing the specified entry point and the number, order and type of arguments to be passed which must agree with the C code. In the case of the STATS-PLANE, we have set up two un-needed arguments in the I section as examples. The first argument passes in an integer. The second one passes in a floating point value. Note that the :VAX-TYPE is :F-FLOATING and the :MECHANISM is :REFERENCE. This is because C routines only pass double floating point numbers as arguments while the standard Vax calling sequence limits each argument to one word. Thus, the address of the value must be passed. You should refer to the VaxLisp Reference Manual for more information.

   The DEFINE-EXTERNAL-ROUTINE declaration must be evaluated before the corresponding CALL-OUT is evaluated.

2. The T section declaration is more complicated as many more arguments are being used. Note that in order to pass values back the :REFERENCE mechanism must be used.

3. Since this is an image operator using the *per-pixel* method, a DEFINE-P-SECTION declaration is needed too.

4. See the other *per-pixel* example for a discussion of this part.

5. The values being passed back need to have a variable to be bound too. Because of the way some Lisps work, it is a good idea to initialize floating point variables with ***unique*** values.

6. Call the I section and pass it two values. Note the match of the name to the DEFINE-EXTERNAL-ROUTINE declaration. The values being passed could be calculated by a Lisp expression as long as the arguments are defined to be :ACCESS :IN.

7. Call the *per-pixel driver*. It will call the P section routine for all the pixels in the *plane*. Note that the list of output *planes* is NIL.

8. Call the T section to obtain the result values. The arguments should be variables and not expressions when :ACCESS :IN-OUT is used.

9. The remainder of the Lisp code calculates the statistics based on the accumulated values and places them on the *plane's* association list.

[24]Except for the P section which is called by the C-PER-PIXEL-DRIVER and has no arguments.

10. Of course, DEFIOP or DEFINE-OPERATOR will also generate the DEFINE-EXTERNAL-ROUTINE forms for you if you want. This example shows the equivalent form for STATS-PLANE using the CALL-I-SECTION and CALL-T-SECTION forms. The arguments are represented using C declarations (except that the variable names must match the Lisp variable names) from which the appropriate argument definition is constructed for the DEFINE-EXTERNAL-ROUTINE. Because this format currently expects a variable, the constants 0 and 0.0 had to be set up in additional Lisp variables first this time.

11. The `#include` obtains definitions needed for accessing pixels in the P section. The accumulators are defined.

12. The I section routine is defined. The arguments must match the DEFINE-EXTERNAL-ROUTINE declaration in number, type, and order .

    For illustrative purposes the un-needed arguments are displayed. Note that `abc` is an address and not the value itself.

    The accumulators are initialized. This can not be done with a static declaration as STATS-PLANE gets called more than once.

13. The P section is called once for each pixel. See the *per-pixel* example for discussion of a similar P section.

14. In order to avoid using some artificial value for the initial maximum and minimum values, this logic is used instead. Note that it depends on `number pixels` being initialized properly.

15. The T section routine passes back the values of the accumulators. This is done by using the addresses passed to the T section to store the values. All values passed back must be passed back this way.

Lisp Code for STATS-PLANE

```
(define-external-routine                          ; Note 1
  (stats-i-sect :image-name "STATS_PLANE"
                :entry-point "STATS_PLANE_I_SECT")
  (initial-pixels :access :in
                  :lisp-type fixnum
                  :mechanism :value
                  :vax-type :longword)
  (abc :access :in
       :lisp-type single-float
       :mechanism :reference
       :vax-type :f-floating))
```

```
(define-external-routine                          ; Note 2
  (stats-t-sect :image-name "STATS_PLANE"
                :entry-point "STATS_PLANE_T_SECT")
  (number-pixels :access :in-out
                 :lisp-type fixnum
                 :mechanism :reference
                 :vax-type :longword)
  (sum-v :access :in-out
         :lisp-type single-float
         :mechanism :reference
         :vax-type :f-floating)
  (sum-row-v :access :in-out
             :lisp-type single-float
             :mechanism :reference
             :vax-type :f-floating)
  (sum-column-v :access :in-out
                :lisp-type single-float
                :mechanism :reference
                :vax-type :f-floating)
  (sum-vv :access :in-out
          :lisp-type single-float
          :mechanism :reference
          :vax-type :f-floating)
  (maxval :access :in-out
          :lisp-type single-float
          :mechanism :reference
          :vax-type :f-floating)
  (minval :access :in-out
          :lisp-type single-float
          :mechanism :reference
          :vax-type :f-floating))

                                                  ; Note 3
(define-p-section
  stats-plane-p-section
  "STATS_PLANE"
  "STATS_PLANE_P_SECT")

(defiop stats-plane (input-pl)                    ; Note 4

"Places statistics about a plane on its associations list."

"This operator obtains the following values from a plane  and
places them on the plane's association list.  The input plane
is returned as the result.
```

```
:maxval             - maximum pixel value

:minval             - minimum pixel value

:mean               - mean pixel value

:average            - average pixel value

:standard-deviation - the standard deviation of the pixels.

:center-of-gravity  - the location of the center-of-gravity"

(let                                              ; Note 5
  ((maxval (random 1.0)) (minval (random 1.0)) (row 0) (col 0)
   (number-pixels 0) (sum-row-v (random 1.0))
   (sum-column-v (random 1.0))
   (sum-v (random 1.0)) (sum-vv (random 1.0)))

  (declare (fixnum number-pixels row col)
           (single-float maxval minval sum-row-v
            sum-column-v sum-vv))

  (call-out stats-i-sect 0 0.0)                   ; Note 6
  (c-per-pixel-driver stats-plane-p-section       ; Note 7
                      (list input-pl)
                      NIL
                      :limits limits
                      :mask-plane mask-plane
                      :mask-value mask-value
                      :bounds-action bounds-action)
  (call-out stats-t-sect number-pixels            ; Note 8
                         sum-v
                         sum-row-v
                         sum-column-v
                         sum-vv
                         maxval
                         minval)
  (let                                            ; Note 9
    ((mean 0.0) (average 0.0)
     (sd-v-2 0.0) (stan-dev 0.0)
     (numpixs (coerce number-pixels 'single-float)))

    (declare (single-float mean average
```

```
            sd-v-2 numpixs stan-dev))

      (if (= number-pixels 0)
          (vis-error "Number of pixels sampled was zero!"
                     :error-level :warning)
          (progn
            (setq average (/ sum-v numpixs))
            (setq mean (if (> number-pixels 1)
                           (/ sum-v (1- numpixs))
                           average))
            (setq sd-v-2 (- (/ sum-vv numpixs)
                            (* average average)))))

      (if (>= sd-v-2 0.0)
          (setq stan-dev (sqrt sd-v-2))
          (vis-error "Standard deviation is zero!"
                     :error-level :warning))

      (if (/= sum-v 0.0)
          (progn
            (setq row (/ sum-row-v sum-v))
            (setq col (/ sum-column-v sum-v)))
          (vis-error "Invalid center of gravity!"
                     :error-level :warning))

      (setf (plane-association input-pl :maxval) maxval)
      (setf (plane-association input-pl :minval) minval)
      (setf (plane-association input-pl :mean) mean)
      (setf (plane-association input-pl :average) average)
      (setf (plane-association input-pl :standard-deviation)
            stan-dev)
      (setf (plane-association input-pl :center-of-gravity)
            (list row col))

      input-pl
      )))


(defiop stats-plane (input-pl)                    ; Note 10

"..."

"..."
```

```
(let
  ((maxval (random 1.0)) (minval (random 1.0)) (row 0) (col 0)
   (number-pixels 0) (sum-row-v (random 1.0))
   (sum-column-v (random 1.0))
   (sum-v (random 1.0)) (sum-vv (random 1.0))
   (init 0) (abc 0.0))

  (declare (fixnum number-pixels row col init)
           (single-float maxval minval sum-row-v
            sum-column-v sum-vv abc))

  (call-i-section "STATS_PLANE"
                  "STATS_PLANE_I_SECT"
                  "int init"
                  "float *abc")

  (call-p-section "STATS_PLANE"
                  "STATS_PLANE_P_SECT"
                  (list input-pl)
                  NIL)
  (call-t-section "STATS_PLANE"
                  "STATS_PLANE_T_SECT"
                  "float *sum-v"
                  "float *sum-row-v"
                  "float *sum-column-v"
                  "float *sum-vv"
                  "float *maxval"
                  "float *minval")
  (let
    ((mean 0.0) (average 0.0)
     (sd-v-2 0.0) (stan-dev 0.0)
     (numpixs (coerce number-pixels 'single-float)))

    (declare (single-float mean average
              sd-v-2 numpixs stan-dev))

    (if (= number-pixels 0)
        (vis-error "Number of pixels sampled was zero!"
                   :error-level :warning)
        (progn
          (setq average (/ sum-v numpixs))
          (setq mean (if (> number-pixels 1)
                         (/ sum-v (1- numpixs))
                         average))
```

```
          (setq sd-v-2 (- (/ sum-vv numpixs)
                          (* average average)))))

      (if (>= sd-v-2 0.0)
          (setq stan-dev (sqrt sd-v-2))
          (vis-error "Standard deviation is zero!"
                     :error-level :warning))

      (if (/= sum-v 0.0)
          (progn
            (setq row (/ sum-row-v sum-v))
            (setq col (/ sum-column-v sum-v)))
          (vis-error "Invalid center of gravity!"
                     :error-level :warning))

      (setf (plane-association input-pl :maxval) maxval)
      (setf (plane-association input-pl :minval) minval)
      (setf (plane-association input-pl :mean) mean)
      (setf (plane-association input-pl :average) average)
      (setf (plane-association input-pl :standard-deviation)
            stan-dev)
      (setf (plane-association input-pl :center-of-gravity)
            (list row col))

      input-pl
      )))
```

C Code for STATS-PLANE

```
                                                    /* Note 11 */
#include   "llvs:llvs_per_pixel.h"

#define plane (*llvs_gpdf[0])(0)

static int number_pixels;
static float sum_v;
static float sum_row_v;
static float sum_column_v;
static float sum_vv;
static float max_v;
static float min_v;
                                                    /* Note 12 */
stats_plane_i_sect(init,abc)
int init;
```

```
float *abc;
{
printf("initial-pixels = %D, abc = %F",init,*abc);

number_pixels = 0;
sum_v = 0.0;
sum_row_v = 0.0;
sum_column_v = 0.0;
sum_vv = 0.0;

}
                                                    /* Note 13 */
stats_plane_p_sect()
{
  float val;

val = plane;

sum_v += val;
            /* The cast is necessary! */
sum_row_v += ((float)llvs_usercommon.current_row) * val;
sum_column_v += ((float)llvs_usercommon.current_column) * val;
sum_vv += val * val;
                                                     /* Note 14 */
if (number_pixels == 0) {
 max_v = val;
 min_v = val;
 }

else if (val > max_v) max_v = val;

else if (val < min_v) min_v = val;

number_pixels++;

}
                                                    /* Note 15 */
stats_plane_t_sect(numpixs, sumv, sumrowv, sumcolv,
                   sumvv, maxv, minv)
int *numpixs;
float *sumv, *sumrowv, *sumcolv, *sumvv, *maxv, *minv;
{

*numpixs = number_pixels;
```

```
*sumv = sum_v;
*sumrowv = sum_row_v;
*sumcolv = sum_column_v;
*sumvv = sum_vv;
*maxv = max_v;
*minv = min_v;

}
```

6 Standard Image Operators

This section describes the standard image operators provided as part of LLVS. Most image operators allow the user to specify a standard set of parameters. These standard arguments are always CommonLisp &key argumemnts. If they are allowed, they are listed in the form for the operator. All other arguments required or allowed by an operator are described with the operator. Unless otherwise noted, the result of an operator is the resultant *plane*.

6.1 average-plane

This operator produces a result *plane* by averaging the pixels in the *window area* of the input *plane*.

```
(average-plane plane &key window
                          limits
                          mask-plane
                          mask-value
                          bounds-action)
```

- plane

  This is the input *plane*.

- window

  The *window* determines the *window area* and positioning around the *current reference* in the *plane*. A *window* of (0 0) specifies a window area of one pixel in the input *plane*. A *window* of (0 1) species a *window area* of four pixels with the *current reference* in the input *plane* being in the upper left hand corner of the window. A *window* of (-1 1) specifies a *window area* of nine pixels with the *current reference* being in the center and so on. If the *window* is not given it is calculated by the formula
  $(02^{\max(0,il-rl)} - 1)$
  where *il* is the *level* of the input *plane* and *rl* is the *level* of the result *plane*.

6.2 convert-plane

This operator converts a *plane* to a new *plane* of the specified type. The new *plane* is allocated by this operator and is returned as the value of the operator. This conversion may result in loss of significance when converting from a more general type to a less general type (i.e. :INT to :BYTE).

Convert-plane is able to convert planes whose pixel values are represented by standard Lisp numeric types (i.e. FIXNUM, INTEGER, etc) into *planes* that can be passed to image operators written in C. Convert-plane can also convert

to a *plane* whose pixels are represented by the Lisp type T. Conversion of *planes* containing non-numeric pixels will result in an error report.

```
(convert-plane plane type &key  limits
                                mask-plane
                                mask-value
                                bounds-action)
```

- plane

  This is the input *plane*.

- type

  This is the result type and must satisfy `numeric-plane-type-p` or be T. Allowed values are :BIT, :BYTE, :SHORT, :INT, :FLOAT, and T.

6.3 convolve-plane

This operator *convolves* an input *plane* using a rectangular weighting array. The result is the convolved *plane* which is allocated by the operator. All arithmetic operations are done using floating point operations.

```
(convolve-plane plane &key weighted-array
                           limits
                           mask-plane
                           mask-value
                           bounds-action)
```

- plane

  This is the input *plane*.

- weighted-array

  This must be a two dimensional *based array* containing numeric values. The size of this array determines the *window* in the input *plane* accessed for each result pixel. This *window* is placed such that its (0 0) element coincides with the *current reference*. Each element of the weighting array is multiplied by the value of the pixel in the corresponding position of the *window*. The result pixel is the sum of these products.

  The convolve operator in the old Vision System allowed more parameters than this operator allows. In particular a divisor could be specified as well as the relative position of the weighting array to the input pixel window. The divisor can be applied to the weighting array before it is given to the operator. Non-symetric weighting arrays can be handled by using larger symetric weighting arrays.

6.4 create-old-image

This function creates a file containing a *plane* in the format used by the old Visions system. A warning message is displayed if the *plane* being saved is not square with the length of each side equal to 2^{level}. The *plname* and *pldescr* fields of the created old image are obtained either from arguments to the operator or from the association list of the *plane* using the keys :name and :description. The *location, background-value,* and *associations* of the *plane* are not saved.

`(create-old-image plane pathname &key plane-name plane-description)`

- plane This is the *plane* that will be written to the file.
- pathname

  This is a string specifying the file to be created to hold the old image.
- plane-name

  This is a string that will be written to the name field of the image file.
- plane-description

  This is a string that will be written to the description field of the image file.

6.5 extract-masked-plane

This operator determines the extent of a region in a *plane* and returns a new *plane* containing that region. This operator is a CommonLisp *multi-valued* function that returns three results[25]. The primary result is a *plane* just large enough to contain the region. The secondary result is a *plane* of *type* :BIT that contains the extent of the region. The third result is an *alist* that may be used as the *limits* argument for further processing. The limits bound the extracted region.

This operator is designed to allow the user to extract a region of an image for more intensive processing under the control of the secondary result used as a *mask-plane*. *Extract-masked-plane* makes two passes over the original input *plane*. The first plass determines the maximum and minimum row and column numbers of the rectangle that contains the region. The second pass extracts that rectangle and constructs the :BIT *plane*.

Using this operator without the *mask-plane* argument duplicates the input *plane* in an expensive way.

```
(extract-masked-plane plane &key (margin 0)
                                 limits
                                 mask-plane
```

[25]The user of the operator should use the CommonLisp MULTIPLE-VALUE-SETQ form if both results are desired.

```
                                mask-value
                                bounds-action)
```

- plane

  This is the original *plane* from which the new plane will be obtained.

- margin

  For many image operators that work on regions to generate valid results, requires that they be able to access pixels outside the region (i.e. convolve-plane with a mask). By specifying a margin greater than zero, extract-masked-plane will extract that many extra rows and columns on each side of the extracted rectangle containing the region.

6.6 format-plane & format-plane-to-file

These two operators convert a plane to a character string representation. The format used to convert the numeric values is specified using the CommonLisp FORMAT function control string. FORMAT-PLANE sends the characters to the specified CommonLisp *stream* while FORMAT-PLANE-TO-FILE sends the string to the specified file.

```
(format-plane stream
      format-string
      plane
      limits
        mask-plane
      mask-value)

(format-plane-to-file filepath
                      format-string
      plane
      limits
      mask-plane
      mask-value)
```

6.7 gaussian-smooth-plane

This operator produces a result *plane* by doing a gaussian smoothing of the pixels in the *window area* of the input *plane*. The result pixels are determined by convolving the pixels in *window* of the input *plane* using a mask containing a gaussian distribution.

```
(gaussian-smooth-plane plane &key radius
                                  st-deviation
                                  limits
```

```
                                mask-plane
                                mask-value
                                bounds-action)
```

- plane

  This is the input *plane*.

- radius

  A square array containing the gaussian distribution is constructed. The size of the array depends on the *radius* as in $(2 \times radius) - 1$. The array is normalized to conserve energy under the kernel.

- st-deviation

  This is the standard deviation used to generate the gaussian distribution. If it is not supplied it is generated from the *radius*. The larger this value is the less filtering effect this image operator has.

6.8 get-old-image

This function creates a new *plane* and initializes the pixels to the pixel values found in an image file created by the old Visions System. The level of the new plane is computed from the size of the old image by the formula

$$level = \log_2 \max(rowsize, columnsize)$$

where *rowsize* and *columnsize* are values found in the image file. The *plname* and *pldescr* fields of the old image are placed in the new *plane's* association list with the keys `:name` and `:description`. This name and description are also displayed to the default output stream. The *location* of the new *plane* is (0 0) and its *type* is obtained from the image file. The result of the function is the new *plane*.

```
(get-old-image pathname)
```

- pathname

  This is a string specifying the file containing the old image.

6.9 histogram-plane

This operator generates a one dimensional histogram of the input *plane* and returns it as a *based-array* of one dimension.

```
(histogram-plane plane &key minval
                             maxval
```

```
                    (scale 1)
                    limits
                    mask-plane
                    mask-value
                    bounds-action)
```

- minval

  This is the minimum value and becomes the base of the result *based vector*. Pixels whose value is less than this value are ignored. This value is obtained from the *plane's* association list or defaults to 0 if not specified.

- maxval

  This is the maximum value and becomes the limit of the result *based vector*. Pixels whose value is greater than this value are ignored. This value is obtained from the *plane's* association list or defaults to 255 if not specified.

- scale

  Every pixel is divided by this value before being truncated to an integer index into the histogram vector.

6.10 histogram-2d-plane

This operator generates a two dimensional histogram of the input *planes* and returns it as a two dimensional *based-array*.

```
(histogram-2d plane1 plane2 &key range1
                                 range2
                                 limits
                                 mask-plane
                                 mask-value
                                 bounds-action)
```

- range1

  This is a three element vector containing *minval*, *maxval*, and *scale* values for *plane1* as defined for the *histogram-plane* operator.

- range2

  This is a three element vector containing *minval*, *maxval*, and *scale* values for *plane2* as defined for the *histogram-plane* operator.

6.11 linear-map-planes

This operator maps several input *planes* to an output *plane* using the formula
$result_{r,c} = (offset + \sum_{k=1}^{N} coeff_k * input^k_{r,c})/divisor$
where N is the number of input *planes*. All arithmetic operations are done in floating point and the result converted if necessary.

```
(linear-map-planes planes &key coeff
                                (offset 0)
                                 divisor
                                (type :float)
                                 limits
                                 mask-plane
                                 mask-value
                                 bounds-action)
```

- planes

  This must be a *plane* or a list of *planes*.

- coeff

  This must be a Lisp sequence of numbers. An error is generated if the sequence is shorter than the number of *planes*. If it is not supplied, a sequence of 1s is used.

- offset

  This is the offset value used in the mapping. It defaults to zero.

- divisor

  This is the divisor used in the mapping. It defaults to the number of input planes.

- type

  This is the result *plane* type and must satisfy `numeric-plane-type-p`.

6.12 linear-project-plane

This operator increases the *level* of the input *plane* to the *level* specified. The result pixels are determined by a linear interpolation function applied to the pixels around the *current referent* in the input *plane* to produce each of the pixels at the corresponding location of the output *plane*. If the level does not change, this operator just copies the *plane*.

The interpolation uses a linear interpolation from the neighborhood of the input pixels. The interpolation is a weighted sum based on the distance to the *ancesters* of the new pixel.

```
(linear-project-plane plane &key limits
                                 mask-plane
                                 mask-value
                                 bounds-action)
```

- plane

  This is the input *plane*.

6.13 maximum-plane

This operator produces a result *plane* by using the maximum pixel value in the *window area* of the input *plane*.

```
(maximum-plane plane &key window
                          limits
                          mask-plane
                          mask-value
                          bounds-action)
```

- plane

  This is the input *plane*.

- window

  The *window* determines the *window area* and positioning around the *current reference* in the *plane*. A *window* of (0 0) specifies a window area of one pixel in the input *plane*. A *window* of (0 1) species a *window area* of four pixels with the *current reference* in the input *plane* being in the upper left hand corner of the window. A *window* of (-1 1) specifies a *window area* of nine pixels with the *current reference* being in the center and so on. If the *window* is not given it is calculated by the formula
  $(02^{\max(0,il-rl)} - 1)$
  where *il* is the *level* of the input *plane* and *rl* is the *level* of the result *plane*.

6.14 median-plane

This operator produces a result *plane* by using the median pixel value in the *window area* of the input *plane*.

```
(median-plane plane &key window
                         limits
                         mask-plane
                         mask-value
                         bounds-action)
```

- plane

  This is the input *plane*.

- window

  The *window* determines the *window area* and positioning around the *current reference* in the *plane*. A *window* of (0 0) specifies a window area of one pixel in the input *plane*. A *window* of (0 1) species a *window area* of four pixels with the *current reference* in the input *plane* being in the upper left hand corner of the window. A *window* of (-1 1) specifies a *window area* of nine pixels with the *current reference* being in the center and so on. If the *window* is not given it is calculated by the formula
  $(02^{\max(0,il-rl)} - 1)$
  where *il* is the *level* of the input *plane* and *rl* is the *level* of the result *plane*.

6.15 merge-planes

This function constructs a new *plane* from a list of *planes*. The *size*, *level*, and *location* of the new *plane* are determined in the same way `union-plane-limits` determines its result. The precedence of each *plane* in the merge is determined by its order in the list of planes argument. When several *planes* have a value for a position in the *conceptual plane* of the result, the value is taken from the one with the highest precedence value which is the first one occurring in the list. If no *plane* has a pixel in that position, the *background-value* of the first *plane* in the list is used.

```
(merge-planes planes &key limits
                          mask-plane
                          mask-value)
```

- planes

  This is a list of *planes*.

6.16 minimum-plane

This operator produces a result *plane* by using the minimum pixel value in the *window area* of the input *plane*.

```
(minimum-plane plane &key window
                          limits
                          mask-plane
                          mask-value
                          bounds-action)
```

- plane

  This is the input *plane*.

- window

  The *window* determines the *window area* and positioning around the *current reference* in the *plane*. A *window* of (0 0) specifies a window area of one pixel in the input *plane*. A *window* of (0 1) species a *window area* of four pixels with the *current reference* in the input *plane* being in the upper left hand corner of the window. A *window* of (-1 1) specifies a *window area* of nine pixels with the *current reference* being in the center and so on. If the *window* is not given it is calculated by the formula
  $(02^{\max(0,il-rl)} - 1)$
  where *il* is the *level* of the input *plane* and *rl* is the *level* of the result *plane*.

6.17 random-sample-plane

This operator generates a new *plane* that consists of pixels obtained from the input *plane*. The result pixels are determined by generating a random number that is used to obtain one pixel in *window area* of the input *plane*.

```
(random-sample-plane plane &key window
                                 limits
                                 mask-plane
                                 mask-value
                                 bounds-action)
```

- plane

  This is the input *plane*.

- window

  The *window* determines the *window area* and positioning around the *current reference* in the *plane*. A *window* of (0 0) specifies a window area of one pixel in the input *plane*. A *window* of (0 1) species a *window area* of four pixels with the *current reference* in the input *plane* being in the upper left hand corner of the window. A *window* of (-1 1) specifies a *window area* of nine pixels with the *current reference* being in the center and so on. If the *window* is not given it is calculated by the formula
  $(02^{\max(0,il-rl)} - 1)$
  where *il* is the *level* of the input *plane* and *rl* is the *level* of the result *plane*.

6.18 read-plane

This operator reads a *plane* in from a file and returns it. The *plane* is placed in the file by the `write-plane` operator. If no :NAME value pair is on the *plane's* association list, the file path name is added.

```
(read-plane filepath)
```

- filepath

  This argument should be a string containing all the information necessary to find the file containing the *plane*.

6.19 region-limits-plane

This operator determines the extent of a region in a *plane* and returns an association list that can be used as the *limits* argument to further image operators to restrict processing to some rectangle of the *plane*.

The operator determines the maximum and minimum row and column numbers of the rectangle that contains the region and returns the values in an association list containing the pairs for :START-ROW, :END-ROW, :START-COL and :END-COL.

Using this operator without the *mask-plane* argument is an expensive way to generate the *limits* for the entire original input *plane*.

```
(region-limits-plane plane &key limits
                                mask-plane
                                mask-value
                                bounds-action)
```

- plane

  This is the original *plane* for which the *limits* are derived.

6.20 region-extents-plane

This operator determines the extent of a region in a *mask-plane* and returns a result *plane* that contains the extent of the region specified as pixel values of 0 and 1.

This operator is designed to allow the user to perform further processing of a *plane* that is restricted to a region of an image by constructing a new *plane* that can be used as a *mask-plane* in that processing.

Using this operator without the *mask-plane* argument is an expensive way to generate a *plane* of all ones.

```
(region-extents-plane &key limits
                           mask-plane
                           mask-value
                           bounds-action)
```

6.21 rigid-transform-plane

This operator will *rotate* an input *plane* to produce a result output *plane*. This operator allocates the result *plane*. The *size* and *location* of the result *plane* is calculated based on the rotation. This operator will also do a translation.

```
(rigid-transform-plane plane &key (rotation 0)
                                  center
                                 (trans '(0 0))
                                  limits
                                  mask-plane
                                       mask-value
                                       bounds-action)
```

- plane

  This is the input *plane* of the operation.

- rotation

  This argument specifies the amount of rotation in degrees. Negative values result in a clockwise rotation while positive values result in a counter-clockwise rotation.

- center

  This argument specifies the center of the rotation and must be a two element list whose first value is the row position and whose second is the column position. If this argument is not specified the *plane* is rotated about the absolute location (0 0).

- trans

  This is the (row,column) amount of the translation and may be non-integer values.

6.22 show-plane, show-plane-edge, and show-plane-vector

These operators display a *plane* on the device specified. The functions return the *plane* given as the argument. As a by-product they call STATS-PLANE if the association list of the input *plane* does not contain :MINVAL and :MAXVAL. The *plane* is converted in an appropriate way for the display.

```
(show-plane plane &key device
                       limits
                       mask-plane
                       mask-value)
```

```
(show-plane-edge horizontal-plane
       &optional vertical-plane
            &key device
                 limits
                 mask-plane
                 mask-value)

(show-plane-vector radius-or-row-plane
   angle-or-column-plane
      &key device
   coordinate-type
                    coding-method
   vector-length
   length-clipping
   tail-location
   tail-size
   arrow-head
                    limits
                    mask-plane
                    mask-value)
```

- plane

  This is the *plane* to be displayed.

- device

  This argument specifies where the *plane* is to be displayed. This argument should be an association list or one of the following a symbol specifying a device. If the argument is an association list the association *(device value)*, where *value* specifies the device to be used. Other values in the list are any device dependent parameters required. Among these might be the plane of the device used for the display or the minumum and maximum intensity levels to be displayed.

  The particular values allowed or required by each device are documented in "Show-Plane and Friends User's Manual" by Robet Heller.

  It is intended that the user will develop a set of these lists for their own use. Show-plane always deposits the association list it last used in *SHOW-PLANE-DESCRIPTOR* whether this list was the input argument or generated through interaction with the user. The user may capture this list and save it for future use.

6.23 standard-sample-plane

This operator generates a result *plane* by the upper left-hand corner sampling method. The result pixels are determined by using the pixel in the upper left

hand corner of the *window area* in the input *plane*.

```
(standard-sample-plane plane &key window
                                  limits
                                  mask-plane
                                  mask-value
                                  bounds-action)
```

- plane

  This is the input *plane*.

- window

  The *window* determines the *window area* and positioning around the *current reference* in the *plane*. A *window* of (0 0) specifies a window area of one pixel in the input *plane*. A *window* of (0 1) species a *window area* of four pixels with the *current reference* in the input *plane* being in the upper left hand corner of the window. A *window* of (-1 1) specifies a *window area* of nine pixels with the *current reference* being in the center and so on. If the *window* is not given it is calculated by the formula
  $(0 2^{\max(0,il-rl)} - 1)$
  where *il* is the *level* of the input *plane* and *rl* is the *level* of the result *plane*.

6.24 stats-plane

This operator obtains the following values from a *plane* and places them on the *plane's* association list. The input *plane* is returned as the result.

- :MAXVAL — maximum pixel value.
- :MINVAL — minimum pixel value.
- :MEAN — mean pixel value.
- :AVERAGE — average pixel value.
- :STANDARD-DEVIATION — the standard deviation of the pixels.
- :CENTER-OF-GRAVITY — the location of the center-of-gravity.

It is recommended that users avoid using the &key arguments as they will result in statistics that are valid only for a subset of the pixels in the *plane*.

```
(stats-plane plane &key limits
                        mask-plane
                        mask-value
                        bounds-action)
```

- plane

  This is the *plane* from which the statistics are obtained. These statistics are placed at the beginning of the *plane's* association list or replace the values that already are there.

6.25 translate-plane

This operator translates each pixel from the input *plane* to a pixel of an output *plane*. The translation is done via table-lookup. The table is a vector or matrix. A vector is treated as a 1 by *n* matrix. The number of rows determines the number of output *planes*. The result of the operator is a list of output *planes* which are all allocated by the operator.

Each input pixel is converted to an integer and used as a column index. The values in each row of that column are placed in the corresponding output *planes*. The type of the matrix determines the type of the output *planes*.

```
(translate-plane plane lookup-table &key (min 0)
                                         (max 255)
                                         limits
                                         mask-plane
                                         mask-value
                                         bounds-action)
```

- plane

  This is the input *plane*.

- lookup-table

  This is the table used to map values from the input *plane* to the output *planes* and is represented as a *based-array*.

- min

  For any value from the input *plane* below the column range of the *lookup-table*, this value is placed in the output *planes*. If this value is not supplied and an out-of-range value is encountered, an error is generated.

- max

  For any value from the input *plane* above the column range of the *lookup-table*, this value is placed in the output *planes*. If this value is not supplied and an out-of-range value is encountered, an error is generated.

6.26 write-plane

This operator writes a *plane* to a file. Included with the actual image data on the file is the *plane's level*, *location*, *background-value*, and *association list*. This file may be read with the `read-plane` operator.

`(write-plane plane filepath)`

- plane

  This is the *plane* to be placed in the file.

- filepath

  This should be a string containing all of the information needed to create the file. If the file already exists, a new version of the file is created.

A Some Lisp Forms

The forms described in this appendix are useful for both using and writing image operators.

A.1 array-type-from-plane

This function returns a CommonLisp type that can be used to represent the pixels in a plane as elements in a CommonLisp array.

`(array-type-from-plane plane)`

A.2 build-limits-c

This function builds the LIMITS_C structure containing the limits to pass to image operators using the *per-plane* method. An error is generated if the :START-ROW, :END-ROW, :START-COL, :END-COL and :LEVEL values are not given. If :DELTA-ROW or :DELTA-COL are not specified, they are set to 1. See the C Support section of this reference manual for a description of the structure.

`(build-limits-c limits)`

- limits — the *limits* as constructed by `union-plane-limits`, etc.

A.3 build-mask-values-c

This function builds the MASK_VALUES_C structure for an image operator using the *per-plane* method. It returns the MASK_VALUES_C structure in a form useable by the C routine — see the C Support section of this reference manual. The mask values are always passed as integers.

`(build-mask-alues-c mask-value)`

- mask-value - the mask values.

A.4 build-plane-info-c

This function prepares a structure containing information about a *plane* for use by the image operators written using the *per-plane* method. This information includes the *plane's level* and *location*. See the C Support section of this reference manual.

`(build-plane-info-vector-c planes)`

- planes - a list of the planes required by the *per-plane* image operator.

A.5 build-plane-info-vector-c

This function prepares the *LLVS-PLANE-INFO-VECTOR* special variable for use by the image operators written using the *per-plane* method. See the C Support section of this reference manual.

The *LLVS-PLANE-INFO-VECTOR* variable holds the information. It is allocated statically so that it does not move during garbage collects. It gets passed to the image operators written in C using the *per-plane* method. A vector of pointers is used so that the some internal LLVS Lisp functions can still access the fields of each individual structure.

`(build-plane-info-vector-c planes)`

- planes - a list of the planes required by the *per-plane* image operator.

A.6 construct-plane

This function constructs a new *plane* of the appropriate *level*, *size*, and *location* for use in an operation. The *background-value* defaults to 0. The result is the new *plane*.

This function is capable of constructing a *plane* whose elements can be any valid Lisp value. *Planes* whose elements are not the standard image elements can not be used with most of the image operators.

`(construct-plane type limits)`

- type

  This is the *type* of the new *plane* and should satisfy *plane-type-p*.

- limits

  This is the association list returned by `union-plane-limits` or `intersect-plane-limits` or given by the user which describes the extent of the *plane* to be constructed.

A.7 c-per-pixel-driver

This routine calls the *driver* routine written in C that calls the P section for each pixel of the *conceptual plane* of the operation. This routine prepares the addresses and linkages needed for the C routines. The result is the list of output *planes* suppiled as an argument.

The *conceptual plane* of operation is figured in the same way as `construct-plane` determines the attributes of the *plane* it constructs. All the *planes* supplied as arguments, excluding the mask plane, are used in this determination. The *size*

and *location* of the *conceptual plane* determines the looping and indexing control used.

The P function of the image operator accesses each *plane* by doing an indirect function call through a set of C variables containing addresses. Each variable contains the address of the *access function* for the corresponding *plane*. These variables are set up by the *driver* which uses the types of the actual *planes* passed in as arguments to determine which access function addresses to place in each variable.

```
(c-per-pixel-driver p-section
                    input-planes
                    output-planes
                  &key mask-plane
                       (mask-value 1)
                       (bounds-action :error)
                       limits)
```

- p-section

  The argument is a *p-section* descriptor used to obtain the address of the image operator's P function.

- input-planes

  This is the list of *planes* needed as input for the operation. The *driver* sets up two sets of access functions for each *plane*. One set of access functions converts each pixel from the *plane* to C `int` type while the other set converts to C `float` type.

  Planes are referenced by number in the C code. This number is determined by the order of the *planes* as passed in this list argument. The numbering starts at zero.

- output-planes

  This is the list of *planes* needed as output for the operation. The *driver* sets up two sets of access functions for each *plane*. One set of access functions converts C `int` type values to the pixel type in the *plane* while the other set converts from C `float` type. *Since the C compiler does not check to make sure that the type of pixel value passed in to the access function matches the type expected by the access function, the user should be careful to make sure they match or un-expected result pixel values will occur.*

  Planes are referenced by number in the C code. This number is determined by the order of the *planes* as passed in this list argument. The numbering

starts at zero. The numbering is separate for the input and output *planes*. Thus, it is not possible to accidentally set a pixel in an input *plane* or get a pixel from an output *plane*.

- mask-plane

  This is the mask-plane to be used in the operation. If it is not NIL, then the *driver* compares each pixel from this *plane* with each the value of mask-value. If the pixel matches any value, then the P function is called for that position in the output *plane*. If they are not equal that pixel position is skipped. If mask-plane is NIL, every pixel position is used. All *mask-plane* pixels and values of the *mask-value* argument are converted to integer (using truncation) for the comparison.

- mask-value

  This must be a single value or a sequence.

- bounds-action

  This parameter specifies how out-of-bounds references[26] to the *planes* are to be treated. The value must be one of the following symbols.

  - :ERROR

    An out-of-bounds reference will generate an error.

  - :BACKGROUND-VALUE

    The *background-value* of the appropriate *plane* is used on a reference while a set is ignored.

  - :NEAREST-VALUE

    The nearest existing pixel in the *plane* is used on a reference while a set is ignored.

- limits

  The indexing over the conceptual plane can be modified by specifying a begining row and column, an ending row and column, and a row and column stride. The begining row (column) must be less than or equal to the ending row (column). All these values are optional and if not specified, the value chosen is that which will maximize the number of pixels in the *conceptual plane* that are indexed. If the *limits* argument is NIL, the values used are chosen this way as well. The *level* is included here as the values have no meaning independent of the *level* at which the operation is conducted.

[26] An out-of-bounds reference is a reference to a non-existent pixel. Since the *conceptual plane* being used may extend past the boundary of any particular *plane*, it is necessary to provide some action in this case.

The *limits* argument specifies any restrictions on the indexes over the *conceptual plane*. The values are given as an association list where the first element of each pair is the item as specified below and the second element is the value.

- :LEVEL
  This is the *run level* at which the image operator is to work. If this value is not supplied the value used is the maximum level of all the *planes* passed to the *driver* via the *planes* argument.
- :START-ROW
  Specifies the beginning row indexed.
- :END-ROW
  Specifies the end row indexed.
- :DELTA-ROW
  Specifies the step size added to the row index.
- :START-COL
  Specifies the beginning column indexed.
- :END-COL
  Specifies the last column indexed.
- :DELTA-COL
  Specifies the step size added to the column index.

A.8 define-p-section

When the *per-pixel* method is used the `c-per-pixel-driver` needs to obtain the address of the P function of the image operator. This macro returns a DEFVAR form which defines a symbol and binds it to the information needed to find the address. The binding is saved so that when a suspended file of the LLVS system is resumed, the linkages are re-created. As a side affect, DEFINE-P-SECTION appends a CommonLisp SETF form to the global variable *LLVS-P-SECTION-LIST*. This *form* is used to recompute the address of a P section after a suspended Lisp file is reloaded using the form

```
(mapcar \#'eval *llvs-p-section-list*)
```

You do not have to include any DEFINE-P-SECTION forms if you have used the CALL-P-SECTION form in conjunction with DEFINE-OPERATOR or DEFIOP.

```
(define-p-section name image-name entry-point)
```

- name

  This is used as the name of the CommonLisp variable that is created as in `(defvar name ...)`.

- image-name

  This is a CommonLisp string which is used to obtain the sharable image containing the P section.

- entry-point

  This is a CommonLisp string which is the name of the entry point of the P section procedure in the image.

A.9 describe-plane

This function displays in an easy to read way all the information about a *plane*. This information is kept in the *plane's* descriptor including the *type*, *size*, *location*, *level*, *background-value* and the values on the association list. The result is the input *plane*.

`(describe-plane plane)`

- plane — This specifies the *plane* to be described.

A.10 find-all-planes

This function returns a list of all the variables in the specified package that are bound to a *plane*.

`(find-all-planes (&optional (package 'user))`

- package

  This specifies the CommonLisp *package* which will be searched to find the symbols bound to *planes*. The default value for this argument when it is not supplied is the user's package.

A.11 get-based-array-params

`Get-based-array` provides a convenient interface to `get-parameter` for the operation of obtaining a *based-array* as a parameter to an image operator.

This routine takes either a Lisp array or a *based-array* and returns, as a CommonLisp multiple-value, the floating-point version of the array specified and the offsets.

In LLVS, anywhere a based-array is used a Lisp array may also be used. This function is provided in order to make using *based-arrays* as easy as possibe. The

routine returns two structures. One is a vector representing the *offsets* field of a based-array, and the other is the array representing containing the array values. The routine also converts the values of the array to `single-float`. If the argument to `get-based-array-params` is an ordinary array, the offsets used will be 0.

`(get-based-array-params name an-array &key ask)`

- name

  This is the name passed to the `get-parameter` call.

- an-array

  This is the value passed as the :ARG-VALUE argument of the `get-parameter` call. It should be NIL, an array, or a *based-array*.

- ask

  This is the value passed as the :ASK argument of the `get-parameter` call. It should be T or NIL.

A.12 get-parameter

This function obtains the value for a parameter needed by the image operator. All image operators should use the `get-parameter` mechanism to obtain needed parameters so that the interface to the end user is standardized.

`Get-parameter` trys several different ways to obtain the value needed. The first value obtained is returned as the result. If no value is found, the *default* value is returned[27].

The order in which values are searched for is:

1. arg-value — if not NIL.
2. associations — if not NIL.
3. package — if *LLVS-USE-PACKAGE* is not NIL and the symbol is bound in the package referenced by *LLVS-USE-PACKAGE* and it's value is not :NO-VALUE.
4. ask the user — if ASK is not NIL.
5. default — if the other methods fail.

[27] In most cases NIL will be used to specify a default action to some other function. If the image operator that is calling `get-parameter` needs to make sure a value is supplied it can make :NO-VALUE the default value in the *iop-parameter* and check for it being returned.

```
(get-parameter name parameter &key (verify-default NIL)
                                   (arg-value NIL)
                                   (associations NIL)
                                   (prompt "~&~A? ")
                                   (help "No help supplied.")
                                   (allowed-values t)
                                   (display nil)
                                   (default nil)
                                   (ask NIL))
```

- name

  This is the name supplied for use by the prompting logic — see *prompt* below. It is also the name used to access variables in a *package* or on the association list.

- parameter

  This argument is an *iop-parameter* structure used to describe arguments to the image operators. Since image operators tend to have many of the same arguments and the handling of these arguments is always the same, this structure provides a convenient way of describing those arguments in one place[28]. The fields in this structure are used by *get-parameter* as follows:

  - allowed-values

    This should be a CommonLisp type specifier that is *T* if the argument is an allowed value for this parameter. Any value obtained must satisfy this type or an error is generated.

    For ease of use, integers values are automatically converted to **float** when the :ALLOWED-VALUES specifies a subtype of **float**. Integral float values are automatically converted to integers as well.

  - default

    This is the default value supplied when the value is not obtained elsewhere. NIL is allowed as a default value. If there is no useful default and a value *must* be supplied, the slot in the *iop-parameter* structure should be set to :NO-VALUE. In this case, :NO-VALUE will be returned as the value of **get-parameter** and can be checked by the image operator. If the default value is used, it is checked unless the value is NIL or :NO-VALUE — these are returned unchecked.

  - display

    This is NIL or the symbol of a function of one argument like the CommonLisp DESCRIBE function. This function is used to display

[28] See the Appendix – Standard Parameter Definitions for a list of pre-defined *iop-parameters*.

the default value, or any other value obtained, in any interactive dialogue with the user. If *display* is NIL, the CommonLisp DESCRIBE function is used.

- prompt

  This is the prompt sent to the user at the terminal when the parameter has not been found by the previous means. The user must then enter a value in response. The **get-parameter** routine will check whether the process is running from the batch queue or interactively. For non-interactive use, **get-parameter** will not request the value from the user. In this case if a default value is not supplied, the process will be aborted with an error.

  The *prompt* should be a string in the form acceptable by the CommonLisp FORMAT function and can have one value parameter designator. The value supplied is the *name* given to **get-parameter** as in

```
(format *query-io* (iop-parameter-prompt llvs-plane-def) name)
```

- help

  This is a description of the parameter and the allowed values for it. This information is presented to the user if they enter a request for help in response to a prompt for the parameter sent by **get-parameter**. The *help* should be a string in the form acceptable by the CommonLisp FORMAT function and can have several value parameter designators. The value supplied is the *name* given to **get-parameter** as in

```
(format *query-io* (iop-parameter-prompt llvs-plane-def)
                   name name name name ...)
```

All of these fields may be over-ridden by supplying the appropriate argument on the **get-parameter** call. Also, if the *parameter* argument is NIL, the values used will be the default values listed for those arguments.

- verify-default

  This value is ORed with the value of *VERIFY-DEFAULT*. If this result is not NIL and the supplied *default* will be used as the result of **get-parameter**, **get-parameter** will display the default value on the user's terminal and allow the user to specify if that value should be used. If the user specifies that the default is not to be used they must enter

```
(defstruct iop-parameter
  "defines each parameter for an image operator"
  (allowed-values nil) ;should be a type specifier
  (default nil)
  (display nil :type symbol)
  (prompt :type string)
  (help :type string))
```

Figure 6: Iop-parameter Structure

another value. This supplied value is then checked and re-displayed. The user is allowed to verify it again.

If the user's process is not interactive then *verify-default* has no effect.

- arg-value

  If this argument is not NIL, it is assumed to be the value required. This will generally be the value of an argument of the function calling `get-parameter`.

- associations

  This argument should be an association list (usually obtained from a *plane* involved in the operation). The *name* argument is used to obtain the value using *(assoc name associations)*. If this is not NIL, then the CDR of this result is returned.

- default

  A default value may be calculated and supplied via this argument. It will be used in place of the :DEFAULT value given in the IOP-PARAMETER structure if it is not NIL.

- prompt

  A prompt may be supplied via this argument. It will be used in place of the :PROMPT value given in the IOP-PARAMETER structure if it is not NIL.

- help

  Help text supplied via this argument. It will be used in place of the :HELP value given in the IOP-PARAMETER structure if it is not NIL.

- allowed-values

  A CommonLisp type specifier for checking the value obtained for a parameter may be supplied via this argument. It will be used in place of the

:ALLOWED-VALUES value given in the IOP-PARAMETER structure if it is not NIL.

- display

  A function used to display the value of the parameter may be supplied via this argument. It will be used in place of the :DISPLAY value given in the IOP-PARAMETER structure if it is not NIL.

- ask

  If this is not NIL and a value for the parameter has not been found by other means, the user will be prompted to supply the value.

A.13 get-pixel

While most image operators may be applied to *planes* containing numeric pixel data only, it is possible to construct *planes* where each pixel location contains a more complex structure such as a Lisp list. Very often the user will want to contruct such a *plane* that is geometrically isomorphic to some image data *plane*. This function allows the user to access elements of *planes* using the same coordinate system as used by the image operators[20]. The value of the function is the element of the *plane*. For elements that do not exist an error is generated.

`Get-pixel` may be used with the CommonLisp SETF form.

`(get-pixel level plane row column)`

- level

  This is the *primary level* used to transform the *row* and *column* parameters before accessing the *plane*. If it is NIL, the *level* of the argument *plane* is used.

- plane

  This is the plane from which an element is to be extracted.

- row

  This is the row coordinate at which to access the data and is relative to the absolute origin.

- column

  This is the column coordinate at which to access the data and is relative to the absolute origin.

[20] See the discussion of *level* and *location* in the Representation of Image Data section.

A.14 get-plane-limits

This function provides a convenient way of obtaining the constraints to place on the conceptual plane over which the image operators index. It is essentially a series of calls to **get-parameter** for each of the seven parameters using standard *iop-parameter* structures for each value. The value of the function is an association list acceptable to **c-per-pixel-driver**. The seven parameters are :LEVEL, :START-ROW, :END-ROW, :DELTA-ROW, :START-COL, :END-COL, and :DELTA-COL.

```
(get-plane-limits alist)
```

- alist

  This should be an association list usually given as an argument when the image operator was called. There are three cases:

  1. *alist* is NIL and *VERIFY-DEFAULT* is NIL.
     Nil is returned as the result.
  2. *alist* is NIL and *VERIFY-DEFAULT* is T.
     The user is asked for the *level* which the image operator is to use.
  3. *alist* is not NIL.
     For each parameter found in *alist*, **get-paramter** is called to obtain the value of the parameter. If the association list pair for that parameter has a non-NIL second element, that value is used as the :arg-value argument to **get-parameter**. The *verify-default* argument is NIL each time.
     The seven parameters are described in the discussion of the *limits* argument to the **c-per-pixel-driver**.

A.15 intersect-plane-limits and intersect-plane-limits*

These routines generate constraints that can be used for the value of the *limits* argument to most image operators. The limit constrains the processing to the area of intersection of the specified *planes*. The result of the function is an association list containing pairs for :START-ROW, :END-ROW, :START-COL, :END-COL and :LEVEL. The level is included here because the other values depend on the value for :LEVEL.

```
(intersect-plane-limits* limits &rest planes)

(intersect-plane-limits limits planes)
```

- limits

  This is the *limits* for the *conceptual plane* for which the constraints are to be determined. Values for :START-ROW, etc, in this association list have higher precedence than those obtained from the *planes* specified. If *limits* is NIL then the maximum of the *levels* of the argument *planes* is used.

- planes

  These are the *planes* for which the intersection is determined. The first form of the function allows the *planes* to be listed as in

```
(intersect-plane-limits* limits red green blue)
```

  while the second is used as in

```
(intersect-plane-limits limits (list red green blue))
```

A.16 llvs-fun

This macro generates a CommonLisp DEFUN form. It is used in place of DEFUN for functions provided by LLVS because it sets up additional help information. `llvs-fun` is used identically to DEFUN except that the *documentation-string* argument is not optional and an additional documentation string (*help*) is required. This *help* string is saved using the CommonLisp *documentation* form with the *doc-type* of *llvs-help* .

```
(llvs-fun name lambda-list documentation-string help forms)
```

A.17 llvs-help

This function stores textual information on the *plist* of the *symbol* specified. The information is placed on the *plist* using the CommonLisp function *documentation* with *doc-type* of *llvs-help* and *llvs-form*. This function is used to supply documentation for the structures, constants, etc defined by LLVS. The CommonLisp DESCRIBE function is used to obtain any available information about a symbol.

```
(llvs-help symbol help-string &optional form)
```

```
(defstruct (based-array :constructor NIL)
  "A based array."
  (offsets :type simple-vector)
  (values :type simple-array))
```

Figure 7: Based Array Structure

A.18 llvs-macro

This macro generates a CommonLisp DEFMACRO form. It is used in place of DEFMACRO for functions provided by LLVS because it sets up additional help information. `llvs-macro` is used identically to DEFMACRO except that the *doc-string* argument is not optional and an additional documentation string (*help*) is required. This *help* string is saved using the CommonLisp *documentation* form with the *doc-type* of *llvs-help* .

```
(llvs-macro name lambda-list doc-string help forms)
```

A.19 llvs-suspend

This function should be used to save a suspended Lisp session because it will automatically restore any addresses of external routines used when the suspended session is resumed.

```
(llvs-suspend (&optional (suspend-file "LLVS_USER.SUS")))
```

A.20 make-based-array

As an aid in the use of the LLVS the concept of a *based array* is provided. A *based array* differs from a CommonLisp array in that indexes are not necessarily 0-origined. This type of array is very useful for doing such things as histogramming pixel values from a *plane* where the range of pixel values may be from -5 to 5 for example. With a '*based vector* whose base is -5, the pixels can be used directly as an index into the vector.

A *based array* is implemented as a CommonLisp structure that contains both the CommonLisp array and the bases or offsets for the indexes used to access elements of the array. These offsets are represented as values to be subtracted from every index. Thus,

```
(setq a (make-based-array '((-2 2))))
```

would contruct a vector that in FORTRAN would be represented as

```
DIMENSION a(-2:2)
```

and

```
(setq a (make-based-array '((5 8) (0 9))))
```

would create an array that in FORTRAN would be represented as

```
DIMENSION a(5:8,0:9)
```

The `make-based-array` macro has the same arguments as the CommonLisp `make-array` macro with the exception that the dimensions are represented as shown above and that only *simple* arrays may be constructed.

Normal CommonLisp arrays are allowed wherever LLVS allows *based arrays* to be used.

A.21 make-plane-from-array

This function creates a *plane* from a *based-array*. The location of the *plane* is obtained from the offsets of the *based-array*. If the *based array* is not of rank 2, an error results. The result is the new *plane*.

Note — *While all two dimensional Lisp arrays may be made into* **planes**, *not all such* **planes** *can be passed to image operators written in* C[30]. *The* `convert-plane` *image operator may be used to convert to an acceptable* **plane** *type.*

```
(make-plane-from-based-array based-array)
```

A.22 make-based-array-from-plane

This function creates a *based-array* from a it plane. The location of the *plane* is used as the offsets of the *based-array*. Users should be aware that when converting from *planes* to *based-arrays* and back it is possible to obtain a different *type* of *plane* than than the original[31]. The functions `plane-type-from-array` and `array-type-from-plane` may be used to determine the resultant types.

```
(make-based-array-from-plane based-array)
```

[30]Lisp arrays of CommonLisp types `bit`, `(unsigned-byte 8)`, `(unsigned-byte 16)`, `(signed-byte 32)`, and `single-float` will result in *planes* of acceptable type in VaxLisp.

[31]This results because *planes* are not necessarily stored as CommonLisp arrays. Because the representation used for some *types* of *planes* can not be mapped efficiently into a particular implemtation of CommonLisp array, other structures may be used (i.e. Alien structures in VaxLisp).

A.23 new-plane

This function contructs a new *plane* which is returned as the result. The *plane* pixel data is not initialized.

```
(new-plane type level size &key (location '(0 0))
                                (background-value 0)
                                (associations nil)
                                (allocation :dynamic))
```

- type

  This is the *type* of data to be stored in the *plane* and should satisfy `plane-type-p`.

- level

  This is the *level* of the new *plane* given as a non-negative integer.

- size

  This is the *size* of the new *plane* given as a list of two integers — (number-of-rows number-of-columns).

- location

  This is *location* of the new *plane* given as a list of two integers.

- background-value

  This is the *background-value* of the new *plane*.

- associations

  This is the association list of the new *plane* and is usually NIL.

- allocation

  If this argument is :STATIC, `new-plane` attempts to allocate the *plane* in an area that is not subject to garbage collects.

A.24 numeric-plane-type-p

This function returns T if its argument is a *plane type* that can be processed by all image operators. While *planes* can be contructed whose pixels are arbitrary Lisp structures, only *planes* containing pixels of certain specific types can be processed by all image operators.

```
(defun numeric-plane-type-p (ty)
 (member ty
         '(:bit
           :byte
           :short
           :int
           :float)))
```

A.25 pass-plane-data & plane-pixels-c

These functions obtain the actual pixel data for passing to a *per-plane* image operator. For PASS-PLANE-DATA, if the *plane* does not exist, a dummy *plane* is passed. A non-existent *plane* is denoted by a NIL. A *plane* whose pixels can't be passed to C causes an error report.

For PLANE-PIXELS-C, if the *plane's* pixels are not the proper type to be passed to C, the *not-alien-action* argument is checked. If nil, NIL is returned, otherwise an error is reported.

```
(pass-plane-pixels plane)
```

```
(plane-pixels-c plane &optional (not-alien-action t))
```

- plane - the plane whose pixels are requested.
- not-alien-action - flag checked when the pixels are not of the appropriate format for C.

A.26 plane-association

This function returns the value of an association from the *associations* list of a *plane*. This function may be used with the CommonLisp SETF form. The *key* argument specifies the key whose value is to be returned. If no value if found, NIL is returned.

```
(plane-association plane key)
```

A.27 plane-associations

This function returns the *associations* list of a *plane*. This function may be used with the CommonLisp SETF form.

```
(plane-associations plane)
```

A.28 plane-background-value

This function returns the *background-value* of a *plane*. This function may be used with the CommonLisp SETF form.

```
(plane-background-value plane)
```

A.29 plane-column-dimension

This function returns the number of columns of a *plane*.

```
(plane-column-dimension plane)
```

A.30 plane-direction-p

This function returns T if its argument is :INPUT or :OUTPUT.

```
(defun plane-direction-p (dir)
  (member dir '(:output :input)))
```

A.31 plane-level

This function returns the *level* of a *plane*. This function may be used with the CommonLisp SETF form.

```
(plane-level plane)
```

A.32 plane-location

This function returns the *location* of a *plane* as a two element list of integers.

```
(plane-location plane)
```

A.33 plane-row-dimension

This function returns the number of rows of a *plane*.

```
(plane-row-dimension plane)
```

A.34 plane-size

This function returns the *size* of a *plane* represented as a list of two integers.

```
(plane-size plane)
```

A.35 plane-type

This function returns the *type* of a *plane*. The result will be :BIT, :BYTE, :SHORT, :INT, :FLOAT, or T.

```
(plane-type plane)
```

A.36 plane-type-p

Planes must be one of the following types. Type *T* denotes a *plane* containing pixels that are arbitrary Lisp objects. This function returns T if the argument is a valid type specifier for a *plane*.

```
(defun plane-type-p (ty)
 (cond
  ((numeric-plane-type-p ty) t)
  ((eq ty t) t)
  (t nil)))
```

A.37 plane-type-from-array

This function returns a *plane type* that can be used to represent the elements of a CommonLisp array as pixels in a plane.

```
(plane-type-from-plane based-array)
```

A.38 result-plane

This function constructs a new *plane* of the appropriate *level*, *size*, and *location* for use in an operation. The *background-value* defaults to 0. The result is the new *plane*.

This function is capable of constructing a *plane* whose elements can be any valid Lisp value. *Planes* whose elements are not the standard image elements can not be used with most of the image operators.

`Result-plane` differs from `construct-plane` in that if the *special* variable *LLVS-RESULT-PLANES-HOLDER* is not NIL, the first element of its value is returned as the result without any further processing. Also, if `result-plane` does actually allocate a *plane* and the *special* variable *LLVS-ALLOCATE-STATIC-PLANE* is T, the *plane* will be allocated in a *static* area of memory. *LLVS-RESULT-PLANES-HOLDER* is controlled by the `use-plane` form.

`Result-plane` always binds *LLVS-ALLOCATE-STATIC-PLANE* to NIL and pops the first element off of the *LLVS-RESULT-PLANES-HOLDER* list.

```
(result-plane type limits)
```

- type

  This is the *type* of the new *plane* and should satisfy `plane-type-p`.

- limits

  This is the association list returned by `union-plane-limits` or `intersect-plane-limits` or given by the user which describes the extent of the *plane* to be constructed.

A.39 set-pixel

This function is similar to `get-pixel` but sets an element of the *plane* instead of returning it. While most image operators may be applied to *planes* containing numeric pixel data only, it is possible to construct *planes* where each pixel location contains a more complex structure such as a Lisp list. Very often the user will want to contruct such a *plane* that is geometrically isomorphic to some image data *plane*. This function allows the user to set elements of *planes* using the same coordinate system as used by the image operators[32]. For elements that do not exist the operation does nothing. The result of the function is the value set.

```
(set-pixel level value plane row column)
```

- level

  This is the *primary level* used to transform the *row* and *column* parameters before seting value in the *plane*. If the value is NIL, the *level* of the argument *plane* is used.

- value

  This is the value to be stored. An error is reported if this value is of a type that can not be stored in the plane.

- plane

  This is the plane that will be modified.

- row

  This is the row coordinate at which to set the data and is relative to the absolute origin.

- column

  This is the column coordinate at which to set the data and is relative to the absolute origin.

[32] See the discussion of *level* and *location* in the Representation of Image Data section.

A.40 static-plane

This macro sets the *special* variable *LLVS-ALLOCATE-STATIC-PLANE* to T, adds the *name* to the list bound to *LLVS-STATIC-PLANE-LIST*, executes the *form* binding the result to *name*. The result is the *plane* allocated.

```
(static-plane name form)
```

A.41 union-plane-limits* and union-plane-limits

These routines generate constraints that can be used for the value of the *limits* argument to most image operators. The limit constrains the processing to the area of union of the specified *planes*. The result of the function is an association list containing pairs for :START-ROW, :END-ROW, :START-COL, :END-COL and :LEVEL. The level is included here because the other values depend on the value for :LEVEL.

```
(union-plane-limits* limits &rest planes)
```

```
(union-plane-limits limits planes)
```

- limits

  This is the *limits* for the *conceptual plane* for which the constraints are to be determined. Values for :START-ROW, etc, in this association list have higher precedence than those obtained from the *planes* specified. If *limits* is NIL then the maximum of the *levels* of the argument *planes* is used.

- planes

  These are the *planes* for which the union is determined. The first form of the function allows the *planes* to be listed as in

  ```
  (union-plane-limits* limits red green blue)
  ```

  while the second is used as in

  ```
  (union-plane-limits limits (list red green blue))
  ```

A.42 unique-plane

This form is intended for use in image operators that are composed of calls to several other image operators. Its effect is to nullify the effect of `use-plane` and `static-plane`. For example, assume an image operator `my-op` calls image operators `op1`, `op2`, and `op3` as in

```
(defiop my-op (pl)
" "
" "
(op1 (op2 (op3 pl))))
```

If the user calls my-op with

```
(use-plane im1i8 (my-op my-plane))
```

then *plane* im1i8 will be set to the result of the op3 operator instead of the expected result from the my-op operator. If, however, my-op were written as

```
(defiop my-op (pl)
" "
" "
(op1 (unique-plane (op2 (op3 pl)))))
```

plane im1i8 would be set to the result of the my-op operator.

A.43 use-plane

This form is used when the user wishes a specific *plane* to be used as the result *plane* for an image operator. This macro sets the *special* variable *LLVS-RESULT-PLANES-HOLDER* to the value of its first argument and executes the specified form which should be an image operator. If the first argument is not a *plane* or a list of *planes*, an error report is generated.

```
(use-plane plane form)
```

The *plane* argument may be a list of *planes* or a single *plane*.

A.44 vis-error

This function generates an error message.

```
(vis-error error-text &key (error-level :fatal)
                           values)
```

- error-text

  This should be a string of text describing the error. This string should be a control string as understood by the CommonLisp FORMAT function.

- error-level

  This defines the level of the error. This level can be one of the following:

  - :WARNING

    This level causes the message to be displayed and control is returned to the caller.

  - :FATAL

    This level causes the message to be displayed and the image operator function to be aborted by a call to the CommonLisp ERROR function.

  - :INFORM

    This level causes the message to be displayed and control to be returned to the caller. It differs from :WARNING in the preamble text displayed.

- value

  This is the list of values to be supplied to the CommonLisp FORMAT function along with the *error-text*. A single value is allowed in place of a list.

B Standard Parameter Definitions

The IOP-PARAMETER definitions listed in this appendix are available for use by the writers of image operators.

llvs-plane-def

```
(defconstant llvs-plane-def
  (make-iop-parameter
    :allowed-values 'plane
    :default :no-value
    :display 'describe-plane
    :prompt "Specify the ~A plane: "
    :help

"Enter an actual plane that already exists. A plane contains the
actual image and is composed of pixels arranged as two dimensional
array of numbers."

  ))
```

llvs-result-type-def

```
(defconstant llvs-result-type-def
  (make-iop-parameter
    :allowed-values 'numeric-plane-type
    :default :unsigned-byte
    :display nil
    :prompt "Specify the result plane type: "
    :help

"Enter :BIT, :UNSIGNED-BYTE, :SIGNED-BYTE, :INTEGER, or
:SINGLE-FLOAT.  This is the type of result plane that will be
generated."

  ))
```

llvs-level-def

```
(llvs-help 'level
```

```
"This is the level at which the image operator will work. The level
is a measure of the resolution of a plane and is used to control the
stride in a plane when planes of different levels are used in the
same operation. If you enter NIL, the image operator will pick the
level. "

  )

(defconstant llvs-level-def
  (make-iop-parameter
    :allowed-values '(integer 0 10)
    :default 0
    :prompt "Enter the level? "
    :help (documentation 'level 'llvs-help)
  ))
```

llvs-bounds-action-def

```
(llvs-help 'bounds-action

"This parameter controls how out-of-bounds references are
handled by the image operator. An out-of-bounds reference is
a reference to a pixel that does not actually exist in a
plane. The allowed values for this parameter are:

 :ERROR - generate an error message and halt.
 :NEAREST-VALUE - use the nearest existing pixel.
 :BACKGROUND-VALUE - use the plane's background-value.

  )

(defconstant llvs-bounds-action-def
  (make-iop-parameter
    :allowed-values 'out-of-bounds-check
    :default :error
    :prompt "Enter the out-of-bounds action? "
    :help (documentation 'bounds-action 'llvs-help)
  ))

(deftype out-of-bounds-check ()
 '(satisfies bounds-action-p))
```

```
(defun bounds-action-p (ac)
(member ac
  '(:error :nearest-value
    :background-value :no-checking)))
```

llvs-mask-plane-def

```
(llvs-help 'mask-plane

"This parameter specifies the mask plane which is used to
control to which pixels in the conceptual plane the image
operator is applied. If this parameter is NIL, the image
operator is applied to all pixels. The mask plane is an
actual plane accessed in the same way other planes in the
operation are accessed. Each pixel is compared to the mask
value parameter. Where they match the operation is applied."

  )

(defconstant llvs-mask-plane-def
  (make-iop-parameter
    :allowed-values 'plane
    :default nil
    :display 'describe-plane
    :prompt "Specify the mask plane: "
    :help (documentation 'mask-plane 'llvs-help)
  ))
```

llvs-mask-value-def

```
(llvs-help 'mask-value

"This is the value or values used to decide which pixels in
the mask plane specify a location in the other planes at
which an operation is to be accomplished. Where the value of
the pixel in the mask plane matches the operation is done.
This parameter may be a value or a sequence of values."

    )

(defconstant llvs-mask-value-def
  (make-iop-parameter
```

```
    :allowed-values 'number-sequence
    :default 1
    :prompt "Enter the mask plane matching value sequence? "
    :help (documentation 'mask-value 'llvs-help)
  ))

(deftype number-sequence ()
 '(satisfies number-sequence-p))

(defun number-sequence-p (seq)
  (or (numberp seq) (every 'numberp seq)))
```

llvs-start-row-def

```
(llvs-help 'start-row

"This parameter specifies the row number in the conceptual plane at
which the image operator begins. It should be an integer."

   )

(defconstant llvs-start-row-def
  (make-iop-parameter
    :allowed-values 'fixnum
    :default 0
    :prompt "What is the starting row? "
    :help (documentation 'start-row 'llvs-help)
  ))
```

llvs-end-row-def

```
(llvs-help 'end-row

"This parameter specifies the row number in the conceptual plane at
which the image operator stops. It should be an integer."

   )

(defconstant llvs-end-row-def
  (make-iop-parameter
    :allowed-values 'fixnum
    :default 0
```

```
    :prompt "What is the last row? "
    :help (documentation 'end-row 'llvs-help)
  ))
```

llvs-delta-row-def

```
(llvs-help 'delta-row

"This specifies the stride over rows in the conceptual plane and is
usually the value 1. A value of 2 would skip every other row."

  )

(defconstant llvs-delta-row-def
  (make-iop-parameter
    :allowed-values 'fixnum
    :default 0
    :prompt "What is the delta row? "
    :help (documentation 'delta-row 'llvs-help)
  ))
```

llvs-start-col-def

```
(llvs-help 'start-col

"This parameter specifies the column number in the conceptual plane
at which the image operator begins. It should be an integer."

  )

(defconstant llvs-start-col-def
  (make-iop-parameter
    :allowed-values 'fixnum
    :default 0
    :prompt "What is the start col? "
    :help (documentation 'start-col 'llvs-help)
  ))
```

llvs-end-col-def

```
(llvs-help 'end-col
```

```
"This parameter specifies the column number in the conceptual plane
at which the image operator stops. It should be an integer."

)

(defconstant llvs-end-col-def
  (make-iop-parameter
    :allowed-values 'fixnum
    :default 0
    :prompt "What is the last column? "
    :help (documentation 'end-col 'llvs-help)
  ))
```

llvs-delta-col-def

```
(llvs-help 'delta-col

"This specifies the stride over columns in the conceptual plane and
is usually the value 1. A value of 2 would skip every other column."

)

(defconstant llvs-delta-col-def
  (make-iop-parameter
    :allowed-values 'fixnum
    :default 1
    :prompt "What is the delta col? "
    :help (documentation 'delta-col 'llvs-help)
  ))

(llvs-help 'limits
```

llvs-limits-def

```
"The limits parameter is an association list with pairs formed using
the following keys - :LEVEL, :START-ROW, :END-ROW, :DELTA-ROW, :START-COL,
:END-COL, :DELTA-COL."

   )

(defconstant llvs-limits-def
```

```
(make-iop-parameter
  :allowed-values 'list
  :default NIL
  :prompt "Not used"
  :help (documentation 'limits 'llvs-help)
))
```

llvs-margin-def

```
(llvs-help 'margin

"This is used to specify how many extra rows and columns outside of
an area are to be used in an operation. For an example, see
EXTRACT-MASKED-PLANE."

)

(defconstant llvs-margin-def
  (make-iop-parameter
    :allowed-values 'fixnum
    :default 0
    :prompt "What is the margin? "
    :help (documentation 'delta-col 'llvs-help)
  ))
```

llvs-window-def

```
(llvs-help 'window

"The window determines the window area and positioning
around the current reference in the plane. A window of (0 0)
specifies a window area of one pixel in the input plane. A
window of (0 1) species a window area of four pixels with
the current reference in the input plane being in the upper
left hand corner of the window. A window of (-1 1) specifies
a window area of nine pixels with the current reference
being in the center and so on. If the window is not given it
is calculated by the formula
 (LIST 0 (EXPT 2 (MAX 0 (1- (-IL RL)))))
where IL is the level of the input plane and RL is the level
of the result plane.")

(defconstant llvs-window-def
```

```
     (make-iop-parameter
       :allowed-values 'list-of-two-integers
       :help (documentation 'window 'llvs-help)
              :prompt "Enter window: "
       :default :no-value))

(deftype list-of-two-integers ()
 '(satisfies list-of-two-integers-p))

(defun list-of-two-integers-p (the-list)
  (and (consp the-list)
       (eq 2 (length the-list))
       (integerp (first the-list))
       (integerp (second the-list))))
```

llvs-integer-def

```
(defconstant llvs-integer-def
     (make-iop-parameter
       :allowed-values 'integer
       :help "Enter an integer."
       :default 0))
```

llvs-float-def

```
(defconstant llvs-float-def
     (make-iop-parameter
       :allowed-values 'single-float
       :help "Enter a floating point value."
       :default 0.0))
```

llvs-number-def

```
(defconstant llvs-number-def
  (make-iop-parameter
   :allowed-values 'number
   :default :no-value
   :prompt "Enter ~A value: "
   :help "Enter a valid Lisp number."))
```

llvs-filepath-def

```
(defconstant llvs-filepath-def
     (make-iop-parameter
       :allowed-values 'string
       :default :NO-VALUE
       :display 'prin1
       :prompt "Enter the ~A file path: "
       :help
"A reference to a file is needed. This should be given as a string
in double-quote marks."
             ))
```

llvs-weighted-array-def

```
(defconstant llvs-weighted-array-def
     (make-iop-parameter
       :allowed-values 'based-array-or-array
       :default :no-value
       :display nil
       :prompt "Enter the ~A array: "
       :help

"The weighted based array is a structure of type
based-array. The based array you specify should contain the
offsets, and the cross correlation mask. The mask must be a
two dimensional array. The offsets are used to center the
mask over the image; \#(0 0) is exactly over that current
pixel. "

))

(deftype based-array-or-array ()
 '(satisfies based-array-or-array-p))

(defun based-array-or-array-p (a)
  (or (based-array-p a)
      (arrayp a)))
```

C C Support

This section documents some of the C macros, structs, and functions available to the writer of an image operator.

C.1 LLVS_PER_PIXEL.H

The following definitions are in the LLVS:LLVS_PER_PIXEL.H file. This file is meant for *per-pixel* image operators.

C.1.1 llvs_usercommon

This `struct` holds the current row and current column of the *conceptual plane*. It is accessible to the P function of an image operator using the *per-pixel* method and is written to by the `c-per-pixel-driver` each time the P section is called.

This definition is brought in by including file `llvs:llvs_per_pixel.h`.

```
globalref struct {
    int current_row;
    int current_column;
    } llvs_usercommon;
```

C.1.2 access function arrays

These definitions are brought in by including file `llvs:llvs_per_pixel.h`.

```
#define EXTERNAL globalref

#define MAXPLANE 9

EXTERNAL    int (*llvs_gpdi[MAXPLANE])();
#define GPDI(p) (*llvs_gpdi[p])(p)

EXTERNAL  float (*llvs_gpdf[MAXPLANE])();
#define GPDF(p) (*llvs_gpdf[p])(p)

EXTERNAL        int (*llvs_gpoi[MAXPLANE])();
#define GPOI(p,r,c) (*llvs_gpoi[p])(p,r,c)

EXTERNAL      float (*llvs_gpof[MAXPLANE])();
#define GPOF(p,r,c) (*llvs_gpof[p])(p,r,c)

EXTERNAL        int (*llvs_gpai[MAXPLANE])();
#define GPAI(p,r,c) (*llvs_gpai[p])(p,r,c)
```

```
EXTERNAL      float (*llvs_gpaf[MAXPLANE])();
#define GPAF(p,r,c) (*llvs_gpaf[p])(p,r,c)

EXTERNAL     void (*llvs_spdi[MAXPLANE])();
#define SPDI(v,p) (*llvs_spdi[p])(v,p)

EXTERNAL     void (*llvs_spdf[MAXPLANE])();
#define SPDF(v,p) (*llvs_spdf[p])(v,p)

EXTERNAL         void (*llvs_spoi[MAXPLANE])();
#define SPOI(v,p,r,c) (*llvs_spoi[p])(v,p,r,c)

EXTERNAL         void (*llvs_spof[MAXPLANE])();
#define SPOF(v,p,r,c) (*llvs_spof[p])(v,p,r,c)

EXTERNAL         void (*llvs_spai[MAXPLANE])();
#define SPAI(v,p,r,c) (*llvs_spai[p])(v,p,r,c)

EXTERNAL         void (*llvs_spaf[MAXPLANE])();
#define SPAF(v,p,r,c) (*llvs_spaf[p])(v,p,r,c)

EXTERNAL                    void (*llvs_wpdi[MAXPLANE])();
#define WPDI(p,ro,co,rs,cs,vec) (*llvs_wpdi[p])(p,ro,co,rs,cs,vec)

EXTERNAL                    void (*llvs_wpdf[MAXPLANE])();
#define WPDF(p,ro,co,rs,cs,vec) (*llvs_wpdf[p])(p,ro,co,rs,cs,vec)

EXTERNAL                  void (*llvs_wpai[MAXPLANE])();
#define WPAI(p,r,c,rs,cs,vec) (*llvs_wpai[p])(p,r,c,rs,cs,vec)

EXTERNAL                  void (*llvs_wpaf[MAXPLANE])();
#define WPAF(p,r,c,rs,cs,vec) (*llvs_wpaf[p])(p,r,c,rs,cs,vec)
```

C.1.3 LLVS SCRATCH AREA

Many image operators need a fairly large area to put temporary values in. LLVS provides an area that can be shared by all image operators thus reducing the amount of memory required to use LLVS. This area can be obtained by using this macro. The *type* defines the type required to the C compiler. The area is guaranteed to be of size in bytes of LLVS SCRATCH AREA SIZE.

The primary use will probably be to receive the pixels returned by a window function. The user is free to use an EXTERN statement to declare the type and organization of the array.

Examples:

```
LLVS_SCRATCH_AREA(int);

LLVS_SCRATCH_AREA(float);

LLVS_SCRATCH_AREA(char);
```

This definition is brought in by including file `llvs:llvs per pixel.h`.

```
#define LLVS_SCRATCH_AREA(type) \
  EXTERNAL type llvs_scratch[1]

#define LLVS_SCRATCH_AREA_SIZE 2048
```

C.1.4 Miscellaneous

```
#define PI         3.141592653589
#define PI2        6.283185307178
#define PI_OVER_2 1.570796326945

#define NULL 0
#define TRUE 1
#define FALSE 0

#define MIN(x,y) (((x) > (y)) ? (y) : (x))
#define MAX(x,y) (((x) < (y)) ? (y) : (x))

EXTERNAL    int (*llvs_gpdi[MAXPLANE])();
```

C.2 LLVS_PER_PLANE.H

The following definitions are in the LLVS:LLVS PER PLANE.H file. This file is meant for *per-plane* image operators.

C.2.1 PLANE

This structure describes the format of a plane passed to a *per-plane* image operator written in C. Each plane is passed to the C routine as a separate argument of type PLANE.

```
typedef struct {
    int plane_base[1]; /* plane data starts here */
    } PLANE;
```

C.2.2 MAXPLANE

This is the maximum number of *planes* that an image operator can have as arguments to one C routine.

```
#define MAXPLANE 9
```

C.2.3 PLANE INFO

Information is available for each plane used by a *per-plane* image operator. This information is passed in an array of PLANE_INFO[MAXPLANE]. The Lisp function BUILD-PLANE-INFO-VECTOR-C may be used to build the array. The background-value is passed as a C float value if the plane is a floating point plane, C int type otherwise.

```
typedef struct {
    long int datatype; /* data type */
#define BIT   0 /* bit type */
#define BYTE  1 /* unsigned byte type */
#define SHORT 2 /* signed 16 bit type */
#define INT   3 /* integer type */
#define FLOAT 4 /* float type */
    long int level; /* plane level */
    long int row_location; /* row location */
    long int column_location; /* column location */
    long int row_dimension; /* row dimension */
    long int column_dimension; /* column dimension */
    union {
long int fixnum; /* background value */
float flonum; /* dito, but as a flonum */
    } background;
    } PLANE_INFO;
```

C.2.4 MASK_VALUES

The mask values are passed as an array of type int. The Lisp function BUILD-MASK-VALUES-C can be used to prepare structure to be passed to C.

```
typedef struct {
    long int num_values;        /* number of values */
    int value_base[1];          /* value(s) start here */
    } MASK_VALUES;
```

C.2.5 LIMITS

The *limits* standard argument values are passed in this structure. The Lisp function BUILD-LIMITS-C may be used to prepare this structure.

```
typedef struct {
   long int startrow,endrow,deltarow,startcol,endcol,deltacol,level;
   } LIMITS;
```

C.2.6 TRANSLEVEL

This macro converts a row or column index from the *conceptual plane* into a row or column index into the actual *plane*. The *result* is the resultant row or column index. *Current* is the row or column index to be converted. *Deltalevel* is a positive or negative integer denoting the difference in levels between the *conceptual plane* and the actual *plane* — (conceptual level - actual level). *Location* is the row or column location of the actual plane.

```
#define TRANSLEVEL(result,current,deltalevel,location) \
if (deltalevel < 0) result = (current << -deltalevel) - location; \
else result = (current >> deltalevel) - location
```

C.2.7 GET PIXEL

This macro gets a pixel from a *plane* using a :BOUNDS-ACTION of :BACKGROUND-VALUE. *Result* is the resultant pixel converted to the type of *result*. *Bkgv* is the value to be used if the pixel is not in the actual *plane*. *Plane* is a pointer to a PLANE structure. *Row* is the row index in the actual *plane*. *Col* is the column index in the actual plane. The macro includes a case statement to determine the actual type of the *plane* and convert the pixel from that type from the type of *result* variable.

```
#define GET_PIXEL(result,bkgv,plane,row,col,pl_info) \
 if ((row) < 0 || (row) >= pl_info.row_dimension || \
     (col) < 0 || (col) >= pl_info.column_dimension) \
   result = (bkgv); \
 else { \
  register llvsoffset; \
  llvsoffset = (col) + ((row) * pl_info.column_dimension); \
  switch (pl_info.datatype) { \
    case BIT: \
      result = GETBIT(((char*)  plane->plane_base),llvsoffset); \
      break; \
    case BYTE: \
      result = ((unsigned char*)  plane->plane_base)[llvsoffset]; \
      break; \
    case SHORT: \
      result = ((short int*)  plane->plane_base)[llvsoffset]; \
      break; \
    case INT: \
```

```
    result = ((int*)  plane->plane_base)[llvsoffset]; \
    break; \
  case FLOAT: \
    result = ((float*)  plane->plane_base)[llvsoffset]; \
    break; \
  }}
```

C.2.8 SET_PIXEL

This macro sets a pixel in the *plane*. If the pixel does not exist, nothing is done. *Value* is this value is converted to the proper type and put in the pixel. *Plane* is a pointer to a structure of type PLANE. *Row* is the row index into the actual plane. *Col* is the column index into the actual plane. The macro includes a case statement to determine the actual type of the *plane* and convert the pixel to that type from the type of *value*. *Out-of-bounds* references are ignored.

```
#define SET_PIXEL(value,plane,row,col,pl_info) \
 if ((row) >= 0 && (row) < pl_info.row_dimension && \
     (col) >= 0 && (col) < pl_info.column_dimension) { \
  register llvsoffset; \
  llvsoffset = (col) + ((row) * pl_info.column_dimension); \
   switch (pl_info.datatype) { \
    case BIT: \
      SETBIT(((char*)plane->plane_base),llvsoffset,value); \
      break; \
    case BYTE: \
      ((unsigned char*)  plane->plane_base)[llvsoffset] = value; \
      break; \
    case SHORT: \
      ((short int*)  plane->plane_base)[llvsoffset] = value; \
      break; \
    case INT: \
      ((int*)  plane->plane_base)[llvsoffset] = value; \
      break; \
    case FLOAT: \
      ((float*)  plane->plane_base)[llvsoffset] = value; \
      break; \
    }}
```

C.2.9 Miscellaneous

The following C macros may be of use also.

```
#define MAXPLANE 9
```

```
#define PI          3.141592653589
#define PI2         6.283185307178
#define PI_OVER_2 1.570796326945

#define NULL 0
#define TRUE 1
#define FALSE 0

#define MIN(x,y) (((x) > (y)) ? (y) : (x))
#define MAX(x,y) (((x) < (y)) ? (y) : (x))
```

This macro generates code to check if a *plane* is empty (contains no pixels) and returns a non-zero value if the *plane* is empty.

```
#define ZERO_PLANE(pl_info) \
   (pl_info.row_dimension == 0 || \
    pl_info.column_dimension == 0)
```

This macro generates a check to see if two *planes* are at the same level of resolution and generates a non-zero value if they are not.

```
#define PLANE_LEV_ERR(pl_info,pl1,pl2)\
 (pl_info[pl1].level != pl_info[pl2].level)
```

This macro generates code to determine if two planes are at the same location in the *conceptual plane* and returns a non-zero value if they are not.

```
#define PLANE_LOC_ERR(pl_info,pl1,pl2) \
 (pl_info[pl1].row_location != pl_info[pl2].row_location)
```

This macro generates code to check if two *planes* are of the same size and generates a non-zero value if they are not.

```
#define PLANE_SIZE_ERR(pl_info,pl1,pl2) \
 (pl_info[pl1].row_dimension != pl_info[pl2].row_dimension || \
  pl_info[pl1].column_dimension != pl_info[pl2].column_dimension)
```

This macro generates code to check *limits*. If the check returns a zero value it means that the starting row or column is greater than the ending row or column; or that the dela row and delta column values are less than one.

```
#define LIMITS_ERR(lims)\
   (lims->deltarow <= 0 || \
    lims->deltacol <= 0 || \
    lims->startrow > lims->endrow || \
    lims->startcol > lims->endcol)
```

These macros get or set a bit from the specified place. The bits are indexed by a bit *offset* from a *base* address. The offset should be calculated outside of the macro call to avoid doing it more than once.

```
#define GETBIT(base,offset) \
((base)[(offset) >> 3] >>((offset) & 0x07) & 0x01)

#define SETBIT(base,offset,value) \
if (value != 0) (base)[(offset) >> 3] |= 1 << ((offset) & 0x07); \
else (base)[(offset) >> 3] &= ~(1 << ((offset) & 0x07))
```