Using LLVS Under VMS

James H. Burrill
Robert Heller

COINS Technical Report 87-41

May, 1987

ABSTRACT

This tutorial will show you how to get started with using the Low Level Visions System under the VMS operating system. First it will show you how to use LLVS to process images using standard supplied image processing operations. This is followed by a simple example of how to write a simple image operator of your own and apply it to some image. You will need some familiarity with Lisp.

Using LLVS Under VMS

James H. Burrill & Robert Heller

May 8, 1987

Abstract

This tutorial will show you how to get started with using the Low Level Visions System under the VMS operating system. First it will show you how to use LLVS to process images using standard supplied image processing operations. This is followed by a simple example of how to write a simple image operator of your own and apply it to some image. You will need some familiarity with Lisp.

Contents

List of Figures

1 Introduction - the VISIONS System Approach

A general, partially working, image understanding system as documented in [1] has evolved over the last twelve years at the University of Massachusetts at Amherst. The system design embodies certain assumptions about the process of transforming visual information. Figure 1 is an abstraction of the multiple levels of representation and processing in the VISIONS system. The successful functioning of the system involves extracting image events which are then used to hypothesize scene and object parts for quick access to knowledge structures (called schemas) which capture the object descriptions and contextual constraints from prototypical scene situations. The hierarchically organized schemas contain interpretation strategies for top-down control of intermediate grouping strategies and allow feedback from high-level hypotheses to low-level processing.

The general strategy by which the VISIONS system operates is to build an intermediate symbolic representation (ISR) of the image data using segmentation processes applied to the image data. From the intermediate level data, a partial interpretation is constructed by associating an object label with selected groups of the intermediate tokens. The object labels are used to activate portions of the knowledge network related to the hypothesized object. Once activated, the procedural components of the knowledge network direct further grouping, splitting and labeling processes at the intermediate level to construct aggregated and refined intermediate events which are in closer agreement with the stored symbolic object descriptions.

Briefly, the three levels of representation are:

1. Low-Level - Here, numerical arrays of direct sensory data are stored, including the results of algorithms which produce point/pixel data in register with the sensory data (e.g. a depth map produced from stereo point matching). This is the level at which the LLVS is used.

2. Intermediate-Level - This is referred to as the Intermediate Symbolic Representation (ISR) because symbolic tokens for regions, lines, and surfaces with attribute lists are constructed for the image events that have been extracted from the low-level data. Aggregate structures produced by grouping, splitting, and/or modifying the primitive image events or other aggregates are also represented symbolically as tokens in the ISR. An interface to the ISR system is provided by the LLVS.

3. High-Level - The knowledge base (called Long Term Memory or LTM) consists of a semantic network of schema nodes, each of which has a declarative and procedural component. The network is organized in terms of a compositional hierarchy of PART-OF relations and a subclass hierarchy of IS-A relations.

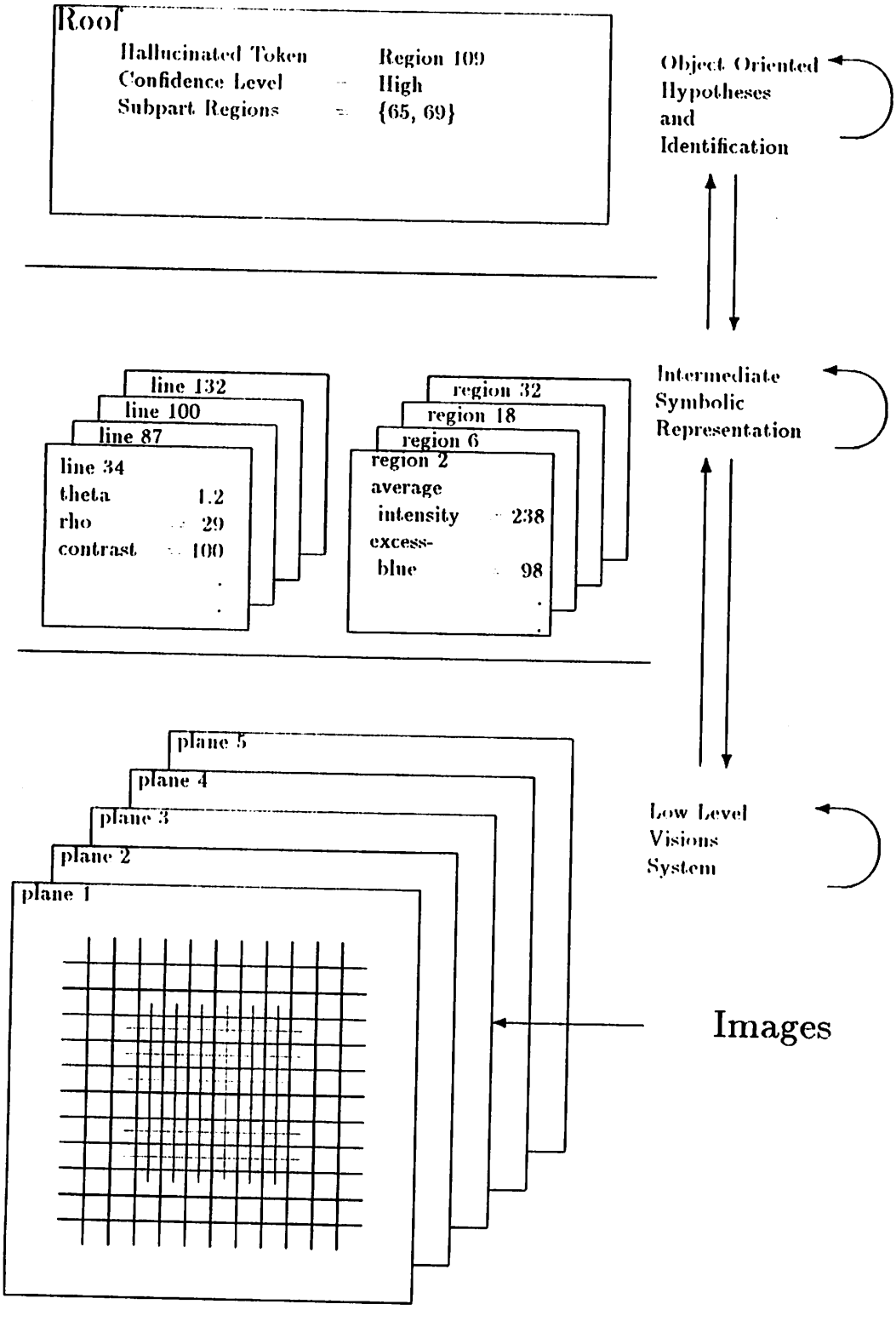

Figure 1: Image Understanding Hierarchy

The interpretation process constructs a network in Short-Term Memory or STM which is composed of image-specific instances of portions of the LTM network.

This manual is concerned with getting you started using and constructing processes that operate on the low level image data. These processes are called "image operators". Following are some examples of what these image operators have been used for:

- Segmentation

  Segmentation processes are applied to the numerical arrays of low-level data to form a symbolic representation of regions and lines and their attributes such as color, texture, location, size, shape, orientation, length, etc.

- Motion and Stereo

  Multiple frames can be analyzed via motion and stereo algorithms to produce displacement fields, and from these displacement fields depth arrays can be extracted.

- Goal-Oriented Resegmentation

  Feedback to the lower-level processes for more detailed segmentation can be requested in cases when interpretation of an area fails, when an expected image feature is not found, or when conflicting interpretations occur.

The Low-Level Visions System (LLVS) is a software environment specifically designed to facilitate all low-level image analysis research for the VISIONS image interpretation project. It is a direct descendent of the Image Operating System which has been in use since 1979 as seen in [2]. LLVS was designed to simplify the process of implementing, testing, and experimenting with new operations to be applied to images. LLVS supplies a set of contructs useful in this work. You can usually ignore various bookkeeping details and focus instead on the operation to be performed. Ways of creating, obtaining, modifying, displaying, and saving images are provided. An effort has been made to make LLVS usable for processing images in a production mode.

The LLVS consists of a high-level interpretive control language (CommonLisp) with efficient "image operators" written in C. The image operators are viewed as local operators to be applied in parallel, at some level of resolution, to all pixels in an input image. The system is useful as both a research tool for algorithm development and as a production system for initial processing of images; it is currently being further developed and extended.

LLVS provides a "friendly" environment for both experienced and naive users through the use of LISP as the high-level control language. LISP is particularly

well suited for an experimental environment since it is an interpreted language and can therefore be modified interactively. Use of LISP allows you to interact with the system using LISP expressions. Images appear as LISP objects. On-line help, interactive verification of parameters, and simple installation of new operators are all provided. The environment supports the dynamic definition of experiments with various images and image operators. LLVS provides control mechanisms (via LISP) by which you can compose image operators and control the application of the image operators.

The computational structure of images in LLVS is based on the concept of a *processing* cone as described in [3][4]; the processing cone is a model for hierarchical parallel array processing and is related to other hierarchical structures. At each level of resolution n, the *cone* contains 2^n by 2^n pixels, with a vector of values, V, stored at each pixel. The corresponding vector element V_i for all pixels at a given level of resolution can be considered as a single two-dimensional entity; this slice across the cone is referred to as a *plane*. Each pixel at level n has four unique descendents at level $n + 1$ and exactly one ancestor pixel at each level $m < n$, $m \geq 0$.

A *plane* has attributes of *level*, *size*, *location*, *type*, *background-value*, and additional user-definable values. Having a size independent of the level and including a location allows part of a plane to be extracted for independent processing in a very efficient way and then related back to the original plane or other parts of the original plane. The type specifies the format in which the pixels of the plane are stored. The background-value provides a value to be used for neighborhood operations which access non-existent pixels.

Image operators are functions which are evaluated on a set of argument planes and which produce one or more output planes. Execution of an image operator is defined as the local parallel application of a prototype function to all pixels in a plane; the function is applied to a neighborhood around each pixel. This approach will allow the translation to hardware array technology to occur in a natural manner.

A pixel-centered coordinate system and correct system handling of boundary conditions greatly simplify the specification of the prototype image operator, since the operator generally does not need to compute subscripts for neighboring pixels or consider special cases for neighboring pixels which fall outside of the plane. The specification of the image operators is independent of the level of resolution of the data to which it will be applied and it is not necessary to specify the data type of the input plane. Images of different data types and ranges can be handled without any extra code in the image operator, since all access to the image data is through special access functions. Thus, image operators can be interactively applied during the debugging stage to a small window of the image (by specifying *limits* for the processing), or a coarse representation of the data (by doing the operation at a different *level*), and then later applied to the full image at fine resolution. Image operators can be easily added to the system for any function desired by the user.

The three most important characteristics which distinguish LLVS from other image analysis systems are:

1. The partitioning of the image analysis problem into interpretive control and non-interpretive image operators allows good dynamic control of the interactive experimental environment without sacrificing image operator efficiency.

2. The structure imposed on the image operators simplifies the construction of these operators which minimizes the time and cost of implementing them.

3. The human engineering of the user-interface and automatic documentation make the system easy to use for a novice, yet flexible and convenient for an expert.[1]

1.1 References

[1] A. Hanson and E. Riseman, "The VISIONS Image Understanding System - 1986", COINS Technical Report 86-62, 1986.

[2] R. R. Kohler and A.R. Hanson, "The VISIONS Image Operating System", *Proc. of 6th International Conference on Pattern Recognition*, Munich, Germany, October, 1982.,

[3] A. R. Hanson, and E.M. Riseman, "Preprocessing Cones: A Computation Structure for Scene Analysis", COINS Technical Report 74C-7, University of Massachusetts at Amherst, September 1974.

[4] A. R. Hanson, and E.M. Riseman, "Preprocessing Cones: A Computation Structure for Image Analysis", in *Structured Computer Vision*, (S. Tanimoto and A. Klinger, Eds.), Academic Press, New York, 1980.

1.2 Acknowledgments

Many people contributed greatly to the ideas expressed in the LLVS system; sometimes suggesting important ideas that were subsequently incorporated, or, strenuously 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.

[1] 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.

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 Jim Burrill contributed the remaining Lisp code.

2 Getting Started

This section will show you enough so that you will be able to use LLVS by yourself to apply some standard operations on images and understand some of what is involved.

2.1 What Does LLVS Do?

2.1.1 A Typical Operation - Smoothing

The basic "object" supported by the LLVS is the *plane*. A *plane* is a rectangular array of numbers. These numbers are called *pixels*. Pixels are usually obtained by *digitizing* a picture. Thus, each pixel is a "sample" of a small area of an actual image. Even if each photon reflected from a scene were recorded as a separate pixel, the resulting pixels would still be samples *but at a* very *fine level of resolution.* Various methods of sampling are used and each one results in a different amount of error — the difference between the actual scene and the sample. Many of the operations applied to *planes* are attempts to reduce the effects of this type of error.

A typical operation performed on digitized images is smoothing — removing some of the error introduced by the sampling and digitizing of the original scene. Smoothing is generally accomplished by scanning each pixel in an image and modifying its value based on the values of its neighbors. For example, one type of smoothing sets the value for each pixel of a new *plane* to the average of that pixel and its eight immediate neighbors from the original *plane*. In the *plane* displayed in Figure 2 the eight neighbors of the pixel having the value 22 are the pixels whose values are 18, 20, 105, 17, 21, 23, 20, and 20.

18	20	105	103
17	22	21	100
23	20	20	102
18	18	19	23

Figure 2: A typical `level 2` *plane*.

A useful smoothing operation is provided by using the Gaussian distribution to weigh the neighbor pixels. Thus, the effect of any neighbor pixel depends not only

on its value but also on its distance from the pixel being modified. The result of this type of smoothing is illustrated in Figure 3. This example used a radius for the Gaussian distribution of three pixels from the pixel being modified. Note how this smoothing tends to eliminate the difference in the image between the top right hand corner and the remainder.

23	39	67	90
21	31	53	78
20	25	39	60
19	21	28	38

Figure 3: The same *plane* after smoothing.

Let us assume that the large difference in pixel values from the top right hand corner and the rest of the image is due to a boundary of some object in the original scene. Typically, we wish to preserve such boundaries because they contain useful information about the scene. If we wish to find the boundary of a road to help guide a vehicle on it, we had better not remove the boundaries as we process the image! Figure 4 shows the result of applying another smoothing operation on the original image that tries to preserve such boundaries. This operation is the modified WEYMOUTH-OVERTON smoothing operator[2]. Much of the work being done in low level Visions research is concerned with developing better ways of doing this type of operation.

All three of the smoothing operations described are dissimilar only in the exact operation performed at each pixel. They are otherwise very much the same — each operation must visit every pixel. The order in which the pixels are visited is not important.

There are many more things that must be done in order to perform some operation on an image. A new image to hold the result must be allocated. The image to be modified must be specified. Any other parameters (such as the radius of the Gaussian distribution) must be obtained.

LLVS was designed to simplify the process of implementing, testing, and experimenting with new operations to be applied to images. LLVS supplies a set of

[2]Burrill, J.H.; "Speeding Up the WEYMOUTH-OVERTON Smoothing Operator"

18	19	102	102
18	19	19	102
21	19	19	101
19	19	19	20

Figure 4: The same *plane* after smoothing with edge preservation.

contructs useful in this work. Among these are the *plane* and a standard way of visiting pixels in a *plane* in the course of doing some operation. You can usually ignore various bookkeeping details and focus instead on the operation to be performed at each pixel. Ways of creating, obtaining, modifying, displaying, and saving images are provided as standard facilities of this system. LLVS will even generate a large part of the code required in any operation.

2.1.2 How is an Image Represented?

An image is represented by one or more objects called *planes*. A *plane* is a rectangular array of numbers. These numbers are called *pixels*.

The *resolution* of the original image is known as the *level*. The *level* is used to relate the resolution of two *planes*. If one *plane* were produced from another by reducing it (e.g. averaging the pixels in a neighborhood), the level of the new *plane* would be less than the old one. The *level* of a *plane* is an integer that 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 in one operation. 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[3]. Figure 5 relates pixels from two *planes* at different *levels*. The *level* of a *plane* has meaning only when compared against the *level* of another *plane*.

Each *plane* has a *type* which is basically the form of the pixel data. The following

[3] 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.

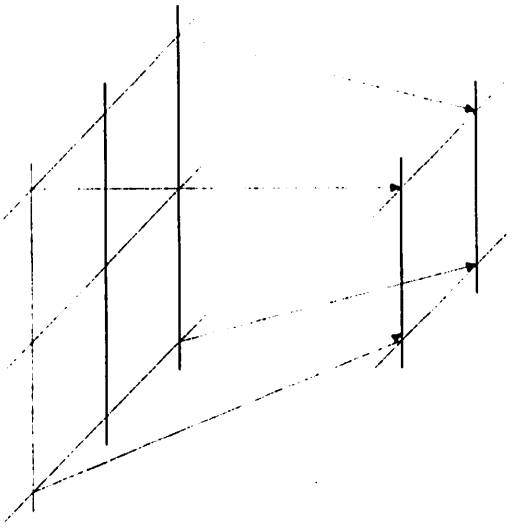

Figure 5: Reducing the Level of a Plane by 1

types of pixels are allowed:

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* (the row and column maximum for the array) of the *plane* is known. A *plane* may be related to another *plane* in a geometric way. This is specified by the *location* of the *plane* which is in terms of an absolute coordinate system — see Figure 6.

LLVS uses the concept of a *conceptual plane* to control the processing of images. 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.

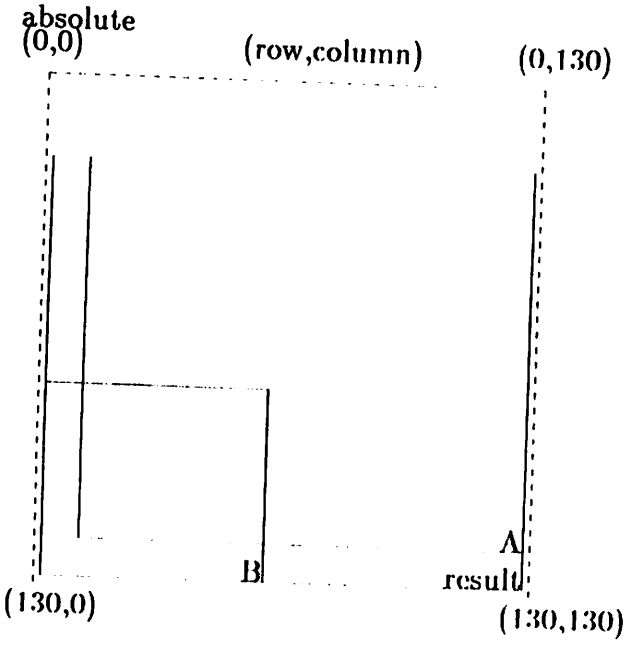

Figure 6: The *conceptual plane*.

The *location* of a *plane* is used to line up two *planes* for processing when involved in the same operation[4]. As an example, consider two *planes* **A** and **B** of the same *level*[5] as shown in Figure 6. 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*[6] 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 reason for having a concept of *location* is so that pieces of *planes* may be extracted for more intensive processing[7]. 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.

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

[5] 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.

[6] 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.

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

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. A simple transformation results in Cartisian coordinates.

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

A *plane* is defined as a structure using the CommonLisp DEFSTRUCT macro. Access to the slots of the structures are allowed using the access functions created by the standard CommonLisp DEFSTRUCT facility.

2.2 An LLVS Session

In this section we go through the actual process of producing the results seen in the last section.

2.2.1 Modifying Your LOGIN Procedure

The use of LLVS under VMS is made easier by defining several VMS "logicals" and "symbols". These logicals are easily defined by including the following line in your LOGIN.COM file:

```
$ @vis3$disk:[llvs]llvs_login.com
```

When you next connect to VMS, this call to a procedure results in the logicals and symbols being defined. The net effect of executing this DCL command will be to define several logicals that map into directories in which the LLVS files are kept.

- LLVS -- the root directory in which is kept most things you will reference.
- LLVS_SYSFUNCTIONS -- the directory containing the code for most standard image operator.
- LLVS_PUBLICFNS -- the directory containing image operators created and supported by individulas using LLVS.
- LLVS_MANUALS -- the directory containing this tutorial, the LLVS reference manual, and other documents.
- LLVS_PROCEDURES -- contains DCL command files of general use.
- LLVS_LISP -- the directory containing most of the Lisp code of LLVS.

2.2.2 A Place to Call Your Own

If you will be creating image operators of your own, you should also have at least one directory set aside for holding those image operators. Executing the following DCL command will tell LLVS where your image operators are.

```
$ define llvs_userfuns vis$disk:[burrill.kohler]
```

Only, instead of using the name of my directory, you should use your own as in

```
$ define llvs_userfuns your_directory_path
```

You may want to include this line in your LOGIN.COM file or, if you have several of these directories, you can put a LOGIN.COM in each directory containing the appropriate line that should be used.

You may accomplish the same task by executing the DCL procedure LLVSASSIGN in the directory that is to be used as in:

```
@llvs_procedures:llvsassign
```

2.2.3 Starting the LLVS System

Starting LLVS is simple --- just enter the DCL command

```
llvs
```

This command starts up the VaxLisp system and loads the programs that comprise the LLVS system. If this command is executed successfully, you will be greeted with the message:

```
; Loading contents of file LLVS$DISK:[LLVS]LLVS_INIT.LSP;3
; Finished loading LLVS$DISK:[LLVS]LLVS_INIT.LSP;3
Low Level Vision System - Version: 1.2
T
Lisp>
```

The characters "Lisp>" are the standard VaxLisp prompt that you will see every time the Lisp system is waiting for you to enter a command. LLVS is simply a set of programs that are available for your use when you are using Lisp. You can create your own Lisp programs or use other Lisp systems at the same time.

It is possible for the LLVS command to abort instead. This generally occurs because you have not been allowed sufficient resource privileges. VaxLisp (and LLVS) requires a large paging file to run well. If an abort happens to you, have your limits increased (30000 blocks in your page file should be sufficient).

```
; Select package (user package) into which to import LLVS symbols.
(setf llvs::*import-symbols* 'user)
; Import all LLVS exported symbols.
(llvs::import-llvs-symbols :all)
; Help the naive user out.
(setq *print-array* nil) ; Cause arrays to print quickly.
(setq *print-pretty* t)  ; Make top level output look nice.
```

Figure 7: Standard LLVS Initialization File

As part of the startup process, LLVS loads an initialization file. If a file named LLVS_INIT.LSP (or LLVS_INIT.FAS) is not found in your LLVS_USERFUNS directory, the standard one is loaded as shown above. This file sets up some defaults such as making available all of the *exported*[8] functions in the LLVS. You may copy this standard initialization file to your LLVS USERFUNS directory and change it to suit your needs.

When LLVS starts up, it uses about 16000 blocks for the VaxLisp memory. You may increase this by specifying the amount you want as in

```
llvs/memory=22000
```

Do not use your entire `paging file quota` *as that will leave no room to load the image operator code written in* C.

2.2.4 Getting Help

There is a reference manual available that documents LLVS in some detail. It may be printed by entering the VMS DCL command

```
print/queue=vis$laser/form=ln03 llvs_manuals:llvs_ref_manual.ln3
```

Have some money available — it's quite long! You can usually find a copy in the Visions Research offices.

A better approach will be to use LLVS to obtain help. An attempt has been made to put the most important information in the system itself so that you may ask the system to supply it to you. This information is available by using the CommonLisp standard information functions DESCRIBE and APROPOS. As an example try

[8]The LLVS uses the *package* facility of CommonLisp. All LLVS symbols are in the package LLVS. They must either be referenced with the package name as in LLVS:DEFIOP; or, the LLVS symbols must be *imported*.

```
(describe 'plane)
```

You will see that LLVS will give you information about what a *plane* is. Try using DESCRIBE with other terms you have encountered *and let us know if the system fails to be informative.*

DESCRIBE will also tell you how to use the various functions provided by LLVS. Try

```
(describe 'gaussian-smooth-plane)
```

This will describe one of the smoothing operators we will be using below.

When you use an image operator you can ask the system to prompt you for any of the arguments. You do this by setting the symbol *VERIFY-DEFAULT* to T as in

```
(setq *verify-default* t)
```

Then for any required arguments whose value is NIL or any *standard* arguments not specified, the system will ask you if the default is acceptable. At this point you may ask for a description of the argument and enter a new value for it.

Sometimes you may not know the exact name of some concept or function. APROPOS is used in this case. For example, suppose you are looking for a way to allocate a new *plane*. A good guess is that the function you want contains the word *plane* in it. Entering

```
(apropos "plane")
```

will list out the names of all the symbols containing the word *plane* (of which there are quite a few). We have tried to use a few well defined concepts in LLVS and to use those concept names in functions concerned with them. Thus, by knowing these concepts, you should be able to find most of what you need.

Appendix A lists the standard operators supplied with the system.

2.2.5 Running an Example Image Operator

Earlier there was an example of smoothing using different smoothing operators. Let us now actually do these various smoothing operations. First you must create or obtain a *plane* to smooth. Some *planes* are already loaded when you enter LLVS. You can obtain a list of these *planes* by entering

```
(find-all-planes)
```

which will list all symbols bound to objects of type *plane*.

Instead of using one of these *planes*, let us construct the *plane* used in the previous example. This is accomplished by

```
(setq example-plane
      (make-plane-from-array
        2
        (make-array '(4 4)
                    :element-type '(unsigned-byte 8)
                    :initial-contents '((18 20 105 103)
                                        (17 22 21 100)
                                        (23 20 20 102)
                                        (18 18 19 23)))))
```

MAKE-ARRAY is a CommonLisp function that creates an array — in this case it is a 4 by 4 matrix of integers. MAKE-PLANE-FROM-ARRAY is an LLVS function that creates a *plane* from an array. In this case, the *plane* is at *level* 2. The SETQ is a CommonLisp form that binds a symbol to a value — in this case we our saving the *plane* we have just created.

Next, we will run the three smoothing operators on this *plane*.

```
(setq average  (average-plane example-plane :window '(-1 1)))
(setq Gaussian (gaussian-smooth-plane example-plane))
(load "llvs_sysfunctions:weymouth")
(setq weymouth (weymouth-enhancement example-plane
                                     :version :modified
                                     :iterations 5))
```

If you try FIND-ALL-PLANES now, it will report three additional *planes* AVERAGE, GAUSSIAN, and WEYMOUTH. The LOAD was done to bring in the WEYMOUTH-OVERTON operator as it is not a standard image operator and therefore is not loaded when LLVS is first entered.

2.2.6 Looking at the Pixels

To display the pixels in an image, we will use FORMAT-PLANE. This will print out the values of the pixels as numbers on your terminal. Normally, you will want to use the more powerful image display devices available when you are generating *interesting* results. However, describing the use of these displays requires a book[9] in itself.

FORMAT-PLANE is very much like the CommonLisp FORMAT function and uses the same way to describe how to format numbers — the pixels of the *plane*. Try using DESCRIBE on FORMAT-PLANE. Then, for each *plane* you generated enter a form similar to

```
(format-plane t " ~4d" example-plane)
```

[9] "Show-Plane and Friends User's Manual" by Robert Heller.

Do not apply FORMAT-PLANE to the larger images as you will not find the result enlightening. Rather, use a function such as

```
(defun dis-plane (pl start-row end-row start-col end-col fmt)
  (format-plane t
                fmt
                pl
                :limits (acons :start-row start-row
                               (acons :end-row end-row
                                (acons :start-col start-col
                                 (acons :end-col end-col nil))))))
```

to display a section of the image at a time (it also illustrates the concept of *limits* as applied to image operators).

2.2.7 Leaving LLVS — or perhaps not

To leave LLVS you simply leave Lisp. In the case of VaxLisp, use the EXIT function. First, remember to save any images you created, and still want, by using the WRITE-PLANE image operator.

You may not need to exit Lisp. You can start up another process in which you can execute VMS DCL commands by entering the LLVS function VMS as in

```
(vms)
```

Or, if you are set up to use DEMACS, you can switch to it by entering a `control-e` character. Returning to Lisp and LLVS is a simple matter of LOGOFF or `control-c` respectively.

3 Writing an Image Operator

This section will illustrate the writing of an image operator.

3.1 The Usual Structure

When you took your first programming course, the instructor probably told you that all programs had an initialization section, a processing section, and a termination section. As image operators are programs it is not surprising that they have the same general form. But, because we use two different languages — Lisp for the control and user interface level and C for the computationally intense level -- this structure is even more explicit in our image operators.

We use Lisp for the top level of every image operator. Because Lisp is a dynamic language it is much more flexible in the types of interaction with the user that it easily supports. You can more easily run experiments when in this type of environment. You try something, see the results, and then try something else based upon those results. You can supply as arguments to a Lisp function other Lisp expressions. You can readily compose new functions via composition of existing functions.

However, Lisp is not as fast as a static language like C even when compiled. On a normal sized image of 256×256 pixels, an operation done at every pixel is done 65000 times. Every microsecond saved at this level is important because the faster we can run an experiment, the more experiments we can run. For this reason we write the pixel processing logic in C.

LLVS is set up to do the indexing over every pixel for you. You do not need to be concerned about how to visit every pixel in the images — how to handle images at different *levels, sizes* and *locations* that are used in the same operation. Instead, you write a C function that is called at every pixel in the *conceptual plane.* This function need only handle one pixel at a time. We call this function the P function.

Very often the P function is concerned with calculating a statistic for an entire image and must do accumulative sums, etc. These sums must be initialized. As the P function is called once for every pixel it is inconvenient for it to initialize the accumulators. Frequently, we must also provide the P function with some values to be used. These values are obtained at the Lisp level and must be passed to the C level. Therefore, there is often an I function called for these purposes. The I function written in C is called to do any initialization required at the C level. Typically, it initializes variables shared with the P function.

We also sometimes have a T function. This function is called to perform any termination required at the C level and to pass back to Lisp any values computed over the entire operation.

Unfortunately, passing values back and forth between Lisp and C is not as

straight forward as passing values between two Lisp functions or between two C functions[10]. LLVS does what it can to ease this burden and reduce the likelihood of errors.

3.2 The Lisp Code

3.2.1 Using DEFIOP and Friends

The top level of an image operator is written as a Lisp function. It has arguments like any function. But, instead of using DEFUN, we use the DEFIOP macro that is supplied by LLVS. DEFIOP will generate the Lisp forms for obtaining the arguments, the interface forms for calling the C level functions, and even the entire body of the Lisp function. Whether or not DEFIOP generates any of this code depends on how the DEFIOP form is written.

The DEFIOP macro generates some initial code in the image operator that binds the variable *OPERATOR-NAME* to the name of the image operator. This is the name used in error messages communicated to you. DEFIOP then generates the code to obtain each argument including both the ones you specify and the standard arguments.

Figure 8 illustrates how a simple image operator is written using the DEFIOP form. This operator takes two planes as input and returns a new plane whose pixels are the absolute value of the difference between the corresponding pixels in the input planes. This DEFIOP form is much like the CommonLisp DEFUN form except that it makes some assumptions because you are defining an image operator. The first assumption is that the arguments PL1 and PL2 are *planes*. Code is generated to obtain these arguments in the standard LLVS way and make sure they are *planes*.

Code is also generated so that four standard image operator arguments are obtained as well. These arguments are:

1. LIMITS — allows specification of a subset of a *plane*.
2. MASK-PLANE — used to disable processing at specific pixels.
3. MASK-VALUE — used with MASK-PLANE.
4. BOUNDS-ACTION — determines the action to take at non-existent pixels.

All four of these standard arguments have defaults that handle the majority of cases and thus values for them usually do not need to be supplied.

The code generated to obtain these arguments includes a call to the GET-PARAMETER function for each one. Use of GET-PARAMETER allows default values to be specified, values to be checked, the user to be prompted interactively,

[10]The alternative would be to use *only* Lisp or *only* C.

```
(DEFIOP difference (pl1 pl2)

"Returns the absolute value of the difference of two planes."

"Example:
  (setq difrg (difference red-plane green-plane))
"

(let ((resultpl)
      (delsum 1.0)
      (delmax 1.0)
      (number-different-pixels 0)
      (union-lims (union-plane-limits* limits pl1 pl2)))

     (declare (single-float delsum delmax) (fixnum number-different-pixels))

     ; allocate the result plane

     (setq resultpl (result-plane :float union-lims))

     ; call the I function

     (call-i-section "DIFFERENCE" "DIFFERENCE_I_SECTION")

     ; call the driver to process each pixel

     (call-p-section "DIFFERENCE"
                     "DIFFERENCE_P_SECTION"
                     (list pl1 pl2)
                     (list resultpl))

     ; call the T function to return some values

     (call-t-section "DIFFERENCE"
                     "DIFFERENCE_T_SECTION"
                     "float *delsum"
                     "float *delmax"
                     "int *number-different-pixels")

     (if *debug-io*          ;display the statistics
         (format *debug-io*
             "~&DIFFERENCE: number of different pixels = ~d
             sum of differences          = ~f
             maximum difference          = ~f"
             number-different-pixels delsum delmax))

     ; return the result plane as the result of the operator

     resultpl))
```

Figure 8: Example DEFIOP Form

help information to be supplied, and various other actions to be taken all in a standard way. 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.

After the arguments PL1 & PL2 in the example in Figure 8, there are two strings. These strings contain the *help* information that is available by use of the CommonLisp DESCRIBE function. The first string is the standard CommonLisp `documentation string`. It should be a simple one sentence description of what the image operator does. The second string is the detailed help information which should be sufficient to allow someone to use the image operator effectively. Both of these strings are required!

The documentation is followed by the *body* of the image operator. The body is written just like the body of any other Lisp function. In an image operator, the body specifies the control - what C functions to call in what order[11].

The forms in the body are processed too. DEFIOP 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 DEFIOP (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)`
  `... )`

[11] If the body is not supplied, DEFIOP will generate a body that allocates one result *plane* and calls the P function with all the input *planes* and that result *plane*. DEFIOP assumes 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".

This form is converted to the non-CommonLisp forms needed to call the C function[12]. 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"
                "int 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`[13]. The variables in the Lisp code of the image operator should be *declare*d as type `fixnum` and `single-float`. The C function is specified as returning an integer value. A result value of `float` or no result value (`void`) are also allowed.

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` every time YYY were called.

- `(CALL-T-SECTION ... )`

  CALL-T-SECTION and CALL-I-SECTION are processed identically.

When DEFIOP generates any forms in addition to the Lisp function for the image operator[14], it places them in a *special* variable called *LLVS-SECTIONS* for your edification.

A more powerful way of defining image operators is provided. The DEFINE-OPERATOR form provides you with more control over the arguments allowed for the image operator. You can specify arguments other than *planes* such as arrays, strings, scalar values and even prohibit specific standard arguments. DEFIOP actually expands into a DEFINE-OPERATOR form:

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

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

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

```
(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)
  "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.

3.3 The C Code

The C code is where we see the three sections of a standard program laid out explicitly (though you are not limited to only three or even three). The first statements in the program must be those declarations needed by the C compiler in order for it to understand the rest of your program. This includes bringing in the standard definitions for LLVS and, in our example, the definitions of the C math library functions.

```
#include <math>
#include "llvs:llvs_per_pixel.h"
```

Figure 9: C Code Preamble

3.3.1 Initializing and Passing Parameters to/from C

Our example image operator returns some statistics about the differences between the two *planes*. One is the maximum difference found. The others are the number of different pixels and the sum of all the differences. These values must be initialized *each* time the image operator is used. Since the P function is called once per pixel, it is necessary to do the initialization in a separate function -- the I function. The variables being initialized must be available to both the P and I functions. Thus, they are defined as C `static` variables.

```
static float delsum,delmax;
static int npix;

void difference_i_section()
{
  npix = 0;
  delsum = 0.0;
  delmax = 0.0;
}
```

Figure 10: An I Function

3.3.2 At Each Pixel ...

The operation at each pixel is simple — calculate the difference between the two pixels and save the absolute value of this difference. The P function is called for every pixel in the *conceptual plane*. If you laid each plane atop one another located by their locations relative to some absolute point, the rectangle enclosing them all is the *conceptual plane*. The GPDF function accesses the pixel from the specified plane (numbered starting at 0 and corresponding to the order in the argument to the CALL-P-SECTION form) at the *current row* and *current column* in the *conceptual plane*[15]. The SPDF function stores the value (x) into the pixel in the specified output plane. After that, the statistics are recorded.

The most important thing to understand about writing the P function is how to obtain and set pixels in the *planes*. This is done by using the access functions as shown in the example. There are different ways to access pixels depending on what type of access is desired and what C format for the pixel is needed. The access functions convert the type of pixel data from the type contained in the *plane* to the type you specify. The GPDF macro insures that the pixel will be in C `float` format regardless of the *type* of *plane*. The SPDF macro insures that the C `float` value, given it as an argument, is converted to the *type* of the *plane* being modified. The different access functions are listed in Appendix B.

3.3.3 When We Are Done

Once all the pixels in the two input *planes* have been processed, the statistics calculated must be returned to the Lisp level for display or other use. This is

[15]GPDF is really a C macro that expands into an indirect call to the appropriate pixel access function.

```
void difference_p_section()
{
  double x;
  x = fabs(GPDF(1) - GPDF(0));
  SPDF(x,0);
  if (x > 0.0) {
    npix++;
    delsum += x;
    if (x > delmax) delmax = x;
    }
}
```

Figure 11: A P Function

accomplished by calling a T function whose job is to handle any termination chores such as returning our statistics. We call the T function with arguments for the values we want. These arguments must all be passed by reference so that the T function can place the value in the Lisp variable. Compare the CALL-T-SECTION in the Lisp code to the specification of the arguments of the T function — they are specified identically!

```
void difference_t_section(Delsum,Delmax,Npix)
float *Delsum, *Delmax;
int *Npix;
{
  *Delsum = delsum;
  *Delmax = delmax;
  *Npix = npix;
}
```

Figure 12: A T Function

This is similar to the way values might be passed into the I function from the Lisp level except that then we may wish to pass values in by value if the I function does not need to change the CommonLisp variable value. Two very important rules must always be followed when passing values to/from the C level:

1. Floating point values must be passed by reference due to the way C expects floating point values to be represented when passed as arguments to a function.

2. Arrays must always be copied into local storage at the C level if they are to be used in a function other than the one they are passed to. In other words — addresses of CommonLisp variables can not be saved in `static` C variables because the CommonLisp storage management logic may move the variable at any time the Lisp level is active.

Appendix C contains the complete listing of both the Lisp and C code for the DIFFERENCE image operator.

3.4 If Yours Is More Complex ...

The example used in this section covers the basics of image operators for the LLVS system. The problems you will encounter will sometimes require you to write image operators that deviate from this straightforward example. It should be obvious that one can pass around more information to and from the C level and that more than one I function, T function, and P function may be used. It is even possible to change the order in which pixels are visited.

There are two sources of information that you may use besides the obvious one of asking someone else. The first is the LLVS Reference Manual which contains all of the details about the system but makes no attempt to show you how to use them. The second is the source code for existing image operators. This code may be found in the LLVS SYSFUNCTIONS directory.

4 Using Your Own Image Operators

Using an image operator consists of two steps -- preparing the C code and using the C code from Lisp. The C code should be prepared before Lisp is entered.

4.1 Preparing the C Code

The C code must first be compiled. Once compiled it must be placed in a VMS Sharable Image so that the VaxLisp CommonLisp system may load it. These steps are entered as DCL commands as shown in Figure 13.

---

1. `$ set default` *directory* `! directory containing the files`
2. `$ @llvs_procedures:llvsassign ! Specify this directory as containing the sharable image.`
3. `$ llvs_addfun` *op_name op_name - entry points*
4. `$ cc` *op_name*`/lis ! Compile the C code.`
5. `$ @`*op_name* `! Create the Sharable image.`

Figure 13: Preparing the C Code

Step 1 places you in the directory in which you have placed your source files for your image operators. Step 2 specifies that directory as one of the places LLVS should look for the Sharable Images. Step 3 creates a file that will be used in step 5 to create the Sharable Image. The first parameter is the name of the Sharable Image and must match the name used in the CALL-I-SECTION, CALL-P-SECTION, and CALL-T-SECTION forms in your Lisp code. The next parameters are the names of the .OBJ object module files to be included in the Sharable Image. These parameters must be followed by a "-" that is not at the end of the command line. The parameters after the "-" specify the entry point names that will be referenced. The result of Step 3 is a file named *op_name*.OPT that contains the information in a form usable by the VMS LINK command. You may find it interesting. Step 4 compiles the source file and creates the .OBJ file. Step 5 creates the Sharable Image.

For example, to prepare the DIFFERENCE image operator the following steps would be used:

```
$ set default vis$disk:[jones.llvs]
```

```
$ @llvs_procedures:llvsassign
$ llvs_addfun difference difference - difference_i_section -
difference_p_section difference_t_section
$ cc difference/lis
$ @difference
```

```
universal=difference_p_section
universal=difference_i_section
universal=difference_t_section
llvs_drivers:c_ppix_driver/share,sys$share:vaxcrtl/share
```

Figure 14: The .OPT File

Now, you simply enter the LLVS system and load the file containing your Lisp code.

4.2 Using the C Code and Lisp

Since an image operator consists of a Lisp function that calls C functions, it is reasonable to assume that you will be in Lisp when you use your image operator. Many facilities are provided at the Lisp level to help you. You may obtain information about a *plane* using functions such as PLANE-SIZE, PLANE-TYPE, PLANE-LOCATION, PLANE-LEVEL, etc. You may even access pixels from these planes using GET-PIXEL. Many of these functions are CommonLisp SETF forms that allow you to modify the *plane*. The following is a list of the ones you will find most useful to start with:

DESCRIBE-PLANE	FIND-ALL-PLANES
GET-PIXEL	LLVS-SUSPEND
MAKE-BASED-ARRAY	MAKE-PLANE-FROM-ARRAY
MAKE-BASED-ARRAY-FROM-PLANE	NEW-PLANE
PLANE-ASSOCIATION	PLANE-ASSOCIATIONS
PLANE-BACKGROUND-VALUE	PLANE-COLUMN-DIMENSION
PLANE-LEVEL	PLANE-LOCATION
PLANE-ROW-DIMENSION	PLANE-SIZE
PLANE-TYPE	PLANE-TYPE-P
SET-PIXEL	STATIC-PLANE
USE-PLANE	

You may use the CommonLisp DESCRIBE function as in

```
(DESCRIBE 'GET-PIXEL)
```

to find out how to use a function. You may also use DESCRIBE to obtain information about a concept as in

```
(DESCRIBE 'BOUNDS-ACTION)
```

To enter the LLVS system, type the DCL command LLVS. If the LLVS command failed, you either forgot to set up your LOGIN.COM file properly (see "Modifying Your LOGIN Procedure") or you do not have sufficient privileges. To load your Lisp code enter

```
(load "file")
```

To execute your image operator, use it like any other Lisp function. If your image operator did not work, the possible causes are legion. However, if the error is "`key not found in tree`", you did not follow the steps in "Preparing the C Code" correctly or have a mismatch in the Sharable Image name or entry point names.

4.3 Using Other People's Image Operators

Sometimes you will want to use an image operator created by someone else that is not in an LLVS directory of image operators (LLVS SYSFUNCTIONS & LLVS PUBLICFNS) You can instruct the system to look in other directories by using the DCL procedure LLVS_ADDDEF. The command would be:

```
$ llvs_adddef sys$share other_directory
```

This adds the *other_directory* to the list of places the system looks (`sys$share`). You can check what directories are searched entering the DCL command

```
$ show logical sys$share
```

4.4 You Made a Mistake!

Nothing works the first time. And, since there are so many ways in which an error can be made, it is difficult to provide a cookbook approach for finding *your* problem. The first thing you must do is decide if the problem is in the Lisp code or the C code — or is it in the interface between the two. You may use the standard VaxLisp forms and debugger for your debugging. STEP is particularly useful. You can determine from the Lisp level alone where the problem exists.

If the problem is in the Lisp code, fix it and re-load it. You can do this without leaving Lisp!

If the problem is in the C code, you must leave Lisp and re-enter because you can not load a Sharable Image with the same name as one previously loaded!

Debugging the C code is more difficult because C is a static language and you are really running from the Lisp level. There are three choices you may consider:

1. Use `printf` statements in the C code to verify logic paths and values,
2. Use the Vax Debugger -- a complex subject in itself, or
3. Use your head and figure out what could cause the symptoms you are seeing.

A combination of the three is usually needed.

If the problem is in the C code, it will have to be changed, re-compiled, and the Sharable Image created again before LLVS is again entered. See steps 4 & 5 in Figure 13.

5 Displaying Images

One of the reasons we are interested in working in Visions research is that we obtain a significant portion of our knowledge and understanding of the world through our eyes. It is understandable that we would like to use the same facilities for understanding the results of our image operators. We want to display the results as images.

5.1 Specifying the Device

If there were just one display device, all you would have to do would be to say something like

```
(show-plane p1)
```

and out would come your image on the display device. We have several display devices and more may be obtained. The devices available are:

:GRINNELL2 This is a good, general purpose, *easy to use* display. It can display black & white or 24 bit color images. It has four overlay planes that are useful for displaying segmentations super-imposed on the original image. It may be used for graphics, vectors, and has color lookup tables.

:IP8500 This device is primarily suited for digitizing images from a camera. It will also display black & white and 24 bit color images. It lacks overlay planes and a graphics capability. One of its nice features is the ability to display images in sequence. Over 800 images may be placed on a special disk and then shown as a movie on the display. While very powerful, the IP8500 is not as easy to use as GRINNELL2.

:COMTAL The COMTAL display can also display black & white and full 24 bit color images. Four overlay planes are available and can be used to display an overlay in 16 different colors. The COMTAL supplies facilities for rapidly changing the color mapping of any plane. The COMTAL, like the IP8500, requires extra commands to configure the type of display desired.

:SYMBOLICS-LGP This device is the laser printer used primarily for documentation. It is very good for displaying segmentations, lines, etc but not very good for displaying images. It has a graphics capability.

:LN03 This device is also a laser printer but with slightly less capability than the SYMBOLICS-LGP. It does display images better but the displays tie up the printer for a long time.

:TERMINAL Some terminals may also be useful for displays. Any terminal using the REGIS protocol (e.g. VT125, VT240) may be used for displaying images in a limited way. There is also support for Tektronix 4100 series terminals.

In order to use a device, the device specific drivers must be loaded. This is accomplished by using the form

```
(llvs-use-device device)
```

where `device` is one of the devices listed above. The loading usually takes awhile but, once loaded, this code is available from then on. You may switch to another device by entering another LLVS-USE-DEVICE form. LLVS keeps track of the last device you used or specified. If you do not explicitly specify a device, the last device is the one used.

5.2 Generating the Display

You may display a black & white image using the DISPLAY-INTENSITY form. This form actually calls SHOW-PLANE but is set up to make things easier for you[16]. DISPLAY-INTENSITY has a required argument which is the *plane* to be displayed. An optional argument allows you to override some of the defaults that DISPLAY-INTENSITY picks for you based on the device you specified with LLVS-USE-DEVICE.

A color image is displayed in much the same way except that color images are represented by three *planes* — red, green, and blue. DISPLAY-COLOR will display a color image on a device. It assumes that the its first argument is the red *plane*, the second is the green *plane* and the third is the *blue* plane. DISPLAY-COLOR also allows you to override the defaults it chooses.

Both of these functions call SHOW-PLANE to actually display the *planes*. DISPLAY-COLOR calls SHOW-PLANE three times. SHOW-PLANE binds the list of all of the controlling parameters for the particular device that it used to the symbol *SHOW-PLANE-DESCRIPTOR*. You may look at this list to see what information is available and needed to control a particular device. Figure 15 is an intensity display as it appears on the Symbolics Laser Printer.

5.3 Other Display Formats

It is also possible to display segmentations that are represented as a *plane*. SHOW-PLANE-EDGE will draw a line on the display between two regions of a *plane* that have different pixel values. Figure 16 is a display of a segmentation of the image in Figure 15 using SHOW-PLANE-EDGE.

[16] You may wish to display the function to see what is actually being done.

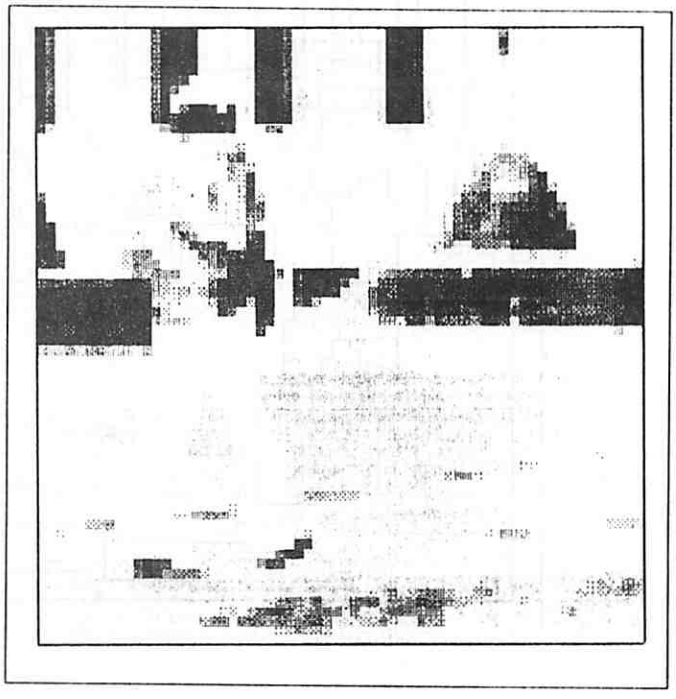

Figure 15: Display from SHOW-PLANE

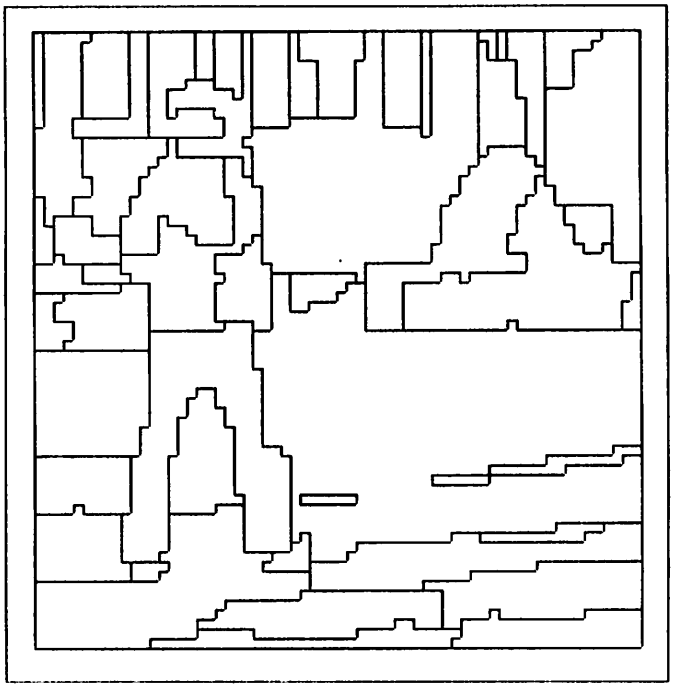

Figure 16: Display from SHOW-PLANE-EDGE

A vector field may be displayed using SHOW-PLANE-VECTOR. The vector is represented as two *planes* — the first *plane* represents the *row* displacement while the second *plane* represents the *column* displacement of the vector. The tail of the vector is placed in the position of the display corresponding to the pixels in the *planes* and the head is positioned using the *row* and *column* displacements[17]. Figure 17 is the display of the vectors generated by the gradients in the image in Figure 15.

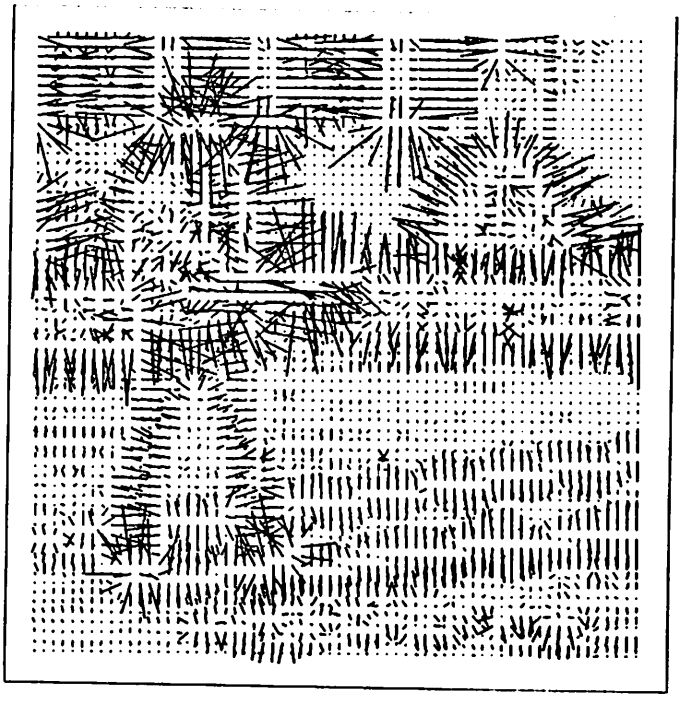

Figure 17: Display from SHOW-PLANE-VECTOR

[17]Polar coordinates may be used instead.

6 What We Didn't Tell You

This document has been made as brief as possible so that you will have an easier time concentrating on the important aspects of LLVS. To make it brief we had to leave out discussion of many features and ways that you can customize the system for your needs. This information is available in other documents.

- Low Level Vision System

  This is the reference manual for the system. It contains a description of all of the supported features.

- Show-Plane and Friends Users' Manual

  This manual has detailed information about parameters that may be used to customize displays. If you should decide to build your own display function you may want to use some of the general purpose functions described here. This manual also describes functions that are useful for positioning displays in non-standard ways — the View Port Generating Functions.

- Sharable Libraries Available in LLVS SYSFUNCTIONS

  This manual describes functions that are available for direct access to the display devices and VMS services.

A What Image Operators Are Available?

There are many standard operations that all users tend to apply to various images. The following is a list of the ones provided with LLVS.

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

convert-plane This operator converts a *plane* to a new *plane* of the specified type.

convolve-plane This operator *convolves* an input *plane* using a rectangular weighting array.

create-old-image This operator creates a file containing a *plane* in the format used by the old Visions system.

extract-masked-plane This operator determines the extent of a region in a *plane* and returns a new *plane* containing just that region.

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.

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.

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

histogram-2d-plane This operator generates a two dimensional histogram of the input *planes* and returns it as a two dimensional *based-array.*

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-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. 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 *ancestors* of the new pixel.

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

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

merge-planes This function constructs a new *plane* from a list of *planes*. 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.

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

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*.

read-plane This operator reads a *plane* in from a file and returns it.

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*.

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

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.

show-plane, show-plane-edge, and show-plane-vector These operators display a *plane* on the device specified. The *plane* is converted in an appropriate way for the display.

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*.

stats-plane This operator obtains some standard statistics from a *plane* and places them on the *plane's* association list.

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.

write-plane This operator writes a *plane* to a file. This file may be read with the `read-plane` operator.

B List of Access Functions

GPDI(p) — Return the pixel at the current reference in *plane* p as an integer.

GPDF(p) — Return the pixel at the current reference in *plane* p as a C `float`.

GPOI(p,r,c) — Return the pixel offset from the current reference by r rows and c columns in *plane* p as an integer.

GPOF(p,r,c) — Return the pixel offset from the current reference by r rows and c columns in *plane* p as a C `float`.

GPAI(p,r,c) — Return the pixel at row r, column c in *plane* p as an integer.

GPAF(p,r,c) — Return the pixel at row r, column c in *plane* p as a C `float`.

SPDI(v,p) — Set the pixel at the current reference in *plane* p to the value v represented in C as an integer.

SPDF(v,p) — Set the pixel at the current reference in *plane* p to the value v represented in C as a `float`.

SPOI(v,p,r,c) — Set the pixel at the current reference offset by r rows and c columns in *plane* p to the value v represented in C as an integer.

SPOF(v,p,r,c) — Set the pixel at the current reference offset by r rows and c columns in *plane* p to the value v represented in C as a `float`.

SPAI(v,p,r,c) — Set the pixel at row r and column c in *plane* p to the value v represented in C as an integer.

SPAF(v,p,r,c) — Set the pixel at row r and column c in *plane* p to the value v represented in C as a `float`.

WPDI(p,ro,co,rs,cs,vec) — Return as integers in vec the *window* of pixels around the current reference. The size of the *window* is rs rows by cs columns. The top, left pixel is offset by ro rows and co columns from the current reference.

WPDF(p,ro,co,rs,cs,vec) — Return as `floats` in vec the *window* of pixels around the current reference. The size of the *window* is rs rows by cs columns. The top, left pixel is offset by ro rows and co columns from the current reference.

WPAI(p,r,c,rs,cs,vec) — Return as integers in vec the *window* of pixels starting at location (r c). The size of the *window* is rs rows by cs columns.

WPAF(p,r,c,rs,cs,vec) — Return as `floats` in vec the *window* of pixels starting at location (r c). The size of the *window* is rs rows by cs columns.

C An Example Image Operator

The complete example image operator DIFFERENCE is listed here. There are some other things that need to be done before it can be used. These subjects are covered in the section "Using Your Image Operators".

Lisp

```
(defiop difference (pl1 pl2)

"Returns the absolute value of the difference of two planes."

"Example:
  (setq difrg (difference red-plane green-plane))
"

(let ((resultpl)
      (delsum 1.0)
      (delmax 1.0)
      (number-different-pixels 0)
      (union-lims (union-plane-limits* limits pl1 pl2)))

    (declare (single-float delsum delmax) (fixnum number-different-pixels))

    ; allocate the result plane

    (setq resultpl (result-plane :float union-lims))

    ; call the I function

    (call-i-section "DIFFERENCE" "DIFFERENCE_I_SECTION")

    ; call the driver to process each pixel

    (call-p-section "DIFFERENCE"
                    "DIFFERENCE_P_SECTION"
                    (list pl1 pl2)
                    (list resultpl))

    ; call the T function to return some values
```

```
    (call-t-section "DIFFERENCE"
                    "DIFFERENCE_T_SECTION"
                    "float *delsum"
                    "float *delmax"
                    "int *number-different-pixels")

    (if *debug-io*          ;display the statistics
      (format *debug-io*
            "~&DIFFERENCE: number of different pixels = ~d
            sum of differences          = ~f
            maximum difference          = ~f"
            number-different-pixels delsum delmax))

    ; return the result plane as the result of the operator

    resultpl))
```

C Code

```
#include <math>
#include "llvs:llvs_per_pixel.h"

static float delsum,delmax;
static int npix;

void difference_i_section()
{
  npix = 0;
  delsum = 0.0;
  delmax = 0.0;
}

void difference_p_section()
{

  double x;

  x = fabs(GPDF1 - GPDF0);

  SPDF(x,0);
```

```
  if (x > 0.0) {
    npix++;
    delsum += x;

    if (x > delmax) delmax = x;

    }
}

void difference_t_section(Delsum,Delmax,Npix)
float *Delsum, *Delmax;
int *Npix;
{
  *Delsum = delsum;
  *Delmax = delmax;
  *Npix = npix;
}
```

D Using Control-C and Control-B With Per Plane Operators

This appendix describes how to integrate the standard LLVS ^C and ^B handlers into an image operator written with the *per-plane* method[18] under VMS. These handlers can also be used with any C function called from LLVS.

It is usually useful and necessary to be able to monitor and possibly abort the operation of an image operator. The `c-per-pixel-driver` provides a mechanism for image operators coded with the *per-pixel* method. For *per-plane* image operators, this mechanism must set up by the writer of the operator.

The Control-B (monitor function) and Control-C (abort function) handlers are set up with three C functions and one VaxLisp macro form. The VaxLisp macro is called `llvs-without-^C^B-interupts` and is simply wrapped around the `call-c` form in the `define-operator` form. This macro is used around the form(s) directly involved with the call-out to the C coded portion of the per-plane operator. This macro behaves like a special sort of unwind-protect.

```
(llvs-without-^C^B-interupts &body forms)
```

This macro disables VaxLisp's Control-C and Control-B handlers (if any) during the execution of the C code. This allows Control-C's and Control-B's to be handled by the C code.

One C function, `llvs_start_pplane_asts`, sets up the handlers (AST — Asynchronous Software Trap) and saves the user-supplied information (function name, where the current row and the current column live, exit handler). This function is generally called once near the beginning of the image operator (although it may be called additional times to set up different function name text, location for the current row and the current column and exit function).

```
void llvs_start_pplane_asts(text,current_row,
                            current_column,exit_function)
char *text;
int *current_row;
int *current_column;
void (*exit_function)();
```

`llvs_start_pplane_asts` takes four arguments. The first argument is the address of a string. This is normally the function name but could contain any message

[18] A *per-plane* operator is generally faster and more difficult to write since the programmer must do all the indexing over the *planes*.

describing the current state of the function. This string is displayed when either a Control-B or a Control-C is entered and processed. The second and third arguments are the addresses of a pair of C `int`s, which normally contain the current row and current column being processed. The fourth argument is an exit function that is called when a Control-C abort is taken. It can be `NULL` if an exit routine is not needed. Usually an exit routine is used to provide for cleaning up loose ends (such as closing open files or freeing up allocated space) before (abnormally) returning to VaxLisp. The first, second and third arguments should point to *static* places. It is up to the user's code to keep the current row and the current column up to date.

The second function, `llvs_check_control_c`, is called in the inner loop(s) and polls for and processes Control-C's.

```
void llvs_check_control_c()
```

This function should be called frequently (once per pixel if possibly) during operator execution. This function won't return if a `^C` was hit *and* the user elected to abort (`A` option at the `^C` prompt). In this case, the exit function specified by `llvs_start_pplane_asts` is called to do whatever cleanup is needed (i.e. free up allocated space, close files, flush buffers, etc.). The function does not return if the abort is taken.

The third function, `llvs_shut_pplane_asts`, dismisses the Control-C and Control-B handlers and should be called just before returning to VaxLisp.

```
void llvs_shut_pplane_asts()
```

If a control-C abort is taken, this function does not need to be called - the `^C` handler will dismiss the `^C` and `^B` ASTs.

The Control-B handler operates totally asynchronously. It displays what the first three arguments to `llvs_start_pplane_asts` point to: a string and two integers. The Control-C handler is different. Because it needs to do both input and output as well as possibly performing a non-local exit, it can't operate asynchronously. Instead the Control-C AST simply sets a flag. The second routine (`llvs_check_control_c`) tests this flag and, if set, queries the user whether to continue or to abort. The information displayed is the same as displayed by the Control-B handler.

Example Lisp code

```
(define-operator per-plane-operator ((plane plane)
                                     &key
                                          limits
                                          mask-plane mask-value)
"Test function"
"This function is just an example of ^C and ^B usage in a
per-plane operator."

  (setq limits (union-plane-limits* limits plane))
  (let* ((result (result-plane
                   (plane-type plane) limits))
         (in-plane-pixels (plane-pixels-c plane))
         (in-plane-info (build-plane-info-c plane))
         (out-plane-pixels (plane-pixels-c result))
         (out-plane-info (build-plane-info-c result))
         (mask-plane-pixels (if mask-plane (plane-pixels-c mask-plane)))
         (mask-plane-info (if mask-plane (build-plane-info-c mask-plane)))
         (mask-value (if mask-plane (build-mask-value-c mask-value))
         (c-limits (build-limits-c limits)))

      (llvs-without-^C^B-interupts
          (if mask-plane
              (call-c "PERPLANE"
                      "PER_PLANE_OPERATOR_MASKED"
                      "PLANE *in-plane-pixels"
                      "PLANE_INFO *in-plane-info"
                      "PLANE *out-plane-pixels"
                      "PLANE_INFO *out-plane-info"
                      "PLANE *mask-plane-pixels"
                      "PLANE_INFO *mask-plane-info"
                      "MASK_VALUE *mask-value"
                      "LIMITS *c-limits")
              (call-c "PERPLANE"
                      "PER_PLANE_OPERATOR"
                      "PLANE *in-plane-pixels"
                      "PLANE_INFO *in-plane-info"
                      "PLANE *out-plane-pixels"
                      "PLANE_INFO *out-plane-info"
                      "LIMITS *c-limits")
              ))
            result))
```

Example C code

```
#include <stdio.h>                  /* standard defs */
#include "llvs:llvs_per_plane.h"  /* per-plane defs */

void per_plane_operator_masked(in_plane,in_plane_info,
                               out_plane,out_plane_info,
                               mask_plane,mask_plane_info,
                               mask_value,limits)
PLANE *in_plane;
PLANE_INFO *in_plane_info;
PLANE *out_plane;
PLANE_INFO *out_plane_info;
PLANE *mask_plane;
PLANE_INFO *mask_plane_info;
MASK_VALUES *mask_value;
LIMITS *limits;
{
static int currow, curcol;  /* currow and curcol */
int per_plane_oper_cleanup();

  /***** omitted details *****/

  /* Establish ^C and ^C processing. */

  llvs_start_pplane_asts("Per-Plane-Operator (with mask)",
                         &currow,&curcol,
                         per_plane_oper_cleanup);

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

    /* per-row setup ... (omited) */

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

      /* per pixel code ... (omited) */

      /* check for ^C */

      llvs_check_control_c();

      }
    }

  per_plane_oper_cleanup();

  /* shut down ^C and ^B handlers */
```

```
  llvs_shut_pplane_asts();
}
```

```
void per_plane_operator(in_plane,in_plane_info,
                        out_plane,out_plane_info,
                        limits)
PLANE *in_plane;
PLANE_INFO *in_plane_info;
PLANE *out_plane;
PLANE_INFO *out_plane_info;
LIMITS *limits;
{
static int currow, curcol;   /* currow and curcol */
int per_plane_oper_cleanup();

  /**** omitted details ****/

  /* Establish ^C and ^C processing. */

  llvs_start_pplane_asts("Per-Plane-Operator",
                        &currow,&curcol,
                        per_plane_oper_cleanup);

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

    /* per-row setup ... (omited) */

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

      /* per pixel code ... (omited) */

      /* check for ^C */

      llvs_check_control_c();

      }
    }

  per_plane_oper_cleanup();

  /* shut down ^C and ^B handlers */

  llvs_shut_pplane_asts();
}

per_plane_oper_cleanup()
{
    /* clean up code... */
}
```

E Linking Sharable Images

This appendix covers some of the basic things users need to to build the VMS sharable images containing the (compiled) C code used by LLVS image operators. It is not meant to *completely* cover VMS sharable images - this is handled quite well by the VMS Linker Manual (Volume 4B of the VMS 4.xxx reference manual set).

To "link" C[19] modules to VaxLisp you need to build a sharable image. Sharable images are created by the VMS Linker (DCL LINK command). Unlike the run-of-the-mill executable images built by the linker, sharable images need some extra information such as externally visible entry points and what to do about certain other "global" things. To make things easier for users, there is a VaxLisp function and a C program available to generate a "vanilla" DCL Procedure file and Linker Options file (and optionally, a transfer vector file). Most of the time you can use either the Lisp function or the C program to generate the files needed to link your image operators.

Generating a "vanilla" LLVS sharable image

There are two ways to generate the files needed to link a LLVS image operator. One way is to use the VaxLisp function `llvs:llvs-addfun` and the other way is with the program `llvs_addfun`.

```
(llvs:llvs-addfun link-opt-exe-name
                  object-list
                  entry-list
        &optional
                  transfer-vector-file)
```

This function generates a DCL procedure file and a Linker options file. It (optionally) will also generate a transfer vector file.

`Link-opt-exe-name` is the name of the procedure file, the name of the options file and the name of the sharable image (this is the *image name* used in the CALL-P-SECTION, etc. This can be any valid pathname, except it must have an empty file type and should not have a version number.

`Object-list` is a list of object file names. The object file names can have the Linker positional qualifiers `/OPTIONS`, `/LIBRARY`, `/INCLUDE=(...)` or `/SELECTIVE_SEARCH` added to the names — each element of `object-list` is considered a file specification with optional file-qualifier for the linker.

`Entry-list` is a list of universal entry points. These are the entry points you wish to actually use in the sharable image. These are in fact the names of the entry

[19] or FORTRAN, Pascal, etc.

points used by the CALL-P-SECTION, etc. `Transfer-vector-file` is the name of a transfer vector file to generate. If it is `NIL` (the default), a transfer vector file is not generated or used.

The name of the DCL procedure file generated is returned.

```
$ llvs_addfun (link-file objfile objfile - entry entry [-
transfer-vector])
```

This program (invoked as a DCL command) does the same things as the VaxLisp function `llvs:llvs-addfun`. It is provided as an alternative to the lisp function for use while at DCL level.

Example

This is an example of the use of llvs_addfun program.

```
$! generate link files for image operator FUN with object
$! files o1, o2 and o3 and entry points a, b and c
$!
$ llvs_addfun fun o1 o2 o3 - a b c
Files created:
MYDSK:[MYDIR]FUN.COM - Command procedure file
MYDSK:[MYDIR]FUN.OPT - Link options file
To link MYDSK:[MYDIR]FUN.EXE, use the DCL command:
$ @MYDSK:[MYDIR]FUN.COM
$ type fun.com
$! FUN.COM - Machine generated DCL procedure file to link
$! the sharable image MYDSK:[MYDIR]FUN.EXE, for use with the LLVS.
$!
$ LINK/SHARE=MYDSK:[MYDIR]FUN.EXE -
MYDSK:[MYDIR]FUN.OPT/opt,-
o1,-
o2,-
o3,-
llvs:userlink/opt
$ exit
$ type fun.opt
! FUN.OPT - Machine generated LINK options file for linking
! the sharable image MYDSK:[MYDIR]FUN.EXE, for use with the LLVS.
!
Universal=a
Universal=b
Universal=c
```

As can be seen, the two files generated (FUN.COM and FUN.OPT) are fairly simple. The procedure file (FUN.COM) contains a LINK command which references the generated options file (FUN.OPT), the object files (o1, o2 and o3) and the file llvs:userlink.opt. This last file simply references the two needed sharable images needed: llvs_drivers:c_ppix_driver (the LLVS per-pixel driver) and sys$share:vaxcrtl (the VAX-C run time library). The options file simply declares the three entry points (a, b and c) as "universal" symbols. This makes them accessible from "outside" of the sharable image.

Where Sharable Images Live

Sharable images live in the directory pointed to by the VMS logical name SYS$SHARE. Normally this is a system logical name bound to something like SYS$SYSROOT:[SYSLIB]. This logical name can in fact be a pathname search list. The LLVS login file redefines this logical as a process logical bound to the list:

```
LLVS_USERFUNS:,
LLVS_PUBLICFNS,
LLVS_DRIVERS:,
LLVS_SYSFUNCTIONS:,
SYS\$SYSROOT:[SYSLIB]
```

LLVS_USERFUNS is initially bound to whatever SYS$LOGIN is bound.

There is a procedure file in LLVS_PROCEDURES named `llvsassign.com` which redefines LLVS_USERFUNS to the current default directory. You should put a line like:

```
$ @llvs_procedures:llvsassign
```

in a LOGIN.COM file in the directory where you do your LLVS work.

There is also another procedure file referenced as the DCL command `llvs_adddef` which can be used to add additional paths to a search list. It is used like this:

```
$! add some$disk:[somedirectory] to the list of places where
$! sharable images live.
$ llvs_adddef sys$share some$disk:[somedirectory]
```

Alterations

While the "vanilla" procedure and options files generated by llvs:llvs-addfun or llvs_addfun are probably fine for most image operators, there will come a time when additional work must be done. Special treatment is needed for C externs and FORTRAN Common blocks, sharing data areas across two or more sharable images, or creating sharable images of libraries of code used by two or more other sharable images. *Note: You will probably have no need to read the rest of this document unless you need to do something unusual.*

C extern

VAX-C implements "externs" as overlaid, sharable, read/write program sections. FORTRAN Common blocks are implemented the same way. This will cause problems when you use these sort of variables in a sharable image. Sharable images which have sharable *and* writable program sections are supposed to be *installed.*

Installation actually causes *everyone* who uses such an *installed* image to use the *same* memory for these program sections. This is usually not what you want. If you use externs in C you should include the noshare modifier on the declaration (see section 6.2 of the VAX C Reference Manual). This makes the program section non-sharable. It is still "shared" within the sharable image because it is also an overlaid program section and the linker will assign the same address to the beginning of the program section for each link module. FORTRAN doesn't allow for a source language feature to alter the sharability of common blocks. You must fix this with the psect_attr linker option in the link options file:

```
C Typical FORTRAN Common declaration:
        common /z/ a,b,c
! link option to go with it:
psect_attr=z, noshr
```

Sharing Data Between Two or More Sharable Images

You can't use the extern storage class to share data between multiple sharable images. Instead, you need to use the VAX C globaldef and globalref storage class keywords. You need to link the sharable image with the globaldef first and declare the globaldef name as an entry point (i.e. as a Universal symbol). Then, the other sharable images are linked with the first. The storage gets allocated as part of the first sharable image's data space which is then accessed by the other sharable images. This is in fact how image operators which use the per-pixel method get access to llvs_usercommon (the structure containing current_row and current column) and the access function vectors. These variables are allocated by the C per-pixel driver with a globaldef and are referenced by the image operators with globalrefs (which are brought in by llvs:llvs_usercommon.h).

Sharing Code Libraries

Sometimes you need to access common library functions. These libraries should be separately linked as sharable images, preferably with transfer vectors. Some commonly used libraries have already been linked and reside in llvs_sysfunctions directory. These are described in a separate manual. Linking a sharable image from a library is just like linking any other sharable image. To avoid having to re-link *all* of the sharable images that depend upon the library *every* time the library is re-linked, you should use a transfer vector (which can be generated by either llvs:llvs-addfun or llvs_addfun). You must reference a sharable image in an options file using a line like:

```
llvs_sysfunctions:grlib2/share
```

If more than one sharable library is to be referenced, separate them with commas, using a trailing dash for continuation:

```
llvs_sysfunctions:grlib2/share,-
llvs_sysfunctions:grlib1/share
```