ATLAS Internal Note

Brandeis Image Analysis Libraries

Version 32

Kevan S Hashemi
Brandeis University
16 June 2003

Abstract: We have prepared a library of image analysis routines for PCs running Windows 95/98/NT, and another for Macintoshes running MacOS. We used a multi-platform compiler to compile both libraries from the same Pascal source code. This note describes how to use the libraries. We place particular emphasis upon the 'dynamic link library' for Windows 95/98/NT. We describe the routines exported by the libraries, and define their data structures.

Introduction

We have prepared a library of image analysis routines for PCs running Windows 95/98/NT, and another for Macintoshes running MacOS.
We used a multi-platform compiler to compile both libraries from the same Pascal source code. This note describes how to use the libraries. We place particular emphasis upon the 'dynamic link library' for Windows 95/98/NT. We list the routines exported by the libraries, describe what each routine does, and define their data structures.

The Macintosh library has existed for some time. We finished the prototype PC library on January 15, 1999. Both libraries contain BCAM (Boston CCD Angle Monitor), BCAL (Boston CCD and Laser), RASNIK (Red Alignment System of NIKHEF), and X-Ray MDTQA (Muon Drift Tube Quality Assurance) analysis routines. But they do not contain any routines that communicate with data acquisition hardware. We have such routines for the Macintosh, but not for the PC. Similarly, we have display and menu interface software for the Macintosh, but not for the PC.

Before we proceed, we will say a few words about our compiler. We use 'Code Warrior', a multi-platform, multi-language compiler written by Metrowerks Incorporated (www.metrowerks.com). It compiles Pascal, C, and Java source files, and can link object code from

all three languages into the same executable file. It will compile for the 68000, Power PC, or x86 processors. It runs on either a Macintosh or a PC. You can take a Code Warrior file, which contains object code and link lists for a software project, and move it from one platform to another and use it without modification.

Pascal and C both have their own standard input and output routines, and these can be compiled from the same source code for execution on a Macintosh or a PC. But neither standard Pascal nor standard C provides graphical user interface (GUI) routines. If you want a graphical user interface, you have to look outside the standard Pascal and C libraries. When you compile for the Macintosh, you can use the MacOS graphical routines. When you compile for the PC, you can use the Windows graphical routines. Some later version of Code Warrior may provide a library of operating-system independent graphical routines, but such a library does not yet exist, and we know of no other compiler that provides such a library.

We decided that the most efficient way for us to support our analysis programs is by maintaining one master copy of the source code, and compiling this copy for all necessary platforms. Compiling for different platforms is possible only if the source code does not contain any graphical routines. We have learned, however, that a graphical user interface is a practical necessity in the laboratory.

We arrived at the following solution. All the analysis code is written in standard Pascal. We compile this code into an ‘analysis library’ for both MacOS and Windows. The analysis library routines call ‘system routines’ such as ‘display_ccd_line’ and ‘new_image_ptr’. Some of these system routines you can implement with dummy procedures without affecting the numerical results of any of the analysis routines, but others you must implement fully. That is to say, some are some ‘optional’ and some ‘essential’ system routines.

Fortunately, all the essential system routines, such as memory allocation and file input-output, can be implemented in standard Pascal. It is only the optional system routines, such as the graphical routines, that require operating-system specific code for a fully-functional implementation. For your convenience, we have implemented all the essential system routines in standard Pascal, and provided dummy procedures for all the optional system routines. We have compiled this code into what we call our ‘operating system library’. You can use our operating system library with our analysis library, and thus save yourself the labor of defining any routines at all. You will have no graphical user interface, but you will get answers from the analysis routines. If you want to implement a graphical user interface, you can write your own operating system library to implement routines such as ‘display_image’ and ‘display_ccd_line’.

If you find a bug in our analysis library, we would of course like to fix it, send the fixed library to you, and have you use it right away. With a conventional library, however, your being able to use the the new version right away would be contingent upon your having a compiler to link your graphical user interface code to our library. But the Microsoft Win32 supports what it calles ‘dynamic linking’ between an application and a ‘dynamic link library’ (DLL). In dynamic linking, a ‘client program’ links to its library at execution time under the control of the operating system,

rather than at compilation time under the control of the compiler. The DLL sits in the same directory as the client program that uses it. The client program loads the DLL into memory when it starts up. When you receive an upgraded DLL, all you have to do to use it is overwrite the old one on your hard disk and start up your program.

We have compiled two dynamic link libraries: analysis.dll and os.dll. They are the DLL versions of our analysis and operating system libraries. You can make your own client program to uses these libraries, or you can obtain such a program from another laboratory. When you receive the latest version of either library, you can use it immediately.

We have also compiled a simple Windows client program, called pcapp.exe, that uses our libraries. When you run pcapp.exe, it loads both DLLs into memory and opens a text window. Under the control of the keyboard, it reads and writes image files, generates test images, performs image analysis, and writes results in its text window. Alternatively, you can control pcapp.exe by sending and receiving messages via text files. You send instructions to pcapp.exe in a text file called ‘command.txt’, and receive results in a file called ‘results.txt’.

To Macintosh users, we provide static libraries. We cannot make dynamic link libraries for the Macintosh. But we have fully functional definitions of all the analysis library’s external graphics routines, so the operating system library supports a comprehensive display. We also provide stand-alone applications that capture images from VME-based CCD Drivers, display them on the screen, analyze them, and record data to disk. We do all our own laboratory work on Macintoshes.

Where to Get the Libraries

We have compressed a selection of files in a zip archive, which you can pick up with a web browser from www.alignment.hep.brandeis.edu. The archive is called dll.zip. It contains the following files:

file	description
analysis.dll	analysis dynamic link library
os.dll	operating system dynamic link library
analysis.dll.lib	analysis.dll descriptor file
os.dll.lib	os.dll descriptor file
pcapp.exe	an example PC application using analysis.dll and os.dll
pcapp_fco.exe	file-controlled-only version of pcapp.exe
rasnik.i	an example RASNIK image
centroid.i	an example BCAM image
dll.cpp	Visual C++ code that uses analysis.dll and os.dll
dll.pdf	the latest version of this manual (PDF file)

Table: Contents of DLL.ZIP

You will find descriptions of all of the above files in the text below.

Brief Note on Pascal

In order to use the analysis library, you must know the names of the procedures it makes available, and you must know the structure and size of all parameters you pass to and from those procedures. Throughout this note, we will use extracts from our Pascal source code to define data structures. In case you do not read Pascal already, we provide the following quick introduction.

In Pascal, comments are enclosed in curly brackets. Constant definitions follow the reserved word ‘const’, type definitions follow ‘type’, and variable declarations follow ‘var’.

```
const
  pi=3.14;

type
  point_type=record
    x,y:real;
  end;{record}
  circle_type=record
    center:point_type;
    radius:real;
  end;{record}

var
  circle:circle_type;
```

Pascal distinguishes between two forms of routine. A ‘procedure’ is a routine that you call by stating its name. In the following example, the procedure takes a circle variable, and writes its fields to the console. The body of a procedural definition begins with the reserved word ‘begin’ and ends with the reserved word ‘end’.

```
procedure write_circle (circle:circle_type);

begin
  with circle do begin
    write(‘center is at’,center.x,c.y);
    write(‘radius is’,radius);
  end;{with}
end;{procedure}
```

The second form of routine is a ‘function’. You call a function by assigning its value to a variable. In the following example, we define a function that returns a value of type ‘real’, this value being equal to the circumference of a specified circle.

```
function circumference (circle:circle_type):real;

begin
  circumference:=2*pi*circle.radius;
end;{function}
```

Here is how our example procedures might be used.

```
  write_circle(circle);
  write(‘and its circumference is’);
  write(circumference(circle));
```

The ‘circle_type’ is an example of a Pascal structured type. In Pascal, we are free to pass parameters of any type to a routine. Functions can return structured types as well. But we pass only simple types to and from our DLL routines. Instead of passing a structured type, we pass a pointer to a structured type. The following table lists the simple Pascal types we use, and translates them into C types.

Pascal	C	Size (bytes)	Description
boolean	char	1	0 for true, 1 for false
integer	short	2	-32768..32767
longint	long	4	-2147483648..2147483647
real	float	4	floating point number
ptr	*	4	32-bit pointer
string	string	256	see below

Table: Description of Standard Pascal Types

The Pascal 'string' type has no immediate C replacement. The first byte of a Pascal string gives the number of characters in the string. The bytes after that are the string characters. There is no termination character. You must write your own code to translate between C and Pascal strings. Here are two Pascal procedures that do the job. The first procedure turns a Pascal string into a C string. It shifts all the characters one space to the left and adds the C string terminator character to the end.

```
procedure pascal_to_c_string (var s:string);

const
        null_character=chr(0);

var
        char_num,string_length:integer;

begin
        string_length:=length(s);
        for char_num:=0 to string_length-1 do
                s[char_num]:=s[char_num+1];
        s[string_length]:=null_character;
end;
```

The second procedure turns a C string into a Pascal string. It shifts all the characters to the right, and puts the Pascal length byte at the beginning.

```
procedure c_to_pascal_string (var s:string);

const
        null_character=chr(0); {the c-string terminator character}
        max_string_length=255;

var
```

```
        char_num,string_length:integer;

begin
        string_length:=0;
        while (string_length<max_string_length) and (s[string_length]<>null_character) do
                inc(string_length); {increment string_length}
        for char_num:=string_length downto 1 do
                s[char_num]:=s[char_num-1];
        s[0]:=chr(string_length);
end;
```

Because Pascal strings must have a one-byte length character at the beginning, they cannot be more than 255 characters long. As you will see below, we provide a routine called write_string_dll so that you can test your own version of c_to_pascal_string.

Analysis Library Data Structures

Your client program must know the exact structures of all the variables that it will use in its communications with our DLL. Compilers tend to insert padding bytes into structured variables so as to put two-byte variables on two-byte boundaries and four-byte variables on four-byte boundaries. When a four-byte variable is on a four-byte boundary, a 32-bit processor needs only one address cycle fetch it from memory. We make the padding in our data structures explicit by declaring padding bytes of the type ‘packed_byte’. A packed_byte takes up one byte of address space. We assume your compiler will put consecutive packed_bytes on consecutive byte addresses.

```
type {for padding structures}
        packed_byte=-128..127;
```

Here are the constants we use for image manipulation.

```
const {for geometry}
        ccd_origin_h=0.5;{pixels}
        ccd_origin_v=0.5;{pixels}
        max_ccd_pixel_value=255;
        mid_ccd_pixel_value=max_ccd_pixel_value div 2;
        low_ccd_pixel_value=max_ccd_pixel_value div 10;
        black_intensity=0;
        white_intensity=max_ccd_pixel_value;
        ccd_pixel_size=1;{bytes}
        ccd_pixel_mask=$000000FF;
```

All our analysis routines are intended for eight-bit gray-scale images only. We believe we could adapt them to other image formats, but we have not done so. The constants ‘ccd_origin_h’ and ‘ccd_origin_v’ tell the analysis code the coordinates of the center of the top-left pixel of the CCD in the real-valued ‘image-coordinate system’ used by some of the analysis routines.

Of the following types, the only ones you need to know about are ptr and rectangle_type. The former is a generic 32-bit pointer to a byte in memory. The latter defines a rectangular area in the image by giving the left column, right column, top row, and bottom row that form the outermost pixels of the area.

```
type {for geometry}
        ccd_pixel_value_type=integer;
        display_pixel_type = record k,l:integer; end;
        ccd_pixel_type = record i,j:integer; end;
        image_point_type = record h, v:real; end;
        pattern_point_type = record x, y:real; end;
        ccd_line_type = record a,b:ccd_pixel_type; end;
        image_line_type = record a,b:image_point_type; end;
        pattern_line_type = record a,b:pattern_point_type; end;
        image_rectangle_type=record min_h,min_v,max_h,max_v:real; end;
        rectangle_type=record top,left,bottom,right:integer; end;

type {for type-casting pointers}
        rectangle_ptr_type=^rectangle_type;
        ccd_line_ptr_type=^ccd_line_type;

type {for generic pointers}
        ptr=^packed_byte;
```

When you call one of our analysis routines, you specify an image by passing a pointer to an image_type variable. An image_type variable occupies 344 bytes in memory. It does not itself contain the image data, but it has a pointer, called ‘pixel_map_ptr’, that points to the image data.

```
const {for images in memory}
        num_spare_ptrs=10;

        image_type=record
                j_max,i_max:integer;
                analysis_bounds:rectangle_type;
                average,amplitude:real;
                graphics_descriptor_ptr:ptr;
                pixel_map_ptr:ptr;
                pixel_map_size:longint;{bytes}
                pixel_map_width:longint;{bytes}
                pixel_map_height:longint;
```

```
        display_window_ptr:ptr;
        display_window_open:boolean;
        pad_1:padding_byte;
        display_window_name:string;
        pad_2:padding_byte;
        marked_for_disposal:boolean;
        spare_ptrs:array [1..num_spare_ptrs] of ptr;
end;{record}
```

The first field of an image_type is j_max. It gives the number of rows in the image minus one. The second field is i_max, which gives the number of columns minus one. The rows of an image are numbered 0 to j_max from top to bottom (downwards, not upwards). The columns are numbered 0 to i_max from left to right. When you pass an image_type to an analysis routine, you must set i_max and j_max correctly first.

The analysis_bounds field is a rectangle_type. It defines the portion of the image that will be used by an analysis routine. The borders of the rectangle are included in the analyzable portion. The analysis routines analyze the intersection of all columns from analysis_bounds.left to analysis_bounds.right with all rows from analysis_bounds.top to analysis_bounds.bottom. Before you pass an image_type to an analysis routine, you must make sure its analysis_bounds are correct.

The average and amplitude fields are used by a few of the analysis routines to store the average and standard deviation of the image intensity within the analysis_bounds. Any analysis routine that uses these fields will calculate the average and standard deviation itself. You never have to calculate the average and amplitude for an analysis routine.

The graphics_descriptor_ptr is a 32-bit pointer to an operating-system dependent data structure that defines how the image will be displayed. In MacOS, graphics_descriptor_ptr points to an ‘off-screen graphics world’. The off-screen graphics world contains, among other things, a table relating eight-bit gray-scale values to twenty-four-bit RGB (red-green-blue) pixel values. Our analysis routines never use the graphics_descriptor_ptr.

The pixel_map_ptr points to the first byte of the image. Because each pixel is eight bits, the first byte is also the first pixel. Because the pixels are numbered from top to bottom and from left to right, the first pixel is the top-left pixel. The second pixel, at address pixel_map_ptr+1, is the pixel to the right of the first pixel. The simplest form of pixel map, which we call a ‘simple-format pixel map’, is a continuous sequence of image pixels. In this sequence, the first pixel in row n of the image is stored in the location following the last pixel of row $n-1$. The entire simple-format pixel map is takes up (i_max+1)*(j_max+1) bytes. When you pass an image_type to an analysis routine, you must set the pixel_map_ptr correctly first.

If you are displaying an image, your display might run faster if you arrange the image rows to suit the operating system’s display routines. In MacOS, this means adding bytes to each row until the number of columns is a multiple of four, which means that there can be up to three extra columns in the pixel map. An image_type specifies the number of columns in the pixel map with

pixel_map_width. The analysis routines use pixel_map_width every time they reference a pixel in the pixel_map. For completeness, image_type specifies the number of rows in the pixel map with pixel_map_height. At the moment, however, our analysis routines do not use to pixel_map_height. When you pass an image_type to an analysis routine, you must set pixel_map_width correctly. For forward compatibility, we ask that you set pixel_map_height as well. For simple image formats, you will set the width to i_max+1 and the height to j_max+1.

When you display an image, it will most likely be drawn in a window that can be moved, enlarged, reduced, or closed by the user. An image_type's display_window_ptr is a 32-bit pointer to an operating-system dependent data structure that defines such a window. Because the analysis routines have nothing directly to do with the display, they never refer to display_window_ptr.

A image_type's display_window_open field should be true if and only if the display window is open. The title of this window should be set to display_window_name, which is a Pascal string variable. The marked_for_disposal field is true when the user has requested that the display window be closed but the window has not yet been closed by the program. It is not used by any of the analysis routines. When the analysis routines want to dispose of an image, they do so right away by calling the dispose_image_ptr routine. Nor do the analysis routines refer to display_window_open, but they sometimes refer to display_window_name. We recommend that you set display_window_name before you call analysis routines, but you can set it equal to a null string if you like.

Finally, spare_ptrs is spare space in the image_type for whatever purpose you might find. The entries need not be pointers. Our intention is only to provide forty bytes of spare space (ten 32-bit pointers).

The following constants and types are used by the RASNIK analysis routines. The only type you need to study is rasnik_type.

```
const
        rasnik_max_squares_across=100;
        pattern_range=rasnik_max_squares_across div 2;

type
        mask_point_type=record x,y:real; end;
        square_type = record
                center_pp:pattern_point_type;
                center_intensity,center_whiteness:real;
                is_a_valid_square,is_a_code_square,is_a_pivot_square:boolean;
                pad_1:packed_byte;
                x_code,y_code:integer;
                display_outline:rectangle_type;
        end;{record}
        square_array_type=array [-pattern_range..+pattern_range,
                -pattern_range..+pattern_range] of square_type;
        square_array_ptr_type=^square_array_type;
```

```
rasnik_pattern_ptr_type=^rasnik_pattern_type;
rasnik_pattern_type=record {based upon pattern_type of image_types}
        origin:image_point_type;{pattern coordinate origin in image coordinates}
        rotation:real;{radians}
        sin_rotation,cos_rotation:real;
        x_width,y_width:real;{pixels}
        h_width,v_width:real;{pixels}
        error:real;{pixels rms in image}
        square_array_ptr:square_array_ptr_type;
        mask_orientation:integer;
        orientation_name:string[31];
        x_code_direction,y_code_direction:integer;
        valid:boolean;
        pad_1:packed_byte;
end;{record}
rasnik_type=record
        x,y:real;{micrometers}
        magnification:real;{image/object}
        rotation:real;{radians anticlockwize in image}
        error:real;{micrometers rms in mask}
        valid:boolean;
        pad_1,pad_2,pad_3:packed_byte;
end;{record}
```

As agreed by the ATLAS alignment collaboration, a RASNIK reading is the position in mask coordinates of the point in the mask that is projected onto the top-left corner of the top-left pixel in the CCD. The bottom-left edge of the bottom-left square in the mask is point (0,0). The x-direction is to the right, and the y-direction is upwards. In a rasnik_type, x and y give the mask coordinates in micrometers. The ‘magnification’ is the ratio of the square size on the CCD to the size on the mask. The ‘rotation’ is the anti-clockwise rotation of the mask pattern on the CCD in milliradians. The ‘error’ is an estimate of the rms error in the analysis routine’s measurement of x and y. If ‘valid’ is false, the RASNIK analysis failed.

Here are the types used by BCAM and BCAL analysis. Both these devices use the ‘centroid analysis’ routines, which simply find the light-centroid of an image or part of an image.

```
type
        centroid_type=record
                x,y:real;{micrometers centroid position}
                dx_dt,dy_dt:real;{micrometers per count}
                i,j:longint;{column and row closest to centroid}
                t:longint;{threshold in adc counts}
                valid:boolean;
                pad_1,pad_2,pad_3:packed_byte;
        end;{record}
```

The position of the centroid is in micrometers from the top-left corner of the top-left pixel in the image. The x-direction is left to right, and the y-direction is downwards. Our centroid-finding procedure ignores all pixels below a threshold intensity, and subtracts this threshold from the remaining pixels before it incorporates them into its weighted sum. The derivatives of x and y with respect to the threshold are dx_dt and dy_dt respectively. The fields i and j give the column and row of the pixel closest to the centroid. If 'valid' is false, the analysis failed.

Here are the constants and types used by the MDTQA X-ray routines. These routines find the shadows of wires in x-ray images, and combine the shadow positions from stereoscopic pairs of images to determine MDT wire positions in two dimensions.

```
const {for shadow arrangements}
        num_single_tube_shot_fiducials=4;
        num_single_tube_shot_shadows=5;
        num_single_tube_shots=2;

const {for shadow types}
        wire_name='(a wire)';
        tube_left_outer_name='(a left outer edge)';
        tube_right_outer_name='(a right outer edge)';
        tube_left_inner_name='(a left inner edge)';
        tube_right_inner_name='(a right inner edge)';
        max_num_shadows=20;

type {for shadow list}
        shadow_type = record
                rotation:real; {radians anticlockwize from vertical}
                position:real; {micrometers to right of ccd reference point}
                pixel_position:integer; {closest column or row}
                shadow_name:string;{gives the type of shadow}
                pad_1,pad_2:packed_byte;
        end;{record}
        shadow_list_type=record
                pixel_size_um:real;{size of ccd pixels perpendicular to shadows}
                ccd_reference_pixel:ccd_pixel_type;{reference pixel in ccd}
                min_shadow_separation:real;{micrometers}
                image_name:string;{display name of image}
                analysis_bounds:rectangle_type;{boundries for image analysis}
                num_shadows:integer;{number of shadows specified for the image}
                horizontal_shadows:boolean;{shadows run horizontally in image}
                valid:boolean;{for error tracking}
                shadows:array [1..max_num_shadows] of shadow_type;
        end;{record}

type {for shadow arangements}
```

```
point_type=record x,y:real; end;
line_type=record a,b:point_type; end;

single_tube_shot_fiducial_descriptor_type=record
        top_fiducial_separation,bottom_fiducial_separation:real;
                {um separation of wires in top and bottom pairs}
        top_fiducial_offset,bottom_fiducial_offset:real;
                {um offset of top and bottom pairs from x-y origin}
        center_line_polar_angle:real;
                {polar angle of the vector running from the mid-point of the bottom
                fiducial pair to the mid-point of the top fiducial pair (the polar angle
                is anticlockwize from the x-axis)}
        sense_of_film:real;
                {>0 if polar angle of left edge of image is greater than polar angle of
                right edge of image (angle anticlockwize from x-axis)}
        valid:boolean;
        pad_1,pad_2,pad_3:packed_byte;
end;{record}

single_tube_shot_type=record
        image_angle:real;
                {ccd or film angle, degrees clockwize from vertical}
        fiducial_descriptor:single_tube_shot_fiducial_descriptor_type;
                {used to calculate fiducial_positions}
        fiducial_positions:array [1..num_single_tube_shot_fiducials] of point_type;
                {um in x-y coords}
        shadow_list:shadow_list_type;
                {specifies positions and rotations of shadows}
        source_position:point_type;
                {x-ray source position, um in x-y coords}
        film_position:point_type;
                {left edge of image, um in x-y coords}
        wire_to_source:line_type;
                {line passing through wire and source}
        valid:boolean;
                {most recent manipulation was successful}
        pad_1,pad_2,pad_3:packed_byte;
end;{record}

single_tube_type=record
        shots:array [1..num_single_tube_shots] of single_tube_shot_type;
                {stereo image data}
        wire_position:point_type;
                {tube wire position, um in x-y coords}
        valid:boolean;
                {most recent manipulation was successful}
```

```
            pad_1,pad_2,pad_3:packed_byte;
        end;{record}
```

As you can see, the MDTQA data structures are complicated. We tried to put adequate comments in the Pascal definitions, but we invite you to contact us directly if you attempt to use the routines. We will be happy to answer your questions.

ANALYSIS.DLL Export Routines

The Pascal listing below is taken directly from our analysis.dll source code. It declares the routines that analysis.dll exports to other programs. The name of each routine has the suffix '_dll'. Each routine is marked with the 'dllexport' and 'stdcall' qualifiers, which indicate that it is for export and that it obeys the Win32 standard-call convention. The routines take simple parameters. Instead of an image_type, they take a pointer to an image_type.

```
{
        sum_dll returns the sum of two integers.  It is a
        simple function to help programmers connect to
        analysis.dll
}
function sum_dll(a,b:integer):integer;
        stdcall;dllexport;

{
        multiplication_dll sets c equal to the produce of
        a and b.  It is a simple procedure to help programmers
        connect to analysis.dll
}
procedure multiplication_dll(a,b:integer; var c:integer);
        stdcall;dllexport;

{
        write_string_dll writes a string to the consol and to the message
        board. You can use it to test your c-to-pascal string converter.
        If your converter works, the string you pass to this routine will
        come out right.
}
procedure write_string_dll(s:string);
        stdcall;dllexport;

{
        subtract_image_dll subtracts the intensity of image oip_2^ from
        the intensity of oip_1^ ('oip' is 'original image pointer'), adds
```

```
        a constant to avoid negative intensities, and returns the difference
        image  in nip^. If nip is nil when passed to subtract_image_dll, the
        routine allocates space for a new image and returns the pointer to the
        new image_type in nip. If nip is not nil, the routine assumes that
        nip points to an image of the correct dimensions to receive the
        difference image.
}
procedure subtract_image_dll (oip_1,oip_2:ptr;var nip:ptr);

{
        centroid_analysis_dll finds the centroid of intensity
        in the analysis_bounds of the specified image_type.
        pixel_width_um and pixel_height_um are the
        dimensions of the ccd pixels in micrometers.  image_ptr
        points to an image_type.  threshold is subtracted from the
        pixel intensity during analysis.  centroid_ptr is a
        pointer to an existing centroid_type.  The centroid
        position is recorded in this centroid_type.
        This routine is for BCALs and BCAMs.
}
procedure centroid_analysis_dll (image_ptr:ptr;
        threshold:integer;
        pixel_width_um,pixel_height_um:real;
        centroid_ptr:ptr);
        stdcall;dllexport;

{
        centroid_test_image_ptr_dll returns a pointer to
        a newly-generated image_type.  The image_type's
        pixel map contains a white spot on a black background.
}
function centroid_test_image_ptr_dll:ptr;
        stdcall;dllexport;


{
        centroid_write_dll writes the contents of a centroid_type to
        the text window.  centroid_ptr is a pointer to the centroid_type.
}
procedure centroid_write_dll (centroid_ptr:ptr);
        stdcall;dllexport;

{
        centroid_width returns the width, in pixels, of the light spot in image_ptr^. The
        edges of the spot are where its intensity rises above the threshold specified in
        centroid.
```

```
}
function centroid_width_dll (image_ptr,centroid_ptr:ptr):integer;
        stdcall;dllexport;

{
        centroid_height is the vertical equivalent of centroid_width.
}
function centroid_height_dll (image_ptr,centroid_ptr:ptr):integer;
        stdcall;dllexport;

{
        rasnik_analysis_dll obtains a rasnik measurement from
        the specified image_type.  It uses only the pixels within
        the analysis_bounds of the image_type.  pixel_width_um
        and pixel_height_um give the dimensions of the pixels
        in micrometers.  square_width_um gives the mask
        square width, which is the same as its height.  rasnik_ptr
        points to an existing rasnik_type in which the rasnik
        measurement will be recorded. This measurement gives the
        coordinates, in the mask coordinate system, of the point
        in the mask that is projected onto the top left corner of
        the image.
}
procedure rasnik_analysis_dll (image_ptr:ptr;
        square_width_um:real;
        pixel_width_um,pixel_height_um:real;
        rasnik_ptr:ptr);
        stdcall;dllexport;

{
        rasnik_analysis_centered_dll is like rasnik_analysis_dll,
        except that it gives the coordinates, in the mask coordinate system,
        of the point in the mask that is projected onto the center of the
        image analysis bounds.
}
procedure rasnik_analysis_centered_dll (image_ptr:ptr;
        square_width_um:real;
        pixel_width_um,pixel_height_um:real;
        rasnik_ptr:ptr);
        stdcall;dllexport;

{
        rasnik_analysis_ccd_centered_dll is like rasnik_analysis_dll,
        except that it gives the coordinates, in the mask coordinate system,
        of the point in the mask that is projected onto the center of the
        image.
```

```
}
procedure rasnik_analysis_ccd_centered_dll (image_ptr:ptr;
        square_width_um:real;
        pixel_width_um,pixel_height_um:real;
        rasnik_ptr:ptr);
        stdcall;dllexport;

{
        rasnik_analysis_general_dll will perform all available forms
        of rasnik analysis, as specified by its numerous parameters. The
        approximate parameter selects approximate or accurate analysis.
        Approximate analysis is faster. If orientation_code is zero, the
        analysis selects the most likely of the four possible mask orientations.
        If the code is one, two, three, or four, the analysis restricts itself
        to that particular orientation. If reference_code is zero, the analysis
        refers is rasnik reading to the top-left corner of the ccd. If it's one,
        the analysis uses the center of the analysis bounds. It it's two, the
        analysis uses the center of the ccd. There are dummy integer parameters
        to allow for expansion. Set these to zero to disable whatever features
        may be use them later.
}
procedure rasnik_analysis_general_dll (image_ptr:ptr;
        square_width_um:real;
        pixel_width_um,pixel_height_um:real;
        approximate:boolean;
        reference_code,orientation_code,dummy_1,dummy_2,dummy_3:integer;
        rasnik_ptr:ptr);
        stdcall;dllexport;

{
        rasnik_test_image_ptr_dll returns a pointer to
        a newly-generated image_type.  The image_type's
        pixel map contains a pattern of black and grey squares.
}
function rasnik_test_image_ptr_dll:ptr;
        stdcall;dllexport;

{
        rasnik_write_dll writes the contents of a rasnik_type to
        the text window.  rasnik_ptr is a pointer to the rasnik_type.
}
procedure rasnik_write_dll (rasnik_ptr:ptr);
        stdcall;dllexport;

{
        shadows_locate_dll finds shadows in the analysis_bounds
```

```
        of an image_type.  image_ptr points to the image_type.
        shadow_list_ptr points to a shadow_list_type in which the
        shadows the routine should look for are described.  The
        results of analysis are stored in the same shadow_list_type.
        This routine is for MDTQA by X-Ray images.
}
procedure shadows_locate_dll(image_ptr:ptr;shadow_list_ptr:ptr);
        stdcall;dllexport;

{
        shadows_test_image_ptr_dll returns a pointer to
        a newly-generated image_type.  The image_type's
        pixel map contains five vertical black stripes.
}
function shadows_test_image_ptr_dll:ptr;
        stdcall;dllexport;

{
        shadows_update_list_dll gives the user the opportunity
        to change the main parameters of a shadow_list_type
        with the keyboard.  shadow_list_ptr points to the
        shadow_list_type.  If the user does not wish to change
        a parameter, she simply presses the carriage return.
}
procedure shadows_update_list_dll (shadow_list_ptr:ptr);
        stdcall;dllexport;

{
        shadows_write_dll writes shadow positions and rotations
        to the text window.
}
procedure shadows_write_dll (shadow_list_ptr:ptr);
        stdcall;dllexport;

{
        write_image_characteristics writes the average, amplitude,
        maximum, and minimum intensities of an image to the text
        window.  It also writes out the image dimensions and the
        analysis borders.
}
procedure write_image_characteristics_dll (image_ptr:ptr);
        stdcall;dllexport;
```

We intend to extend the list of exported routines to meet your needs. Please do not hesitate to ask us for additions.

ANALYSIS.DLL Import Routines

All the routines imported by analysis.dll must obey the Win32 standard-call convention. Some routines are essential to the analysis library, others merely give the client program the opportunity to display the results of analysis on the screen.

The following routines are essential to the analysis library's RASNIK and MDTQA routines, but not to the centroid routines.

```
procedure dispose_ptr (var p:ptr;id:string);
procedure dispose_image_ptr (var image_ptr:ptr);
function new_ptr (size:longint;id:string):ptr;
function new_image_ptr(i_size,j_size:longint):ptr;
function random_0_to_1:real;
function valid_image_ptr (image_ptr:ptr):boolean;
```

The following routines are essential to the analysis library's data-entry routines, such as shadows_update_list.

```
procedure get_integer (variable_name:string;var value:integer);
procedure get_longint (variable_name:string;var value:longint;base,digits:integer);
procedure get_longreal (variable_name:string;var value:longreal);
procedure get_real (variable_name:string;var value:real);
procedure get_string (variable_name:string;var value:string);
procedure update_boolean (variable_name:string;var value:boolean);
procedure update_integer (variable_name:string;var value:integer);
procedure update_longint(variable_name:string;var value:longint;base,digits:integer);
procedure update_real (variable_name:string;var value:real);
procedure update_string (variable_name:string;var value:string);
```

When an analysis routine recognizes that it has failed, it calls the following routines to report its failure.

```
procedure stack_trace (callee_name,caller_name:string);
procedure warning (warning_description:string);
```

If you implement the following routines properly, the analysis routines will display their results in color within the image windows. But none of these display-related routines are essential. They can all be replaced by dummy procedures, or functions that return zero or nil values. A routine might call show_image to draw its image in the foreground, but if show_image does nothing, the analysis routine will never know the difference. Later, the routine might call display_image_graph, and this procedure can do nothing as well.

```
procedure display_ccd_ellips (image_ptr,rectangle_ptr:ptr;color:integer;icp:ptr);
procedure display_ccd_graph (image_ptr,graph_ptr:ptr;num_points,color:integer;icp:ptr);
procedure display_ccd_line (image_ptr,line_ptr:ptr;color:integer;icp:ptr);
procedure display_ccd_rectangle (image_ptr,rectangle_ptr:ptr;color:integer;icp:ptr);
procedure display_image_graph (image_ptr,graph_ptr:ptr;num_points,color:integer;icp:ptr);
procedure display_i_profile (image_ptr,profile_ptr:ptr;color:integer;icp:ptr);
procedure display_j_profile (image_ptr,profile_ptr:ptr;color:integer;icp:ptr);
procedure display_real_graph (image_ptr,graph_ptr:ptr;num_points,color:integer;icp:ptr);
function front_image:ptr;
procedure hide_image(image_ptr:ptr);
procedure pause (seconds:real;procedure check);
procedure show_image(image_ptr,icp:ptr);
function system_timer:longreal;
procedure xor_display_ccd_line (image_ptr,line_ptr,icp:ptr);
```

The names of these display routines give some indication as to their function, but if you want to implement them yourself, you will need to look at the Pascal code we wrote to implement them in MacOS.

The Default OS.DLL

We provide a version of os.dll that we call our ‘default os.dll’. Our default os.dll is defined in standard Pascal. You are welcome to a copy of the source code. All the routines exported by os.dll are available to the analysis library and the client program. The client program must load both analysis.dll and os.dll into memory.

Our default os.dll implements the following routines using the Pascal new() and dispose() procedures.

```
function new_ptr (size:longint;id:string):ptr;
procedure dispose_ptr (var p:ptr;id:string);
```

The new_ptr routine allocates ‘size’ contiguous bytes in memory and returns a pointer to the first of these bytes. It stores this pointer in a table. The dispose_ptr routine later uses this table look up the size of the block associated with the pointer it is supposed to dispose of.

```
function new_image_ptr(i_size,j_size:longint):ptr;
procedure dispose_image_ptr (var image_ptr:ptr);
```

The new_image_ptr routine, as implemented in the default os.dll, allocates space for a simple-format pixel map of the specified dimensions, and allocates space for a new image_type

variable. It fills in the fields of the image_type, and returns a pointer to the completed record. It performs its memory allocations by calling new_ptr. The dispose_image_ptr routine calls dispose_ptr to de-allocate the image_type and its pixel map.

```
function valid_image_ptr (image_ptr:ptr):boolean;
```

Our default os.dll valid_image_ptr returns false if and only if image_ptr is nil. If you have the analysis routines working together with an image display, this routine should check to see if the user has closed an image's display window. If so, it should return false. It may return false for other display-dependent reasons as well.

```
function random_0_to_1:real;
```

The random_0_to_1 routine is used to generate random pixels for approximate calculations of image properties. In the default os.dll, we have no access to the system clock or random number generator, so we use a pseudo-random series of numbers.

```
procedure stack_trace (callee_name,caller_name:string);
procedure warning (warning_description:string);
procedure get_message(s:string);
```

When one procedure calls another, and the second procedure fails, the first procedure can report this failure with stack_trace. The first procedure gives its own name in 'caller_name', and the name of the other procedure in 'callee_name'. The warning procedure reports a warning described by the 'warning_description' string. Both procedures write to the console window, if the client application has opened one, and to a message board maintained by os.dll itself. You can get messages from the message board by calling get_message. It returns messages one line at a time, in the order in which they were received by the message board. When there are no more messages left, it returns an empty string.

```
function front_image:ptr;
```

The 'front_image' procedure is supposed to return a pointer to the image that is in the foreground of the display. In our default os.dll, front_image simply returns a nil pointer.

All the display routines exported by our default os.dll are implemented with dummy procedures. The 'display routines' are those whose names begin with 'display', as well as the show_image and hide_image routines.

```
procedure pause (seconds:real;procedure check);
function system_timer:longreal;
```

The system_timer and pause routines are called by the analysis routines to provide diagnostic services to the user. In our default os.dll, ‘pause’ is a dummy procedure, and ‘system_timer’ returns zero.

```
procedure read_image_file(var image_ptr:ptr;file_name:string);
procedure write_image_file(image_ptr:ptr;file_name:string;file_creator,file_type:string);
```

Both read_image_file and write_image_file are implemented fully in our default os.dll. Neither is necessary for the analysis library, but you will find them useful when you are testing your client program. Beware, however, that both routines work only with simple-format pixel maps. Also, you must pass them a string parameter, which requires that you translate a string from C, if you are working in C, into the Pascal format.

Image Storage Formats

We use our own ‘simple-format image files’ for storing images on disk. A simple-format image file is a simple-format pixel map on disk, with the first twelve bytes overwritten by a block of header information.

```
type {for file i/o}
        image_file_format=(simple_format);
        image_file_header_type=record
                case image_file_format of
                        simple_format:(
                                j_max,i_max:integer;
                                analysis_bounds:rectangle_type);
        end;{record}
        image_file_header_ptr_type=^image_file_header_type;
```

The image_file_header_type describes the header at the beginning of a simple-format image file. This header is twelve bytes long and over-writes the first twelve pixels of the image. Apart from the header, the simple-format image file is nothing more than a simple-format pixel map on disk. The size of the file is exactly (i_max+1)*(j_max+1) bytes. The header records j_max, i_max, and the analysis_bounds.

For PC-users, there is a complication to writing and reading the image file header. The integers of the header must be stored in reverse-byte order. That is, they must be stored with the high-byte before the low-byte. This is natural on the Macintosh and in VME, but it is not natural to

the PC. On a PC, you must flip the header bytes before you write them to disk, and flip them after you have read them from disk.

Actual Names of DLL Routines

If wsere to use the Code Warrior Pascal compiler, you would find that you can refer to the routines you import from analysis.dll and os.dll by the names we gave them in our own Pascal code. For example, if you want to use the routine we listed above as 'sum_dll', you could declare a DLL import routine of the same name. It turns out, however, that the sum_dll code segment has two names defined in the actual DLL and neither of these is 'sum_dll'. One name is '_sum_dll@8' and the other is '__imp__sum_dll@8' (note the double-underscores).

Our Pascal names do not obey the Win32 naming conventions. Code Warrior adds characters to our Pascal names when it makes a DLL, and so brings the names into compliance automatically, leaving our Pascal code free of the double-underscores and @-signs. According to the Win32 literature, '@8' means a routine requires 8 bytes of stack space. (We have not yet figured out why sum_dll requires eight bytes of stack space. So far as we can tell, it needs only six bytes: two for each of the input parameters, and two for the output. But it may be that the result of a function is passed via a pointer, which would be four bytes.)

Here are the shorter names of the analysis.dll export routines. The longer names are obtained from these by adding '__imp_' to the shorter name. Our sum_dll routine can be called '_sum_dll@8' or '__imp__sum_dll@8'.

_centroid_analysis_dll@20
_centroid_test_image_ptr_dll@0
_centroid_write_dll@4
_multiplication_dll@12
_rasnik_analysis_dll@20
_rasnik_analysis_centered_dll@20
_rasnik_analysis_general_dll@48
_rasnik_test_image_ptr_dll@0
_rasnik_write_dll@4
_shadows_locate_dll@8
_shadows_test_image_ptr_dll@0
_shadows_update_list_dll@4
_shadows_write_dll@4
_sum_dll@8
_write_image_characteristics_dll@4
_write_string_dll@4

Here are the shorter names of the os.dll export routines.

_display_ccd_ellips@16
_display_ccd_graph@20
_display_ccd_line@16
_display_ccd_rectangle@16
_display_i_profile@16
_display_image@8
_display_image_graph@20
_display_j_profile@16
_display_real_graph@20
_dispose_image_ptr@4
_dispose_ptr@8
_front_image@0

_get_boolean@8
_get_integer@8
_get_longint@16
_get_longreal@8
_get_message@4
_get_real@8
_get_string@8
_hide_image@4
_new_image_ptr@8
_new_ptr@8
_pause@12
_random_0_to_1@0
_read_image_file@8
_show_image@8
_stack_trace@8
_system_timer@0
_update_boolean@8
_update_integer@8
_update_longint@16
_update_real@8
_update_string@8
_valid_image_ptr@4
_warning@4
_write_image_file@16
_xor_display_ccd_line@12

How to Connect to the DLL

In the dll.zip archive, we include dll.cpp, a Visual C++ console program that connects to analysis.dll. This program loads the DLL into memory and locates the example routines before assigning them local names. Other compilers simplify the source code by requiring that you supply a DLL descriptor file. Such compilers provide the DLL descriptor file when you use them to generate your own DLLs. So far as we can tell, there is a standard form for DLL descriptors. Our compiler, Code Warrior by Metrowerks, requires DLL descriptor files. The one for analysis.dll is called analysis.dll.lib, and the one for os.dll is called os.dll.lib. We include both in dll.zip. Be sure that your compiler is set to generate standard-call Win32 routines, otherwise the order in which they put parameters on the stack may not agree with the order used by the DLLs, and your computer will crash.

File-Controlled Operation of PCAPP.EXE

Werner Riegler of Harvard pointed out to me that we could provide analysis to Windows programs by means of a disk-based file communication. When you run pcapp.exe, you will see 'file-controlled operation' in the main menu. When you select this option, the program waits for a file called 'command.txt' to be placed in the same folder as pcapp.exe. This file should be text only. Its first line must be one of the following text strings, with no trailing or leading spaces.

String	Meaning
rasnik	perform rasnik analysis
centroid	perform centroid analysis
exit	exit file-controlled operation

Table: Commands Supported by File-Controlled Operation

If the command is anything other than 'exit', the next line in command.txt is the name of an image file sitting in the same directory as pcapp.exe. The program reads this image into memory. Then it reads parameters from command.txt according to the analysis that is to be performed. Each parameter should be on a separate line, and the final parameter must have no line feed after it. For boolean variables, you write 'true' and 'false'. For rasnik analysis, it reads the mask square width in μm, the pixel width in μm, the pixel height in μm, whether the analysis should be approximate, whether the analysis should be centered, and the orientation code. For centroid analysis, it reads the threshold, the pixel width in μm, and the pixel height in μm. Once pcapp has read these parameters, it analyzes the image. It writes its results to the text window, and also to a file called results.txt. It waits until results.txt has been deleted. Then it starts looking for command.txt again.

Your windows program should write command.txt and wait for results.txt. When results.txt appears, your program deletes command.txt, reads results.txt, and then deletes results.txt as well. Your program may have to try a couple of times to delete results.txt, because pcapp is opening the same file every now and then, checking to see if it is still there. If your program tries to delete results.txt while pcapp is reading it, the operating system will not allow the deletion to take place.

The pcapp_fco.exe program starts up in file-controlled operation, and quits when you send it the exit command.

Set-Up Procedure

When you write a program that uses our DLLs, we recommend the following procedure. Try to load os.dll into your program, and call the random_0_to_1 function, which returns a real number and takes no paramters. If you are successful, try to load analysis.dll as well, and call rasnik_test_image_ptr_dll, which returns a pointer and takes no parameters. Be sure that your

program loads both os.dll, and analysis.dll, and loads os.dll before analysis.dll, because analysis.dll calls upon routines in os.dll, and your program may have trouble loading analysis.dll unless os.dll is already loaded. Once you have these two parameter-less routines working, move on to things like sum_dll, and multiply_dll, to see if you can pass parameters to the dlls in the correct order. Call us or e-mail us if you have any questions, we will be glad to help you.

Conclusion

We can provide both Macintosh and PC users with libraries of image analysis routines. There are hardly any Macintosh users left in the ATLAS, so we doubt we will be called upon to provide any Macintosh libraries. This manual described the DLLs we provide for PC users. These DLLs are compatible with Windows 95, Windows 98, and Windows NT. We tested them on all three operating systems. Your data acquisition program can link to them at execution-time. When you want to incorporate the latest version of our analysis library into your data acquisition system, you replace the old version with the new one, and re-start your program.

We provide two DLLs. One contains the image analysis routines, which are operating-system independent, and the other contains fully functional memory allocation and display routines, and dummy graphical routines. You must replace the latter library with one of your own if you want the analysis routines to display their results on the screen.