



Visionscape Perl Script Custom Tool Programmer's Manual

v8.0.0, August 2016

Copyright ©2016
Microscan Systems, Inc.
Tel: +1.425.226.5700 / 800.762.1149
Fax: +1.425.226.8250

ISO 9001 Certified
Issued by TÜV USA

All rights reserved. The information contained herein is proprietary and is provided solely for the purpose of allowing customers to operate and/or service Microscan manufactured equipment and is not to be released, reproduced, or used for any other purpose without written permission of Microscan.

Throughout this manual, trademarked names might be used. We state herein that we are using the names to the benefit of the trademark owner, with no intention of infringement.

Disclaimer

The information and specifications described in this manual are subject to change without notice.

Latest Manual Version

For the latest version of this manual, see the Download Center on our web site at:
www.microscan.com.

Technical Support

For technical support, e-mail: helpdesk@microscan.com.

Warranty

For current warranty information, see: www.microscan.com/warranty.

Microscan Systems, Inc.

United States Corporate Headquarters

+1.425.226.5700 / 800.762.1149

United States Northeast Technology Center

+1.603.598.8400 / 800.468.9503

European Headquarters

+31.172.423360

Asia Pacific Headquarters

+65.6846.1214

Contents

PREFACE	Welcome v Purpose of This Manual v Manual Conventions vi
CHAPTER 1	The Big Picture 1-1 What is Perl? 1-1 Adding a Perl Script to Visionscape 1-2 How Do I Use a Perl Script in My Visionscape Job? 1-2 Differences Between Custom Step and Custom Vision Tool 1-3
CHAPTER 2	Brief Overview of the Perl Language 2-1 Variables and Arrays 2-1 Conditional and Loop Statements 2-3 Using Functions/Subroutines 2-4
CHAPTER 3	Anatomy of a Visionscape Perl Script 3-1 Visionscape Step Interface Subroutines 3-1 Using Visionscape Packages in Your Script 3-3 Adding Input and Output Datums 3-4 Getting and Setting Datum Values for Simple Types 3-7 Getting and Setting Complex Datum Types 3-9 Creating a Complete Script 3-10 Creating Your Own Scripts 3-12

CHAPTER 4	Examples 4-1 Determining If An Input Datum Is Connected 4-1 Preventing Your Script from Running in the Wrong Type of Step 4-2 Hiding the ROI of a Custom Vision Tool 4-3 Setting The Pass/Fail Status Of The Step 4-3 Localizing Your Variables When Using Multiple Instances Of Your Script 4-3 Drawing In The Buffer 4-5 Changing Another Step's Datum Value 4-7 Working With Point Lists 4-8 IO Access 4-10 Accessing A Blob Tree 4-11
CHAPTER 5	Measurements and Coordinate Transforms 5-1 Transforming Points and Lines 5-2 World Space 5-6
CHAPTER 6	Perl Reference 6-1 Introduction 6-1 Perlutil — Perl Utility Package 6-3 Composite-Datum-Step Packages (Common Functions) 6-18 Datums 6-23 Vision Library 6-41 Morphology 6-53 Utility Packages 6-100 Line-by-line Analysis of a Script 6-106 Usage Examples 6-112 References 6-140 Helpful Programming Information 6-143
APPENDIX A	Open Source Software Used in Visionscape A-1 Warranty Regarding Further Use of the Open Source Software A-2 Open Source Software Used A-2 Open Source Software Licenses A-2

Welcome

Purpose of This Manual

The chapters in this manual contain the following information:

- Chapter 1 — Describes what Perl is, how to add scripts to Visionscape, how to use a Perl script, and the differences between the Custom step and the Custom Vision tool.
- Chapter 2 — Describes variables, arrays, conditional statements, loop statements, functions and subroutines, packages and modules, and debugging a Perl script.
- Chapter 3 — Describes subroutines, adding input and output datums, getting and setting simple and complex types, and creating your own scripts.
- Chapter 4 — Provides practical examples of common tasks.
- Chapter 5 — Describes measurements and coordinates transforms.
- Chapter 6 — Describes perlutil, common functions, datums, the vision library, morphology, utility packages, and provides sample scripts and usage examples and other helpful programming information.

Manual Conventions

The following typographical conventions are used throughout this manual.

- Items emphasizing important information are **bolded**.
- Menu selections, menu items and entries in screen images are indicated as: Run (triggered), Modify..., etc.

The Big Picture

Visionscape® allows you to build vision inspections by using our simple and flexible Step Tree based programming model via FrontRunner™. In most cases, you will find the functionality you require already exists in the library of Steps that are installed with Visionscape®. In some cases, however, you may find that you need to add some custom functionality to your inspection. For this reason, we provide the Custom Step and the Custom Vision Tool, which are Visionscape® steps that can run a script written in the Perl language.

A Perl script can be created using any text editor like Notepad or Wordpad in Windows. This script must conform to certain rules that we will cover in the following chapters, but the Custom Step and Custom Vision Tool hide most of the details from you, allowing you to focus on the specific task in your script.

What is Perl?

Perl is an interpreted language that is popular in the Unix world. It has a syntax that is similar to C, but is much simpler to use. For a scripting language, it has excellent performance. We will cover the basic syntax of the language later on, but there are many excellent books on Perl that you can find at any major bookstore, should you wish to learn more.

Adding a Perl Script to Visionscape®

Before learning how to create your own Perl scripts, you should first understand how to add a Perl script to Visionscape®. In order for a newly created Perl script to be loaded, you must perform the following steps:

1. Copy your script to the following folder (on whatever drive you installed Visionscape®):

`\\Vscape\Jobs\perlmod\`

2. Let's assume your script file is named "myscript.pm". In the "perlmod" folder, there is a text file named "perlscr". Open it in any text editor, and add the following line to the end of the file:

`require myscript;`

This line specifies the file name of your script, without the .PM extension. It is case sensitive, and the ';' at the end of the line is required.

The "perlscr" file is the list of all scripts that will be loaded by Visionscape®.

Note: You can copy all the scripts you want into the "perlmod" folder, but Visionscape® will not load them unless you add their names to "perlscr". Because of the differences in the way Windows and an embedded device such as a smart camera handle case in file names, we recommend that you use lower case for your perl script file names.

How Do I Use a Perl Script in My Visionscape® Job?

1. Decide which Perl script you wish to use. You can create your own custom Perl script, or use one of the standard scripts that are installed with Visionscape®.
2. Insert either a Custom Step or Custom Vision Tool into your Job.
3. The Avail Package Scripts datum in the step provides you with a drop-down list of all of the currently installed Perl scripts. Use this list to select the script you want to run. By default, this datum will always

select the “none” script, which is a dummy script that provides no functionality.

After you select your desired script from the list, the datum page will update automatically to show you all of the input datums that were created by the script.

At this point, you will use the Custom Step or Custom Vision tool like any other step within Visionscape®.

Differences Between Custom Step and Custom Vision Tool

- Custom Step:
 - Has no connection to the image buffer and no ROI.
 - Use this step when you are going to perform simple logical operations for which you don’t need an ROI or a connection to the image buffer.
- Custom Vision Tool:
 - Has a connection to the image buffer it was inserted into. This means you can draw graphics into that buffer.
 - Provides an ROI that you can drag around and resize, just like any other vision step.
 - Use this step whenever you wish to draw graphics, perform image processing operations in the image buffer, or if you need to provide an ROI for your user.

Note: The ROI can be hidden, so if you wish to draw in the buffer, but don’t want the ROI, you can simply hide it.

Brief Overview of the Perl Language

This chapter covers the basics of the Perl language so that you can get started writing scripts immediately. The language is actually quite simple, but it also contains many powerful and more complex features. We will not explore the more powerful aspects of the language here since, in most cases, you really only need to understand the most basic aspects of the language in order to create a Visionscape script. Perl's syntax is similar to that of "C", but there are many important differences that we cover here.

TABLE 2-1.

Character	Description
;	Must terminate each line of code in a Perl script
#	Must precede a comment

Variables and Arrays

- All scalar variables must start with a '\$', and all arrays must start with a '@'. This guarantees your variable names won't conflict with any Perl keywords. For example:

```
$MyVar = 10; # assigns value of 10 the variable named $MyVar
@MyArray = (1,2,3,4,5) #initialize 5 element array
$MyArray[3] = 46; #access an individual array element
($var1, $var2) = @MyArray #read 1st two elements of array
```

Notice that the array variable is referred to using the '@' symbol when referring to the array as a whole, but is referenced using the '\$' when referring to a single element of the array. When you refer to a scalar value, always use the \$.

- You don't have to declare your variables (you can't actually); just use them whenever and wherever you need them.
- You don't have to worry about the data type of a variable. Perl variables can hold integers, floating point values, strings, etc.
- By default, variables have Global scope, meaning that if you use a variable in one function, its value will still be available for you to use in other functions. If you wish to force the scope of your variable to be local to the current function, then use the 'my' keyword the first time you use it:

```
my $myVar = 10;
```

- All variable names are case sensitive, so pay attention to what you're typing when writing scripts. The following common mistake will cause a warning message from the Perl parser:

```
$myVar = 10;
++$myvar; # $myVar is NOT the same as $myvar
```

Perl will tell you you're attempting to use an Uninitialized value in this case.

- Variables can be incremented and decremented using "C" syntax:

```
$v = 2;
++$v; # $v now equals 3
--$v; # $v now equals 2
$v += 4; # $v now equals 6
$v /= 2; # $v now equals 3
$v |= 5; # $v now equals 7
```

- Table 2–2 lists the standard operators.

TABLE 2–2. Standard Operators

Operator	Description
>> and <<	Bit Shifting
&	Bitwise AND
	Bitwise OR
&&	Conditional AND
	Conditional OR
^	Exponent
%	Modulo
!	Not
==	Numerical Comparison
eq	String Comparison
.	String Concatenation

Conditional and Loop Statements

- If Syntax

```

if(expression)
{   # the curly braces are ALWAYS required

}
elsif(expression) #Yes, it's spelled 'elsif' NOT 'elseif'
{
}
else
{
}

```

For example:

```

$myString = "";
if($v > 10)
{
    $myString = "V is greater than 10";
}
elsif($v <= 10 && $v >= 0)

```

```
{
    $myString = "V is less than 11 AND not negative";
}
else
{
    $myString = "V must be negative number";
}
```

- For loop Syntax

```
for($v=0; $v <= 10; ++$v)
{
    # curly braces are always required
}
```

- Foreach Syntax

```
## Can be used to loop through an array like this:
foreach $user (@users)
{
    print "This User name = $user";
}
```

- While loop Syntax

```
while(expression)
{
    #curly braces are always required
}
```

- Use the **next** operator to skip to the end of your current loop operation.
- Use the **last** operator to break out of the current loop block.

Using Functions/Subroutines

- Defining a Subroutine:

```
sub MyFunction
{ #note that you don't specify the arguments you want to pass in
}

```

- You can pass as many arguments as you wish to your subroutine, and you can pass a different number each time. You read the arguments using the global array `@_` (individual arguments in the array can be referenced via `$_`).

```
MyFunction (5,6,7); #call our function, and pass in 3 variables

sub MyFunction # inside the function, extract the 3 args from @_
{
    my($var1, $var2, $var3) = @_; # extract them all at once...
    my $var1 = $_[0]; # ...or one at a time
    my $var2 = $_[1];
    my $var3 = $_[2];

    #if the number of arguments varies, use 'foreach'
    # to loop through them
    foreach $foo (@_)
    {
        $sum += $foo; #accumulate the sum of all arguments
    }
}
```

- To force a particular number of arguments, declare your sub like this:

```
sub MyFunction($$$)#cause this function to accept only 3 scalar values
{
    # extract them in the same manner as above
    my($var1, $var2, $var3) = @_;
}
```

- Return variables or arrays using 'return' :

```
sub SumMinMax
{
    my $Max = 0;
    my $Min = 999999;
    my $sum = 0;

    #loop through all args, accumulate a sum, min and max
    foreach $foo (@_)
    {
        $sum += $foo;

        if($foo > $Max)
        {
            $Max = $foo;
        }
        if($foo < $Min)
        {
            $Min = $foo;
        }
    }
    #return the sum
    return $sum, $Min, $Max;
}

#Call our Sum function, the variables
```

```
# $sum, $min and $max will be loaded with the returned values
my($sum, $min, $max) = Sum(1,2,3,4,5,6);
```

Packages and Modules

- A Module is a file that defines one or more Packages. These files generally use the extension “*.pm”.
- A Package is a block of code that acts as an object definition, and will have its own individual namespace when running. This means that all variables used in a Package are private to that Package.
- When you write a Perl script for use in the Visionscape environment, you will create a new Package, and save that Package inside of a Module.
- In order to access functionality in other packages, you must use the ‘**use**’ keyword at the top of your module. For example, all Perl scripts used with a Custom Step or Custom Vision Step must include the ‘perlutil’ package, so include it using the following line:

```
use perlutil;
```

- Call functions in your included packages by using the package name, followed by a double colon, followed by the function name:

```
perlutil::clear_datum_lists();
```

Debugging

There is no debugger for you to use when testing your scripts in Visionscape. The easiest way to debug is by placing ‘print’ commands at key locations in your code (the output of the print commands will be displayed in the avpbackplane debug display window):

```
$MyString = "Testing 1,2,3";
$MyVal = 1024;
```

```
#If you use double quotes with print, Perl knows that
# $MySring refers to the variable
print "MyString = $MyString\n";#Prints "MyString = Testing 1,2,3"
```

```
print "MyVal = $MyVal\n"; #Prints out "MyVal = 1024"
```



```
#If you use single quotes, no attempt is made  
# to interpret variable names  
print 'MyString = $MyString';#Prints out "MyString = $MyString"
```

You can also use 'printf' if you prefer; the syntax is the same as in C.

Note: Always remove all print statements when done with your script.
Otherwise, performance will degrade.

Anatomy of a Visionscape Perl Script

Your Perl script must conform to the same interface that all standard Steps in Visionscape conform to. We have defined a standard set of subroutines that the Visionscape framework will call within your script. You'll need to add one or more of those subroutines to your script in order to interact with Visionscape properly. The information in this chapter explains when each of those subroutines is called.

Visionscape Step Interface Subroutines

- **sub Apply** — Defines all the input datums and output datums that your script requires here. sub Apply is called when the Perl script is first selected into the custom step, and when you click Recreate Step Datums.
- **sub Run** — The bulk of your code goes here. This is the sub that is called every time your custom step runs. Here you will read the state of your input datums, perform whatever logic you require, and then set the state of your output datums. You must return a value of 0 from sub Run to indicate that the step ran successfully. You may return a negative error code to the framework to indicate that the step did not run successfully. This is different from when the step fails. Return a negative number only if your step could not run and perform its

operation as intended. Otherwise, set the Pass or Fail status datum and return 0.

- **sub PreRun** — Sets up resources needed for Run goes here. When your Job is first downloaded to the target, every step is 'PreRun' once. PreRun is not called again until the next time you download. When running your inspection in Setup Manager, however, PreRun is called every time you start a Tryout. You must return a value of 0 from sub PreRun to indicate that your step was successfully initialized. If you return an error code, Visionscape understands that your step could not properly initialize itself, and it (Visionscape) will not start your inspection.
- **sub PostRun** — This is the complement to PreRun; you clean up any memory that you may have allocated at PreRun time here. On a Device, this will be called right before your step is destroyed. In Setup Manager, when you start a Tryout, PostRun will be called first to allow you to cleanup from previous runs, then PreRun will be called, then Run.
- **sub Init** — Called once when your step is first constructed. Similar to PreRun, but where multiple calls can be made to PreRun when running in Setup Manager, Init will only be called once. On the Device, there is no difference between PreRun and Init. Init will be called before PreRun.
- **sub Cleanup** — The complement to Init, called once when your step is destroyed. Similar to PostRun, but it will only be called once in Setup Manager. On the Target, there is no difference between PostRun and Cleanup. Cleanup will be called after PostRun.
- **sub Start** — Called whenever an inspection is started by the user.
- **sub Stop** — Called whenever an inspection is stopped by the user.
- **sub Draw** — Called after sub Run has completed, but only if the user has enabled the "Show Graphics" option in Runtime Manager. If the user has chosen to disable graphics, this function will not be called. The appropriate place to put your graphics code is here. But, you can also draw graphics in sub Run, if required.
- **sub Update** — Called when your step is first loaded from disk, and whenever you click the Re-Parse Package button.

- **sub UIAction** — Called whenever the user changes one of the datum values in your step. This gives you the opportunity to change the number of datums you are displaying:
 - return 0 = Visionscape takes no further action.
 - return 1 = Visionscape calls sub Apply again.
- **Sub FilterStepType** — This sub is called when a Custom Step or Custom Vision Tool is first inserted into a Job, or when it is first created after being loaded from disk. This sub gives you the chance to specify that your script should only be listed for Custom Steps or only for Custom Vision Tools. This is based on the value that you return from the function:
 - return 0 (default) — Your script will be listed for both Step types.
 - return 1 — Your script will be listed only for Custom Vision Tools.
 - return 2 — Your script will be listed only for Custom Steps.

Essentially, use this sub to prevent the user from inserting your script into the wrong type of step.

Using Visionscape Packages in Your Script

Before we take a look at our first example script, you need to understand how to add Visionscape packages to your script, and how to make use of them. The Visionscape packages provide a Perl interface into the C++ libraries that form the Visionscape architecture. Your script can't do anything unless you make use of this "glue".

- **perlutil** — This package is the most important package to understand. Every script you create requires perlutil. It is a collection of utility functions that provides, among other things:
 - The ability to add input and output datums to your step
 - Access to those datums, so you can read and write their values.
 - Access to the input buffer and ROI (if your script is inserted into a Custom Vision Tool)
 - The ability to set the Pass/Fail status of your step.

- Using a package in your script requires the **use** keyword. To add the perlutil package, add the following line to the beginning of your script:

```
use perlutil; #loads the perlutil package
```

- Call functions in your included packages by using the package name, followed by a double colon, followed by the function name.

```
perlutil::clear_datum_lists();
```

- Most of the other packages you'll use will require you to provide an object instance. Here's an example showing how to use the PointDm package, along with perlutil, to access the X and Y data of a point passed into your step.

```
#include the line 'use PointDm;' at the beginning of your script
```

```
#use perlutil to get a ptr to an input point datum  
$ptrMyPoint = perlutil::get_datum(0,0);
```

```
#Get the X value by calling the PointDm's ::X() function.  
# You identify which point by passing in the datum pointer  
$X = PointDm::X($ptrMyPoint);  
$Y = PointDm::Y(($ptrMyPoint);
```

Adding Input and Output Datums

Virtually all of your Perl scripts will need to take in some input values, and they will typically produce some output values. How do you get input data into your script and then output data from your script? You do it just as any other step in Visionscape does, you specify a list of Input and Output datums in sub Apply. The perlutil package provides all the functions you need to add datums to your Custom Step/Custom Vision Tool.

Let's assume we wanted to add two integer input datums to our step. For the first integer datum, we want the user to be able to type in the value on the datum page. The second integer value we want to read from another step. Let's also assume that we wanted to perform some calculation using these 2 input values, and then Output the result using an integer output datum. The following code accomplishes this:

```
sub Apply  
{  
    ## always need to call this 1st to clear out old datums  
    perlutil::clear_datum_lists;
```

```

#our 2 input datums
perlutil::add_datum_int (2, 1, "FirstNumber", 1);
perlutil::add_datum_int (0, 1, "SecondNumber", 1);
#our output datum
perlutil::add_datum_int (1, 1, "OutVal", 1);

}

```

The `perlutil::add_datum_int()` adds our integer datums. It is defined as follows:

```
add_datum_int( datum_type, is_editable, strDatumName, initialValue)
```

- **datum_type** — This value specifies what type of datum you are using:
 - 0 = An input reference datum is added. This means that the user will be able to connect this datum to an integer datum from any other step in the Job.
 - 1 = An output datum. Use this to output data from your script.
 - 2 = An input resource datum is added. This means a text box will be added to the datum page where the user can type in the value to be passed into the script.
- **is_editable** — Set to 1 if you want the datum to be enabled. Set to 0 if you want to disable the datum.
- **strDatumName** — This is a string value that specifies the name of the datum. This is the string that will be displayed on the datum page.
- **initialValue** — This is the default value that will be assigned to the datum.

So, after clearing the datum lists in the example above, we added our first integer input datum with the following line of code:

```
perlutil::add_datum_int (2, 1, "FirstNumber", 1);
```

The first argument is 2, which means the user will get a text box in which to enter the value. The second arg is 1, which means the datum will be editable. The third arg "FirstNumber" is the actual text that will be displayed on the datum page, and the fourth arg is 1, which is the initial value that will be set into the datum.

We added the second integer input datum with this line:

```
perlutil::add_datum_int (0, 1, "SecondNumber", 1);
```

The first argument this time was set to 0, which means that we want the user to be able to connect this datum to the integer datum of some other step.

We added the output datum with this final line of code:

```
perlutil::add_datum_int (1, 1, "OutVal", 1);
```

The first argument was set to 1 in this case, which means we want this to be an output datum.

The perlutil package also provides `add_datum_double()`, `add_datum_string()`, `add_datum_point()`, and methods for every other data type supported by Visionscape. Although Perl does not require that you specify data types, you have to remember that you are interfacing to the C++ Visionscape libraries, and that is why you must always specify the data type of the datum you are adding. Refer to Chapter 6, “Perl Reference” for a complete list of all the perlutil functions.

The following example demonstrates how to add some other common data types:

```
sub Apply
{

    perlutil::clear_datum_lists;

    ##### add input datums #####
    #a floating point resource datum
    perlutil::add_datum_double(2, 1, "My Float Val", 60);

    # Enum datums are represented with a drop down list box
    # The text options you want displayed in the list box must be
    # entered into an array, and that array must then be passed
    # to the add_datum_enum function
    my @options = ("option 1","option 2", "option 3");
    perlutil::add_datum_enum (2,1,"ScriptOptions",\@options, 0);

    ## A status datum will be represented by a check box
    perlutil::add_datum_status(2, 1, "A Status Datum", 1);
```



```

## Add reference datums that will allow us to read in a
## Point and Line datum from other steps
perlutil::add_datum_point(0, 1, "InPoint\nInput Point");
perlutil::add_datum_line(0, 1, "InLine\nInput Line");

}

```

Getting and Setting Datum Values for Simple Types

Now that you know how to add datums to your step, you need to understand how to get the values from the inputs, and set values into your outputs. Once again we will use the perlutil package for this. Just as perlutil provides a long list of **add_datum_###()** functions for each datum type, it also provides a list of **get_datum_###()** and **set_datum_###()** functions. Typically, you will use these functions at runtime, in sub Run(). Consider our example from the previous section where we added two integer input datums, and one integer output datum. Let's say we wanted to read both of the integer values at runtime, add them together, and then output that value via our integer output datum. The code would look like this:

```

sub Run
{
    # read in the integer values
    my $FirstNumber = perlutil::get_datum_int (0, 0);
    my $SecondNumber = perlutil::get_datum_int (0, 1);

    #add them together
    my $sum = $FirstNumber + $SecondNumber;
    #set the sum into our output datum
    perlutil::set_datum_int (1, 0,$sum);

    return 0;
}

```

We read the integer values using the perlutil function **get_datum_int()**:

- **get_datum_int(in_or_out, datum_index)**
 - **in_or_out** — Specifies if datum is an input or output datum
 - 0 = Input datum (use this for both resource and reference datums)

- 1 = Output datum
- datum_index — The 0 based index of the datum. The index is specific to the datum type you specified with the “in_or_out” parameter. In other words, if you specified an output datum, and passed in a datum_index of 1, you’re saying you want the second output datum.

Specifically, we used the following 2 lines of code to read in each of the integer inputs:

```
my $FirstNumber = perlutil::get_datum_int (0, 0);  
my $SecondNumber = perlutil::get_datum_int (0, 1);
```

For the first call to **get_datum_int()**, we passed the arguments (0,0), which means we want the value of the first input datum. In our second call, we passed the arguments (0,1), which means we want the value from the second input datum.

In order to set a value into our output datum, we used the perlutil function **set_datum_int()**:

- set_datum_int(in_or_out, datum_index, value)
 - in_or_out — 0= input datum, 1 = output datum
 - datum_index — The index of the datum you wish to set
 - value — The value you want to set into the datum

Specifically, we used the following line of code to set the output datum:

```
perlutil::set_datum_int (1, 0,$sum);
```

- The first parameter (1) means we are setting an output datum.
- The second parameter (0) means we want the first output datum.
- The third parameter is the variable \$sum, which holds the value we want to set into our datum.

The method we just demonstrated works fine for all simple data types like doubles, strings, integers and StatusDms. You simply substitute the appropriate **get_datum_###()** and **set_datum_###()** functions. The

methodology will change somewhat, though, when you are dealing with more complex types like points and lines.

Getting and Setting Complex Datum Types

Many of the data types in Visionscape can not be represented by a single scalar value. For instance, a Point is represented in Visionscape as a four element array that holds X, Y, angle and scale values. The Line datum holds an A, B and C value (which correspond to the line equation $Ax + By + C = 0$). In order to get and set the data of these types, you will need to first get a pointer to the datum, and then use the package we provide for that data type to access its internal values. Let's create a simple script that reads in a Point datum from some other step, and then extracts the X and Y values and prints them out to the Visionscape debug window:

```
## at start of script, we'll include the following packages
use PointDm;
use perlutil;

sub Apply
{
    #clear all existing datums
    perlutil::clear_datum_lists;

    #add input datum for the point
    perlutil::add_datum_point(0, 1,"InPoint\nInput Point");
}

sub Run
{
    #get the datum pointer for our input point datum
    $pPt = perlutil::get_datum(0, 0);

    ##use the PointDm package to read the X and Y values from the Pt
    $InX = PointDm::X($pPt);
    $InY = PointDm::Y($pPt);

    print "Input Pt X,Y = $InX, $InY";
    return 0;
}
```

The call to `get_datum(0,0)` will return a pointer to the first input datum in our script, not the actual value of the datum. Then, we use the `PointDm` package to extract the X and Y values. When calling the various methods

of the PointDm package, the first argument will always be a pointer to the Point datum you are querying. You will see this type of syntax often in the upcoming examples, where we will get a pointer to a particular object type, and then pass that pointer into its corresponding package in order to make use of it.

Creating a Complete Script

Up until now, we've been looking at the various pieces of a Visionscape Perl script. Now, let's now take a look at a very simple example of a complete script. We'll again use the example of reading in two integer values, adding them together, and then outputting the sum.

```
use perlutil; ##provides access to the perlutil package

## Your script is also a package,

## use the package keyword to specify its name
package add_two_ints;
#
# Register this package so that perl steps know this package is
#available
#
perlutil::register("add_two_ints");

# define the input and output datums here
sub Apply
{

    #must always clear out any old datums first
    perlutil::clear_datum_lists;

    #add two integer input datums
    perlutil::add_datum_int (2, 1, "FirstNumber", 1);
    perlutil::add_datum_int (2, 1, "SecondNumber", 1);

    # add one integer output datum
    perlutil::add_datum_int (1, 1, "Result", 1);
}

sub Run
{
    my $FirstNumber = perlutil::get_datum_int (0, 0);
    my $SecondNumber = perlutil::get_datum_int (0, 1);
    perlutil::set_datum_int (1, 0, $FirstNumber + $SecondNumber);
    $sum = perlutil::get_datum_int (1, 0);
    $outstr = "Sum of numbers is " . $sum;
    return 0;
}
```

```
return 1;
```

Hopefully, at this point, the code in sub Apply and sub Run makes sense to you. The code at the top of the script is what turns this into a complete Perl package that Visionscape can actually load and run. The following information contains descriptions of the key lines that you should understand:

- `use perlutil;`

This line loads the perlutil package so that we can use it in our script. Any and all other packages that you need to add to your script should be added here, at the beginning of the file.

- `package add_two_ints;`

The **package** keyword defines the name of your package. In this case, our package is called “add_two_ints”. This is the name that will be displayed for the user in the “Avail Package Scripts” datum drop-down list when you add a Custom Step or Custom Vision Tool to your Job.

- `perlutil::register("add_two_ints");`

You must always include this line so that your script will be registered in Visionscape when it is loaded.

Note: The name that you pass in must match the name of your package exactly! In other words, because we named our package “add_two_ints”, that exact string must be passed to the **register()** function.

- `return 1;`

This must always be the last line in your script file; it essentially terminates your script. It is important to note that you can define multiple packages within your script file, but you only need to include this line once at the very end. Don't put this at the end of each package; nothing will be loaded by the Perl parser after this line is encountered.

Creating Your Own Scripts

Hopefully we've now given you all the information you need to create a basic Perl script from scratch. However, rather than always starting out with a blank page in a text editor, it is much easier to simply start out with one of the basic scripts that is installed with Visionscape and modify that file. Here is the typical set of steps to follow when you want to create your own script:

1. Go to the folder `\Vscape\Jobs\perlmod` and, using any text editor (Wordpad, Notepad, CodeWright,...), open the file “none.pm”. This file essentially represents an empty script that does nothing. But, all the boilerplate code is there to get you started. The script will look like this:

```

use perlutil;

##### DUMMY PACKAGE #####

package none;

#
# Register this package so that perl steps know this package is
#available
perlutil::register("none");

sub Init          { }
sub Cleanup { }

sub Apply
{
    perlutil::clear_datum_lists;
}

sub Update      { }
sub Start      { }
sub Stop       { }
sub PreRun     { return 0; }
sub PostRun    { }
sub Draw       { }
sub Train      { return 0; }
sub Run        { return 0; }
sub UIAction   { return 0; }
sub FilterStepType { return 0; }

return 1;

```

2. Save the file as a different file name, say "myscript.pm". Make sure you save it in the \Vscape\Jobs\perlmod folder.
3. Modify the package name and the call to perlutil::register(). Make sure the names match:

```

package myscript
perlutil::register("myscript");

```

4. Add code to sub Apply to add whatever datums you require.
5. Add code to sub Run to perform whatever runtime tasks you require.

6. Add your script file name to the “perlscr” file so it can be loaded by Visionscape. Open “perlscr” in your text editor, and add the following line:

```
require myscript;
```

7. Launch FrontRunner, and insert either a Custom Step or Custom Vision Tool into your Job. Open the “Avail Package Scripts” drop-down list, and select your script from the list. Your script will be loaded, and the datum page will update to show you the datums you added in sub Apply.
8. Any errors that may occur when your script is parsed by the Perl parser will be posted to the Visionscape Debug Window.
9. After you make changes to your script, save the file, and then click Re-Parse Package Script on the datum page of your Custom Step or Custom Vision Tool. This will reload the file, and re-parse it. Once you’ve selected your script in a Custom Step or Custom Vision Tool, the script will not be reloaded again unless you click Re-Parse Package Script. It is a common mistake for users to change their script, save the file, and then simply run the Job in FrontRunner without re-parsing.

Note: If you added the name of your script file to the perlscr file, but it does not show up in the list of scripts, that typically means one of the following issues has occurred:

You misspelled the name of the file when you entered it into “perlscr”, or you entered the case incorrectly. In other words, if the file name is myscript.pm, and you entered “require myscript;”, the file will not be loaded.

There are errors in your script. Visionscape will try to load and parse all of the scripts that are listed in “perlscr” the first time you insert a Custom Step or Custom Vision Tool. If it encounters any parser errors along the way, it will write them to the debug window, and stop loading. This means that no scripts after the script with the error will be loaded.

If FrontRunner was running when you modified the perlscr file, you may need to shut it down and restart it in order to force it to reload the

file. However, it is not necessary to restart FrontRunner if you make a change to the perl script.

Always check the debug window whenever your script doesn't show up in the list. In most cases, the error message there will make it clear as to why your script didn't load.

Examples

This chapter provides you with many practical examples. We will demonstrate, via various code segments, how to perform common tasks that are often required in Visionscape Perl scripts.

Determining If An Input Datum Is Connected

When you add a reference input to your script, you are expecting that the user will connect that input to the datum value of some other step in their Job. However, the user may forget. Thus, in most cases, you will want to determine whether or not the datum has been connected.

Include the Composite package in your script:

```
use Composite;
```

Use the following code to check if the datum is connected. If the datum is connected to itself, that means the user never connected it to anything.

```
#get pointer to your reference datum
$Datum = perlutil::get_datum(0,1);

#use the Composite package to get a pointer to the datum
# that our datum is currently connected to
$refDat = Composite::RefTo($Datum);

#When a datum is not connected it points to itself.
# So if the 2 pointers are NOT the same, then
# the user connected it to something
```

```
if($Datum != $refDat)
{
    #safe to use the datum now
}
```

Preventing Your Script from Running in the Wrong Type of Step

You may create a script that needs to run in a Custom Vision Tool, in order to make use of the connection to the buffer so that you can draw graphics, or to make use of its ROI. By default, any script that you create can be inserted into either a Custom Step OR a Custom Vision Tool. You can, however, specify that your script should only run in one of these Step types. You do this by implementing sub FilterStepType, and returning a value that specifies the type of step you wish to run in sub FilterStepType:

```
{
    return 1; #1 specifies a Custom Vision Tool
           # 2 specifies a Custom Step
           #0 specifies both
}
```

By implementing sub FilterStepType (see above), your script will show up only in the list of Available Package Scripts for Custom Vision Tools, and will NOT show up in the list for Custom Steps. Return a 2 if you want your script to show up only in the list for Custom Steps.

If you choose not to implement sub FilterStepType, then you will need to defend against your script being inserted into the wrong step type. In that case, you will want to determine whether or not the user inserted your script into a Custom Vision Tool, and not into a Custom Step. The key difference between these two steps is the fact that one has a connection to the input buffer and one does not, so we will use this difference in order to check the step type:

```
#get pointer to the input buffer datum from the perlutil package
$inbufdm = perlutil::get_input_bufferdm();
```

```
# if this pointer = 0, then we have no input buffer,
# which means this is a Custom Step
# if pointer is not 0, then this must be a Custom Vision Tool
if($inbufdm)
{
```

```

    #safe to access buffer
}

```

You would use the presence or absence of the input buffer to determine whether or not to run the body of your code.

Hiding the ROI of a Custom Vision Tool

It is quite common to want to draw graphics into the image buffer from your script. To do this, you will need to run your script in a Custom Vision Tool so that you can access the buffer. The Custom Vision Tool will also give you an ROI though, and you may not need or want this. You can hide the ROI by simply adding this block of code to sub PreRun:

```

sub PreRun
{
    $inbufdm = perlutil::get_input_bufferdm();
    #if this is a Custom Vision tool, hide the ROI
    if($inbufdm)
    {
        #hide the ROI
        perlutil::setInputSearchAreaVisibility(0);
    }
    return 0; #always return 0 from PreRun to indicate success
}

```

Setting The Pass/Fail Status Of The Step

Your Custom Step or Custom Vision Tool will always pass by default. You may want to cause your step to fail based on some condition. In that case, use the following perlutil statement to set your Step's pass/fail status:

```

perlutil::SetPassed(0); #pass a 0 to cause a failure
perlutil::SetPassed(1); #pass a 1 to cause a pass

```

Localizing Your Variables When Using Multiple Instances Of Your Script

It is important to understand that, if you use your script multiple times within the same Job, your package will only be loaded once by Perl. This means that you only get one copy of all of the global variables that you use within your script. For example, say you have a global variable in your

script named `$counter` that you increment every time your script runs. Now, assume you've inserted two Custom Steps into the same Inspection, and you've selected your script in both of them. When your Inspection runs for the first time, you might assume that the `$counter` variable would equal 1 in the first step, and also equal 1 in the second step. But, in fact, both steps are running the same package and, therefore, they are incrementing the same variable, so `$counter` would equal 2 after running the inspection once, 4 after running the inspection twice, etc. In order to allow you to have separate instances of a given variable every time you insert your script, we have included the `localize` package. Use the `makeLocal` function of this package on every variable that you want to localize. Using the example we just described, here's what your script should look like in order to make your `$counter` variable increment separately for each step (please note that this issue does not apply to datums, you will have separate copies of each datum for every step):

```
use perlutil;
use localize; #include the localize package

package MySample;
#
# Register this package
perlutil::register("MySample");

#####
## pass a reference to the variable you want to localize to the
## makelocal function
## notice this call is made outside of any function, it will be called
## the first time your script is loaded
localize::makelocal(\$counter);

sub Apply
{
    perlutil::clear_datum_lists;
    #add an output datum
    perlutil::add_datum_int (1, 1, "Result", 0);
}

sub Run
{
    ##increment our counter every time script runs
    ++$counter;

    #output the result
    perlutil::set_datum_int(1,0, $counter);
}
```

```

        return 0;
    }
    return 1;

```

Drawing In The Buffer

In order to draw in the buffer, you must use the BufferDm package. You must also make sure that your script is inserted into a Custom Vision Tool, since that step provides access to the input buffer, and the Custom Step does not. The BufferDm package provides a standard set of graphics calls that allows you to draw lines, rectangles, and ellipses and also allows you to set the pen and brush colors that draw the shapes. Refer to Chapter 6, “Perl Reference” for a complete list of available functions within BufferDm. In the following example, we will take in a line datum from another step, and then draw that line such that it goes from one edge of the buffer to the other.

```

use perlutil;
use LineDm;
use Line;
use BufferDm;

##### A Script to Draw a Line passed in from another step
package DrawLine;

#
# Register this package
perlutil::register("DrawLine");

sub Apply
{
    perlutil::clear_datum_lists;

    #input datums
    perlutil::add_datum_line(0,1,"InLine\nInput Line");
    perlutil::add_datum_status(2,1, "Draw Line?", 1);
}

sub PreRun
{
    $inbufdm = perlutil::get_input_bufferdm();
    #if this is a Custom Vision tool, hide the ROI
    if($inbufdm)
    {
        perlutil::setInputSearchAreaVisibility(0); #hide the ROI
    }
}

```

```
sub Run
{

sub Draw
{
    #Get Pointer To The Input Line
    $InLine = perlutil::get_datum_point(0,0);
    $DrawIt = perlutil::get_datum_status(0,1);

    $refto = Composite::RefTo($InLine);
    #Get a Pointer to Our Input Buffer Datum
    $MyBuf = perlutil::get_input_bufferdm;

    #make sure our input line is connected, and we have an input buf

    if(($refto != $InLine) && $MyBuf != 0)
    {
        #Extract the A, B and C Values From the Line
        $ptrLine = LineDm::GetLinePtr($InLine);
        my $A = Line::GetLineA($ptrLine);
        my $B = Line::GetLineB($ptrLine);
        my $C = Line::GetLineC($ptrLine);

        #Get Buffer Dimensions
        $Width = BufferDm::Width($MyBuf);
        $Height = BufferDm::Height($MyBuf);

        #calculate start and end points for our line
        $LeftX = 0;
        $RightX = $Width;
        $LeftY = -($A/$B)*$LeftX - $C/$B;
        $RightY = -($A/$B)*$RightX - $C/$B;
        #Set the Color the Line Will be Drawn in to Purple

        #ARGS = R,G and B Values Between 0 And 255
        BufferDm::SetPenColor($MyBuf, 255,0,255);
        # Draw the Line
        BufferDm::MoveTo($MyBuf, $LeftX, $LeftY);
        BufferDm::LineTo($MyBuf, $RightX, $RightY);
    }

}
return 1;
```


Changing Another Step's Datum Value

Imagine you have an application where you have a Blob tool or a Flaw tool, and you need to calculate a different threshold for these tools each time they run. You can do this by creating a very simple Perl script that can take in a value from another step (say the VarAssign step, in which you could place the logic to calculate the threshold), and then poke that value into the datum of some other step (like the Low Threshold of a Blob step). The script would look like this:

```
use perlutil;
use Composite;
use IntDm;
use Datum;

##### Package to Allow an Integer Resource Datum to be Modified
##### at Runtime, Just Like a Reference Datum
package DatumChange;

# Register this package
perlutil::register("DatumChange");

sub Apply
{
    perlutil::clear_datum_lists;

    #add a reference datum of type double
    # this will connect to the out val of the VarAssign step
    perlutil::add_datum_double(0,1,"SetVal\nValue to Set into Int
Datums", 80);

    #add a reference datum of type integer
    # this will connect to the threshold datum of a blob or flaw tool
    perlutil::add_datum_int(0,1,"ModDatum1\nInteger Datum 1", 0);
}

sub PreRun
{#Turn Off the Precheck Test for Our Integer Datum.
 # By Default, a Step in Visionscape Will Not Run if any of the Steps
 # It Is Connected to have failed. Since the Blob or Flaw We Will

 # Connect to comes after our step, it will always be in a failed
 # state when we run
 $dat = perlutil::get_datum(0,1);
 Datum::PrecheckOff($dat);
 my $flags = Composite::StatusFlags($dat);
 $flags |= 0x00010000; # Set the 'SKIP REGEN' bit
 Composite::SetStatusFlags($dat, $flags);
}

sub Run
```

```

{
    #Get the value that was passed into our script
    $Val = perlutil::get_datum_double(0,0);

    #Get datum handle of our integer datum
    $Datum = perlutil::get_datum(0,1);
    $refDat = Composite::RefTo($Datum);
    # Make sure we're connected to something
    if($Datum != $refDat)
    { #Set the passed in value into our integer datum
        IntDm::SetValue($Datum, $Val);
    }
}

return 1;

```

Working With Point Lists

Several steps within Visionscape will generate Point List datums, which are nothing more than lists of points. The Edge and Fast Edge tools will generate a list of edge points located by the tool. You may want to pull this list of points into your script and perform some operation on them. You may also create a script in which you want to output your own list of points. In the following example, we will take in a PointListDm, iterate through its points, add 20 to each of the X values, and then output our own PointListDm. We will use the PointListDm package.

```

use perlutil;
use Composite;
use PtListDm;

###A Package that demonstrates how to enumerate pts in a point list
package PtList;

# Register this package so custom steps know it is available
perlutil::register("PtList");

sub Apply
{
    # clear all existing datums
    perlutil::clear_datum_lists;

    #####Input point list datum
    perlutil::add_datum_ptlistdm(0, 1, "Input Point List");

    #####Output Point List Datums:#####
    perlutil::add_datum_ptlistdm(1, 1, "Output Point List");
}

```

```

sub Run
{
    # Get handle to the input Pt List
    $InPtList = perlutil::get_datum(0, 0);
    $refIn = Composite::RefTo($InPtList);
    #Get handle to the output Pt List
    $OutPtList = perlutil::get_datum(1, 0);

    # Make sure input reference is assigned
    if($refIn != $InPtList)
    {
        #Call GetPointListByArray() to allow us to access
        # individual pts quickly
        $PtListArr = PtListDm::GetPointListByArray($InPtList);
        # get the number of points in the list
        $PtListSize = PtListDm::GetSize($InPtList);

        #Empty the Output Pt List First
        PtListDm::Empty($OutPtList);
        #Loop through the point list
        for($i =0; $i < $PtListSize; ++$i)
        {
            # Get the X and Y Values From the Array
            $X = PtListDm::GetPointListX($PtListArr, $i);
            $Y = PtListDm::GetPointListY($PtListArr, $i);

            #Add them to our Output Pt List
            PtListDm::AddPoint($OutPtList , $X+10, $Y+10);
        }
        #Be sure to free the array when you are done!!!!
        # Your scripts will leak memory if your omit this call
        PtListDm::FreePointListByArray($PtListArr);
    }
    else
    {
        printf "Error in PtList script: input reference was not
assigned!!\n";
    }

    return 0;
}

return 1;

```

IO Access

You can get and set the state of any IO point from your Perl scripts. The easiest way to do this is by adding an `IOList` datum to your step. This datum will allow the user to select any type of IO (Physical, Virtual, Strobe, etc.) as well as the IO index. The `IOListDm` is the same datum that all steps in Visionscape use when they need to allow the user to select an IO point. The following example demonstrates how to add `IOListDms`, and then how to read and write to them. We'll add one `IOListDm` that allows the user to select an Input IO point, and a second that allows them to select an Output IO point. At run time, we'll read the state of the input: if it's ON, we'll turn the output OFF, and if it's OFF, we'll turn the output ON.

```
use perlutil;
use IOListDm;

###      A Package that demonstrates IO usage
package IOSample;

# Register this package
perlutil::register("IOSample");

sub Apply
{
    # clear all existing datums
    perlutil::clear_datum_lists;
    # Add an IOListDm datum to allow user to select
    # an input IO point
    my $InputType = 1; #1 = Input, 2 = output
    perlutil::add_datum_iolistdm(2, 1, "Input IO Point",
    $InputType);

    # Add an IOListDm datum to allow user to select
    # an output IO point
    my $OutputType = 2;
    perlutil::add_datum_iolistdm(2, 1,"Output IO
Point", $OutputType);
}

sub PreRun
{
    # Prepare the IO objects
    $pIOListDmIn = perlutil::get_datum_iolistdm(0,0);
    if ($pIOListDmIn != 0)
    {
        IOListDm::PrepareIO($pIOListDmIn);
    }
    $pIOListDmOut = perlutil::get_datum_iolistdm(0,1);
    if ($pIOListDmOut != 0)
    {
```

```

        IOListDm::PrepareIO($pIOListDmOut);
    }
    return 0;
}

sub Run
{
    $pIOListDmIn = perlutil::get_datum_iolistdm(0,0);
    $pIOListDmOut = perlutil::get_datum_iolistdm(0,1);
    if ($pIOListDmIn != 0 && $pIOListDmOut != 0)
    {
        # Read state of IO point selected by user in the IOLIST
        datum.
        $InputState = IOListDm::ReadMyIOPoint($pIOListDmIn);

        #Set out output to the opposite state
        if($InputState == 1) ##If input is on
        {
            IOListDm::WriteMyIOPoint($pIOListDmOut, 0);
        }
        else #Must be off
        {
            IOListDm::WriteMyIOPoint($pIOListDmOut, 1);
        }
    }
    return 0;
}
return 1;

```

Accessing A Blob Tree

The Blob tool in Visionscape can find many blobs within its ROI. Using the standard set of steps provided by Visionscape, we only allow you to access one of those blobs, based on some filtering logic entered into the Blob Filter step (typically returns the largest blob). However, you may want to access the data from all of the blobs in order to perform some custom measurement, or to make a pass/fail determination. You can do this in a Perl script by using the **BlobTreeDm** package to access the BlobTree output from a Blob Tool, then the **BlobResult** package, which allows us to access the overall blob data, and then the **Blob** package, which allows us to extract the data for a specific blob (center X, Y, area, etc.). In the following example, we will add a BlobTreeDm to our step. This will allow us to connect the Blob Tree output data from any Blob tool to our step. In sub Run, we will then walk through all of the blobs in the tree, and draw a box around each one:

```

use perlutil;
use Composite;

```

```

use BlobTreeDm;
use BlobResult;
use Blob;
use BufferDm;

#####
# extracts the blob data from input blob tree and
# draws a bounding box around each
#####
package BlobBoxes;

perlutil::register("BlobBoxes");

sub Apply
{
    perlutil::clear_datum_lists;

    #Add a Blob Tree input datum to our step
    perlutil::add_datum_blobtree (0, 1, "BlobTree\nBlob Tree", 0.0);
}

sub PreRun
{
    #Hide the ROI
    if (perlutil::get_input_bufferdm) {
        perlutil::setInputSearchAreaVisibility(0);
    }
    return 0;
}

sub Run
{
    #Get a pointer tot he Blob Tree datum
    $blobtreedm = perlutil::get_datum (0, 0);
    $ref = Composite::RefTo($blobtreedm);

    #Make sure user connected the datum
    if($blobtreedm != $ref)
    {
        #Get a reference to the actual blob tree object
        $blobtree = BlobTreeDm::GetBlobTree($blobtreedm);
        #Use BlobResult package to find out how many
        # blobs are in the tree
        $numblobs = BlobResult::NBlobs($blobtree);

        #Loop through all the blobs
        for($i=0; $i<$numblobs; ++$i)
        { #Get a pointer to the blob at this index
            $blobhan = BlobResult::GetBlob($blobtree, $i);

            #Now you can use Blob package to extract data
            $blobX = Blob::CentroidX($blobhan);
            $blobY = Blob::CentroidY($blobhan);
            print "Blob $i has a center point of $blobX, $blobY";
        }
    }
}

```

```

        ##Get the left, top, right, & bottom most points of the
blob
        $left = Blob::LeftPointX($blobhan);
        $right = Blob::RightPointX($blobhan);
        $stop = Blob::TopPointY($blobhan);
        $bottom = Blob::BottomPointY($blobhan);

        ## Draw a purple rectangle around the blob
        $inbuf = perlutil::get_input_bufferdm;
        if($inbuf)
        { BufferDm::SetPenColor($inbuf, 255,0,255);
          BufferDm::Rectangle($inbuf, $left, $stop, $right,

$bottom);
        }
    }
    return 0;
}
return 1;

```


Measurements and Coordinate Transforms

Performing a measurement will generally involve taking in a point, line, angle or area value from a vision tool, and then calculating and outputting distances, areas, positions, etc., between those inputs. If all points and lines were guaranteed to come in to your Perl script in the same coordinate system, performing your measurement would be fairly straightforward. However, Visionscape makes ample use of preprocessing steps that produce their own output buffers (like the Rect Warp, Sobel and GrayMorph steps). Points and lines that are produced by vision steps running inside these output buffers will be in the coordinate system of that buffer, not the coordinate system of the Snapshot's buffer.

FIGURE 5–1.



For example, the step tree in Figure 5–1 shows two Fast Edge Tools, one directly beneath the Snapshot step, and one inside of a Rect Warp step. The first Fast Edge runs directly inside of the Snapshot step and will produce a line and a point in the coordinate system of the Snapshot

(where location 0,0 equals the top left corner of the image). The second Fast Edge Tool runs inside of the buffer produced by a Rect Warp tool, and will therefore produce a point and a line in the coordinate system of that buffer (where 0,0 equals the top-left corner of the Rect Warp's ROI). Therefore, if you wanted to calculate the distance from the first Fast Edge Tool's point to the second Fast Edge Tool's line, it is up to you to transform the point and line into the same coordinate system first, and then perform your distance calculation.

Transforming Points and Lines

The Part Object

You transform points and lines by using special Visionscape objects known as Parts. A Part object stores the transform information that allows coordinates to be transformed from Pixel units to World units, and from World units back to Pixel units. Point datums and Line datums contain pointers to Parts that contain the information required to transform that point/line up to its parent buffer. For example, consider the sample step tree shown in Figure 5-1. The second Fast Edge Tool that runs inside of the Rect Warp will produce an output Line datum and an output Point datum. Each of these datums has a part associated with it, and that part holds the transform information required to transform the point and line up to the Snapshot buffer's coordinate system, because that is its parent buffer. The line and point datums from the first Fast Edge Tool, also have parts associated with them. But these parts hold information to transform the point and line into the World coordinate system (where calibrated units are used, inches, millimeters, etc.), because that is the parent coordinate system of the Snapshot step. Parts are generally associated with datums, but steps with output buffers (Snapshot, Sobel, Graymorph, etc.) will also have an associated part.

Performing a Transformation

There are various ways to transform your point and line inputs, and the method that you choose depends on the results you want to achieve. In the following example, we will demonstrate three different approaches to transforming a point and a line (it is assumed that you understand the basics of putting together a Perl script at this point; therefore, all code is not shown).

First, we must be sure to include the necessary packages at the top of the script:

```
use perlutil;    # required for all Visionscape perl scripts
use LineDm;     # allows us to work with line datums
use Line;       # allows us to work with raw line data
use PointDm;    # allows access to point datums
use Point;      # allows access to raw points
use Part2DStep; # This object works with Part objects, and allows us
                # to perform transforms on points and lines
```

Set up the inputs to your script to accept a line and point datum. Also, add output datums to hold the transformed line and point:

```
sub Apply
{
    perlutil::clear_datum_lists;
    #input datums
    perlutil::add_datum_point(0, 1, "InPoint\nInput Point");
    perlutil::add_datum_line(0, 1, "InLine\nInput Line");

    #output datums
    perlutil::add_datum_point(1, 1, "TransPt\nTransformed Point");
    perlutil::add_datum_line (1, 1, "TransLine\nTransformed Line");
}
```

At run time, get the line and point datums, and the parts associated with them:

```
sub Run
{
    # Get pointers to the input point and line datums
    $InPointDm = perlutil::get_datum(0, 0);
    $InLineDm = perlutil::get_datum(0, 1);

    # Get pointers to the output datums
    $OutPtDm = perlutil::get_datum(1,0);
    $OutLineDm = perlutil::get_datum(1,1);
```

Using the PointDm and LineDm packages, call the **OwnerPart()** function to get a pointer to the Part object associated with our input point, and the Part associated with our input line:

```
#PointDm::OwnerPart takes a pointer to a Point datum as its only argument
$PtPart = PointDm::OwnerPart($InPointDm);
#LineDm::OwnerPart takes a pointer to a Line datum as its only argument
$LinePart = LineDm::CoordPart($InLineDm);
```

In order to transform the point and line, we will use the **Part2DStep** package. Part2DStep has **TransformPoint()** and **TransformLine()**

functions. Each of these functions requires that you specify the Part for the point/line you want to transform, as well as the Part from the coordinate system you want to transform to. Assume the purpose of this script was to calculate a point to line distance. In that case, we just need to get the point and line into a common coordinate system, and we do that by using the **FindCommonAncestor()** function in **Part2DStep** like this:

```
$CommonPart = Part2DStep::FindCommonAncestor($LinePart, $PtPart);
```

The value returned is a pointer to a part object for the closest parent coordinate system shared by the two input parts. We can now transform the line and the point with the following code:

```
##### Transform the input point #####

#Construct a point object by extracting the
# X and Y values from our input point datum

$InX = PointDm::X($InPointDm);
$InY = PointDm::Y($InPointDm);

#point constructor takes an x an y value
$TransPt = Point::Point($InX, $InY);

#Call Part2DStep::TransformPoint to perform the transform.
# Pass the Part from the Point's coordinate system ($PtPart)
# the Point to be transformed($TransPt),
# and the Part from the coordinate system we're
# transforming to ($CommonPart)

##### The results will be put back into $TransPt.
##### Note that TransformPoint() works only
##### on Point objects, and NOT on PointDm objects.

Part2DStep::TransformPoint($PtPart, $TransPt, $CommonPart);

##### Transform the input line #####

#Call Part2DStep::TransformLine, Pass the
# Part from the input line datum's coordinate system ($Linepart)
```

```

# The Line datum we want to transform ($InLineDm)

# Another Line datum to hold the transform results ($OutLineDm)
#and part from the coord sys we're transforming to ($CommonPart)

# Unlike TransformPoint,
# TransformLine can work directly with a Line datum

Part2DStep::TransformLine($LinePart, $InLineDm, $OutLineDm,

$CommonPart);

Now output the transformed line and point using our output
datums:

#Using the pointer to our output point datum,
# set our transformed point into it.
# Other steps can now access the transformed point results
PointDm::SetPoint($OutPtDm, $TransPt);

# Using the pointer to our output line datum,
# set the transformed line into it.
# Other steps can now access the transformed line results
$OutLine = LineDm::GetLinePtr($OutLineDm);
LineDm::SetLine($OutLineDm, $OutLine);

#end of sub Run
}

```

The preceding example works well for transforming your points and lines as long as getting them into a common coordinate system is all that you need. But, what if your script takes in only one point, and you don't have a second datum with which to find a common ancestor; or, what if it is imperative that your transformed results be in the Snapshot's coordinate system? In these cases, you will need to get the Part attached to the Snapshot step, and use that when you make your calls to TransformPoint and TransformLine. To do this, you will need to include the **Composite**

package, so add “**use Composite;**” to the top of your script. The Composite package works with Step pointers and Datum pointers (both objects are derived from the Composite object). Using Composite, we will be able to find our parent Snapshot, and then get its associated part:

```
##get the step pointer to this Custom Step tool
$pMe = perlutil::get_myStepPointer();

## Call the Composite packages FindParentOfType() function
## to find our parent snapshot step,
## The first argument is the step pointer of our custom step ($pMe),
## the second is the type of
## step (0=Target step, 1=Inspection step, 2=Snapshot step),
## the third should always be 0.
$SnapshotType = 2;
$pSnap = Composite::FindParentOfType($pMe, $SnapshotType, 0);

## Get the Part attached to the Snapshot step
$pSnapshotPart = Composite::OwnerPart($pSnap);
```

Now, you would transform your point or line in the same way as in our first example, only you would substitute \$pSnapshotPart for \$CommonPart in the calls to TransformPoint and TransformLine.

World Space

It's important to point out that all of the transformations described above will keep your data in pixels, regardless of whether your system is calibrated or not. All steps in Visionscape work this way; results are always kept in pixels, and they are only transformed to world units (inches, millimeters, etc.) by the Tolerance step and during results upload (if the user so chooses). If you need to get your point or line data into world coordinates before working with it in your script, then the key is in specifying the proper part when you call the TransformLine and TransformPoint functions. As described previously, the second part you pass to these functions defines the coordinate system you are transforming to. So, you need to get a pointer to the World part; simply construct a new Part2DStep object, and it will default to the World part.

```
#Construct a Part2DStep object
$WorldPart = Part2DStep::Part2DStep();

#perform your transforms.....
```

```
#Delete the allocated object  
Part2DStep::Part2DStep_Delete();
```

You would transform your point or line in the same way as in our first example, only you would substitute \$WorldPart for \$CommonPart in the calls to TransformPoint and TransformLine.

Note: If your system is not calibrated, this method will take you into Snapshot space, since Snapshot and world space are the same in an uncalibrated system.

When you are outputting datums from your script, you need to consider whether you want calibration to be applied to the results or not, because calibration is not applied to all datum types. The following datum types will always be unaffected by calibration:

- Integer
- Double
- Status
- String

The following types will always apply the calibration:

- Distance
- Area
- Angle
- Point
- Line

A common mistake is to calculate a distance in your script, and then output it using a double datum rather than a distance datum.

Performing Measurement Calculations

In the transformation examples, we simply transformed the point and line into a common coordinate system, and then outputted the results in point

and line datums. Most users will want to perform calculations with the transformed data. There are various options for doing this.

Access the Raw Line and Point Data

Get at the raw point data in the transform example by doing the following:

```
PointDm::X($OutPtDm); #extract the X value from the point
PointDm::Y($OutPtDm); #extract the Y value from the point
```

Line data is stored in the form of **AX + BY + C = 0**. Access the raw line data like this:

```
Line::GetLineA($OutLine);
Line::GetLineB($OutLine);
Line::GetLineC($OutLine);
```

Perl has a full compliment of trigonometric math functions, so with the raw point and line data, you can perform whatever calculations you need.

Use Built In Measurement Functions

The Line package has many useful functions built into it for calculating point to line distances, line to line intersection points, angles between lines, etc. Refer to Chapter 6, “Perl Reference” for a complete list of the available functions in the Line package. The most commonly used functions are:

```
#Line to Point Distance: LinePtDist()
# Compute the distance from the input point to the input line.
# This distance is saved in the distance datum pDistDm.
#$pTransLine = pointer to the line object (not a LineDm)
#$pPoint = pointer to a Point object (not a PointDm)
#$pDistDm = pointer to the DistanceDm object that will hold the result
    Line::LinePtDist($pTransLine, $pPoint, $pDistDm);

#Normal point: NormalPt
# Computes the point at which a line that is perpendicular to
# the specified input line, would intersect that line if it ran
# through the specified input point.
$pPoint2 = Point::Point(0.0,0.0); #Construct an empty point
# $pTransLine = pointer to the input Line object
# $pPoint = pointer to the input point
# $pPoint2 = pointer to the point that will hold the results
Line::NormalPt($pTransLine, $pPoint, $pPoint2);
```



```
#Point to Point Line: TwoPtLine
# Computes the line that would run through the two input
$ppLine = Line::Line(); #Construct an empty line object
# $ppLine = The Line object. This will hold the result
# $pPoint = pointer to first input point
# $pPoint2 = pointer to second input point
Line::TwoPtLine($ppLine, $pPoint, $pPoint2);
```


Perl Reference

Introduction

The variable names listed help indicate the data type of the variable. For example:

- `$Composite_` A pointer to a composite object
- `$BinMorphAgent_` A pointer to a binary morphology agent
- `$int_ret` A return value from a function that is an integer
- `$double_ret` A return value from a function that is a double
- `$float_ret` A return value from a function that is a float
- `bool_ret` A return value from a function that is a boolean

Factory calls, which create vision agents, deviate from this scheme by returning `$int_ret`, indicating a pointer to the created vision agent.

Note: The package name and function calls for all packages are case sensitive.

The following available packages are listed alphabetically and in the correct case.

AngleDm	AreaDm	ArithAgent
ArithFact	ArithResults	AutoFocusAgent

```

AutoFocusFact BinMorphAgent BinMorphFact
BinMorphOperatorBlob BlobAgent
BlobDm BlobFact BlobResult
BlobTreeDm Buffer BufferDm
Composite ConvAgent ConvFact
CorrMatchAgentCorrMatchFact CorrPoint
CorrResult CorrSearchAgentCorrSearchTempl
CorrSearchTemplateCorrSrchAgentCorrTempl
CorrTemplate Datum DecimateAgent
DecimateFact DistanceDm DMRAgent
DMRFact DMRResults DMRResultsDM
DoubleDm EdgeFast EdgePrim
EnumDm Fpoint Frect
GradScanAgentGradScanFact GrayMorphAgent
GrayMorphElemGrayMorphFact HistogramAgent
HistogramFact HoughAgent HoughFact
InspectionResultsDmIntDm Kernal
Line LineDm Localize
MaskAgent MaskDm MaskFact
Matrix3x3 Monster OcvAgent
ocvresult Part2DStep perlutil
Point PointDm ProjectAgent
ProjectFact PtListDm Quad
Rect RectDm ROI
SceneAngleAgentSerial ShapeDm
SobelAgent SobelFact StatusDm
Step StringDm TemplateDm
Transform TransposeAgentTransposeFact
VectorFast VectorFastFact VectorFamily
VectorFamilyResultsvideo WarpAgent
WarpFact

```

The remaining sections of this chapter discuss the following:

- “Composite-Datum-Step Packages (Common Functions)” on page 6-18
- “Datums” on page 6-23
- “Vision Library” on page 6-41
- “Morphology” on page 6-53

- “Utility Packages” on page 6-100
- “Line-by-line Analysis of a Script” on page 6-106
- “Usage Examples” on page 6-112
- “Read Digital I/O” on page 6-140
- “Write Digital I/O” on page 6-141
- “Helpful Programming Information” on page 6-143

Perlutil — Perl Utility Package

The perlutil package is the most important package and is discussed first. Perlutil contains hooks to several functions used to configure the Custom Step / Custom Vision Tool. This package will be used by virtually all perl scripts.

Datum Access

A Custom Step / Custom Vision Tool may process data from input datums, create output datums to hold results, or both. Each Custom Step / Custom Vision Tool contains a list of input datums and a list of output datums. The number and type of datums are determined by the Perl script used by the Custom Step / Custom Vision Tool. Access to datum functionality is provided in the perlutil package through the following functions:

- `clear_datum_lists ()` — This function (typically called in the Apply subroutine of a Perl script) clears the input and output datum lists in the step. This needs to be done before recreating the lists through `add_datum_xxxx` calls. All connections to input and output datums are lost when this function is called.
- `add_datum_xxxx` — These functions (typically called in the Apply subroutine of a Perl script) add datums of various types to the perl step. Each function takes 3 or 4 parameters:
 - `$int_inout` — Indicates whether this datum is an input datum (0), an output datum (1), or a resource datum (2).
 - `$int_canedit` — Indicates whether you can edit this datum.

- “str”— This literal is the name of the datum. The format of the name is: “symbolicname \n username”. Datums are accessed from VB through the symbolicname.
- value — This parameter is an initial value for the datum. Some add_datum_xxxx calls do not have this parameter.

An index is returned indicating the position of this datum in the input / output datum list of this step. Input and resource datums return a zero based index. For example, 0x00000004 would indicate the 5th datum in the input datum list. Output datums return a zero based index with the most significant bit set. For example, 80000002 would indicate the 3rd datum in the output datum list. Minus one (-1) is returned if an error occurs. Table 6–1 shows the available calls for adding datums in a Perl script.

TABLE 6–1. Available Calls For Adding Datums

Function	Parameter List
add_datum_angle	(\$int_inout, \$int_canedit, "str", \$double_val)
add_datum_any	(\$int_inout, \$int_canedit, "str")
add_datum_area	(\$int_inout, \$int_canedit, "str", \$double_val)
add_datum_blob	(\$int_inout, \$int_canedit, "str", \$double_val)
add_datum_blobtree	(\$int_inout, \$int_canedit, "str", \$double_val)
add_datum_buffer	(\$int_inout, \$int_canedit, "str")
add_datum_distance	(\$int_inout, \$int_canedit, "str", \$double_val)
add_datum_double	(\$int_inout, \$int_canedit, "str", \$double_val)
add_datum_enum	(\$int_inout, \$int_canedit, "str", char** optionlist, \$int_val) ¹
add_datum_expr	(\$int_inout, \$int_canedit, "str", \$double_val)
add_datum_int	(\$int_inout, \$int_canedit, "str", \$int_val)
add_datum_iolistdm	(\$int_inout, \$int_canedit, "str")
add_datum_line	(\$int_inout, \$int_canedit, "str")
add_datum_mask	(\$int_inout, \$int_canedit, "str", \$double_val)
add_datum_matrix	(\$int_inout, \$int_canedit, "str", \$double_val)
add_datum_point	(\$int_inout, \$int_canedit, "str")
add_datum_ptlistdm	(\$int_inout, \$int_canedit, "str")
add_datum_rect	(\$int_inout, \$int_canedit, "str", \$double_val)

TABLE 6–1. Available Calls For Adding Datums (continued)

Function	Parameter List
add_datum_rectshape	(\$int_inout, \$int_canedit, "str") ²
add_datum_status	(\$int_inout, \$int_canedit, "str", \$double_val)
add_datum_string	(\$int_inout, \$int_canedit, "str", \$double_val)
add_datum_template	(\$int_inout, \$int_canedit, "str", \$double_val)

¹The optionlist is a reference to an array of strings that contains the options for the EnumDm. See “Usage Examples” on page 6-112 for an example showing how to setup an EnumDm.

²If add_datum_rectshape(...) is called in a script, then the rectshape needs to be removed by the following call when the datum lists are cleared:

```
perlutil::remove_datum_rectshape(int inout, int whichone)
```

- **get_datum_xxxx** — These functions return either (1) a pointer to the specified datum, or (2) the value of the specified datum. When a datum pointer is returned, it will either be in the form of a general datum pointer (Datum *) or a pointer to a specific type of datum, like BufferDm*. Specifically:

get_datum returns a pointer to the datum (Datum*)

get_datum_any returns the value of the datum as a string (char*)

Each function takes two parameters:

- **\$int_inout** — Indicates whether this datum is in the input datum list (i.e., was added as either an input datum or a resource datum), or in the output datum list (i.e., was added as an output datum). 0=input datum list, 1=output datum list.
- **\$int_whichone** — The index of the datum. The first datum added to a list has index 0.

Calls for getting datum pointers or values from a Perl script are shown in Table 6–2. The get_datum and get_datum_any functions

can be used on any datum type; even datum types not listed in Table 6–2. For instance, area datum.

TABLE 6–2. Datum Pointers or Values From Perl Script

Return Value	Function	Parameter List
Datum *	get_datum	(\$int_inout, \$int_whichone)
char *	get_datum_any	(\$int_inout, \$int_whichone)
double	get_datum_area	(\$int_inout, \$int_whichone)
BufferDm *	get_datum_buffer	(\$int_inout, \$int_whichone)
double	get_datum_distance	(\$int_inout, \$int_whichone)
double	get_datum_double	(\$int_inout, \$int_whichone)
int	get_datum_enum	(\$int_inout, \$int_whichone)
double	get_datum_expr	(\$int_inout, \$int_whichone)
int	get_datum_int	(\$int_inout, \$int_whichone)
IOListDm*	get_datum_iolistdm	(\$int_inout, \$int_whichone)
LineDm *	get_datum_line	(\$int_inout, \$int_whichone)
PointDm *	get_datum_point	(\$int_inout, \$int_whichone)
double	get_datum_pointx	(\$int_inout, \$int_whichone)
double	get_datum_pointy	(\$int_inout, \$int_whichone)
PtListDm *	get_datum_ptlistdm	(\$int_inout, \$int_whichone)
RectShape*	Get_datum_rectshape	(\$int_inout, \$int_whichone)
int	get_datum_status	(\$int_inout, \$int_whichone)
char *	get_datum_string	(\$int_inout, \$int_whichone)

- **set_datum_xxxx** — These functions (typically called at runtime) set the value of a datum that has been added to the input or output datum list in the perl step. Each function takes three parameters:
 - **\$int_inout** — Indicates whether this datum is in the input datum list (i.e., was added as either an input datum or a resource datum) or in the output datum list (i.e., was added as an output datum). 0=input datum list, 1=output datum list.
 - **\$int_whichone** — The index of the datum. The first datum added to a list has index 0.

- value — Set the datum to this value. (A value in quotes " " indicates a string.)

TABLE 6-3. Set_Datum Functions

Function	Parameter List
set_datum_any	(\$int_inout, \$int_whichone, "val")
set_datum_area	(\$int_inout, \$int_whichone, \$double_val)
set_datum_distance	(\$int_inout, \$int_whichone, \$double_val)
set_datum_double	(\$int_inout, \$int_whichone, \$double_val)
set_datum_enum	(\$int_inout, \$int_whichone, \$int_val)
set_datum_expr	(\$int_inout, \$int_whichone, "val")
set_datum_int	(\$int_inout, \$int_whichone, \$int_val)
set_datum_status	(\$int_inout, \$int_whichone, \$int_val)
set_datum_string	(\$int_inout, \$int_whichone, "val")

Part Coordinate System Support

See also “Part2DStep” on page 6-33.

The perlutil package contains methods that support the creation, processing and deletion of part coordinate systems.

- \$partPointer = GetWorldPart() — Returns a pointer to the world PART (coordinate system).
- \$partPointer = CreateAddPart(\$part) — Creates a part (Part2Dstep object) and adds it to the framework. The pointer to the created part is returned. If the part already exists (\$part is nonzero), then no action is taken. The part is deleted by calling Part2DStep::Delete(\$partPointer).
- ConnectBufferPart(\$part, \$bufferDm) — Connects the buffer datum to the coordinate part specified.

Rectshape Support

See also “ShapeDm” on page 6-37.

The perlutil package contains useful methods for Rectshape objects. A common rectshape object is an input ROI for a step. Multiple ROIs may

also be useful in a Custom Vision Tool as seen in the Mask Generation example (see “Mask Generating Functions” on page 6-14).

- `$int = getInputSearchArea()` — Returns a pointer to the `RectShape` object (`$rectshape`) which is the input search area. This is equivalent to a Shape datum pointer (`$ShapeDm`).
- `DrawMyRoi()` — Draws the ROI associated with the Custom Vision Tool.
- `$float = GetControlPointX ($pointsPtr,$ int index)` — Return the X coordinate of the `index`th control point. Index range: 0-3. `$pointsPtr` returned from `StartControlPoints()`.
- `$float = GetControlPointY ($pointsPtr,$ int index)` — Return the Y coordinate of the `index`th control point. Index range: 0-3. `$pointsPtr` returned from `StartControlPoints()`.
- `$pointsPtr = StartControlPoints($rectshape, $inputBufferDm)` — Get each control point of the `rectshape` and transform to the input buffer coordinate system. A pointer to these points is returned. `DoneControlPoints(...)` must be called after control points are processed to avoid memory leaks.
- `void DoneControlPoints($pointsPtr)` — Must be called whenever `StartControlPoints(...)` is called to avoid memory leaks.
- `void HideRectShape($Rectshape)` — Must call `HideRectShape` for all `rectShape` datums the script has added AND made visible before the specified ROI (`rectshape`) is destroyed. Call this method in the `Cleanup` subroutine of a script.
- `void SetAnchorPointAngle($Rectshape,$ float radAngle)` — Set the angle of the specified `rectshape` to the `radAngle`, which is specified in radians.
- `void SetRectShapeRotateEnable($Rectshape, $bool yesNo)` — If `yesNo` is true, then enable rotation for the specified `rectshape`. If `yesNo` is false, disable rotation.
- `void SetRectShapeAngle ($RectShape, $double_angle)` — Set the angle of the specified ROI (`rectshape`) to `$double_angle` radians.

- `$double = GetRectShapeAngle ($RectShape)` — Return the angle, in radians, of the specified ROI (rectshape).
- `void ShowRectShape ($Rectshape)` — Makes the specified ROI (rectshape) visible to the user. Call this method in the Init subroutine of a script. Must eventually call `HideRectShape($Rectshape)` for each ROI that is made visible. Do not call this method for the default ROI of a Custom Vision Tool.
- `$int = CreateRotatedRectFromInputSearchArea()` — Creates rotated rect object from input ROI information. A pointer to the rotated rect is returned.

Step Access

The following methods provide access to information specific to this Custom Step/Custom Vision Tool.

- `get_input_buf()` — Returns a pointer to the input buffer for this step. Custom Vision Tool only.
- `get_input_bufferdm()` — Returns a pointer to the input buffer datum for this step. Custom Vision Tool only.
- `get_myStepPointer()` — Returns a pointer to the Custom Step or Custom Vision Tool.
- `get_target()` — Returns a pointer to the Vision System or Target step that contains this step.
- `SetPassed(int passflag)` — Allows script to set pass/fail status for this step.
- `get_input_bottom()` — Get the y-coordinate of the bottom of the input search area for this step. Use when the ROI is not rotated.
- `get_input_left()` — Get the x-coordinate of the left of the input search area for this step. Use when the ROI is not rotated.
- `get_input_right()` — Get the x-coordinate of the right of the input search area for this step. Use when the ROI is not rotated.
- `get_input_top()` — Get the y-coordinate of the top of the input search area for this step. Use when the ROI is not rotated.

- `GetInputSearchAreaAngle()` — Returns the angle of the input search area in radians.
- `setInputSearchArea(int l, int t, int r, int b)` — Set the extent of the input search area used by this step. Use when the ROI is not rotated.
- `setInputSearchAreaAngle($double_angle)` — Sets the orientation angle of the input search area in radians to `angle`.
- `setInputSearchAreaRotatable(int_mask)` — If `mask = 0`, disable rotation of the ROI for this Custom Vision Tool. If `mask = 1`, enable rotation of the ROI for this Custom Vision Tool.
- `setInputSearchAreaVisibility(int mask)` — Set the visibility of the input search area (ROI) for this step, as shown in Table 6–4. Valid for Custom Vision Tool only.

TABLE 6–4. Setting Visibility of Input Search Area (ROI)

Mnemonic	Mask Value	Description
VM_NEVER	0x00	// always invisible
VM_PROGRAMMER_ONLY	0x01	// only visible in programmer mode
VM_PROGRAMMER	0x01	// only visible in programmer mode
VM_SUPERVISOR_ONLY	0x02	// visible in supervisor mode only
VM_SUPERVISOR	0x03	// visible in programmer and supervisor
VM_OPERATOR_ONLY	0x04	// visible in operator mode only
VM_OPERATOR	0x07	// visible in all 3 modes
VM_ALWAYS	0xFF	// always visible

These functions get information for any step in the Job; the input `whichstep` is a pointer to a step in the Job. The Composite and Step packages contain methods for getting a pointer to another step: `Composite::FindParentInspectionStep`, `Composite::FindParentOfType`, `Step::FindFromName`, and `Step::FindFromSymName`, to name a few. These functions can be used for the current Custom Step/Custom Vision Tool by passing in the step pointer returned from the `get_myStepPointer()` call.

- `getStepAnchorPointX(int whichStep)` — Return x coordinate of the anchor point (of the input search area) for the specified step. Returns 0.0 if `whichstep` is not valid.

- `getStepAnchorPointY(int whichStep)` — Return y coordinate of the anchor point (of the input search area) for the specified step. Returns 0.0 if whichstep is not valid.
- `getStepInputSearchAreaBottom(int whichStep)` — Get y coordinate of the bottom of the search area for the specified step.
- `getStepInputSearchAreaHeight(int whichStep)` — Return height of the search area for the specified step. Returns 0.0 if whichstep is not valid.
- `getStepInputSearchAreaLeft(int whichStep)` — Get x coordinate of the left of the search area for the specified step.
- `getStepInputSearchAreaRight(int whichStep)` — Get x coordinate of the right of the search area for the specified step.
- `getStepInputSearchAreaTop(int whichStep)` — Get y coordinate of the top of the search area for the specified step.
- `getStepInputSearchAreaWidth(int whichStep)` — Return width of the search area for the specified step. Returns 0.0 if whichstep is not valid.
- `relocatedStepTopLeftX(int whichStep)` — Return x coordinate of the top left corner of the relocated input search area for the specified step. Returns 0.0 if whichstep is not valid.
- `relocatedStepTopLeftY(int whichStep)` — Return y coordinate of the top left corner of the relocated input search area for the specified step. Returns 0.0 if whichstep is not valid.
- `setStepInputSearchArea(int whichStep, int l, int t, int r, int b)` — Set the extent of the input search area of the specified step.
- `setStepInputSearchAreaVisibility(int whichStep, int mask)` — Set the visibility of the input shape of the specified step (see Table 6–4, “Setting Visibility of Input Search Area (ROI),” on page 6-10).
- `prerun(int whichStep)` — PreRun all the child steps of this step.
- `postrun(int whichStep)` — PostRun all the child steps of this step.
- `run(int whichStep)` — Run all the child steps of this step.

Training

A Custom Step / Custom Vision Tool can be made trainable by the script that it runs. The perlutil package contains the following function calls to support trainable custom steps.

- **MakeMeTrainable()** — Makes the Custom Step / Custom Vision Tool trainable. This is called in the Init subroutine of the script.
- **SetTrained()** — Sets the status of a trainable Custom Step / Custom Vision Tool to trained. This is called in the Train subroutine of the script if all train processing succeeded.

When a Custom Step is made trainable, the Perl script it runs must have a Train subroutine.

Utilities/Misc

- **DebugWndMsg(unsigned int flag, char* outputString)** — Sends \$outputString to the debug output. Output is seen in the FrontRunner Debug Window. For a Visionscape GigE Camera, the output can be seen in a Telnet window. Flag indicates the severity of the message and determines whether the message is displayed. The severity determines the display color for the message text. INFO is yellow, WARNING is xxx and SERIOUS is red.

The following are typically defined within a package:

```
sub BEGIN {
    *DWM_INFO           = \1;
    *DWM_WARNING        = \2;
    *DWM_SERIOUS        = \4;
    *DWM_LEVEL          = \15;
    *DWM_SHOWMSG        = \0x00010000;
}
```

This is an example of the use of DebugWndMsg.

```
perlutil::DebugWndMsg($DWM_SERIOUS + $DWM_SHOWMSG, "Message from
script");
```

- **bool GetStartFirsttimeFlag ()** — Return true if this is the first time the custom step has been started (i.e., the Job is ready to run, but has never been run). Otherwise, return false.

- `MarkMonsterMemory()` — The `MarkMonsterMemory()` and `RestoreMonsterMemory()` functions together ensure that a Custom Step/Custom Vision Tool does not leak monster (ASIC) memory. Calling `MarkMonsterMemory()` at the start of the Run subroutine of a Perl script marks the start of free memory on the ASIC before any allocations are done. Calling `RestoreMonsterMemory()` at the end of the Run subroutine of the Perl script restores the ASIC memory back to the state it was in when `MarkMonsterMemory()` was called, regardless of the allocations made in between.
- `register(char *packagename)` — Make package available for use in Perl scripts for a Custom Step / Custom Vision Tool, this needs to be done for every package.
- `RestoreMonsterMemory()` — Restore the monster (ASIC) memory permanent allocation mark to the same location as when `MarkMonsterMemory()` was called. Refer to `MarkMonsterMemory()`.
- `void draw_text_out (int x, int y, char *string)` — Outputs the string onto the display at position (x,y).
- `void ScriptRunsChildSteps($bool yesNo)` — If `yesNo` is true, the child steps of the Custom Step will **not** be run by the Visionscape framework. The script then determines whether or not the child steps are run. If `yesNo` is false, the normal execution model applies.

Note: This function does not run the child steps, but notifies the framework that execution of the child steps is controlled in the script.

Masking Functions

These functions provide the ability to do masked vision steps in Custom Vision Tools. Vision processing that supports masking through Perl are Blob, Sobel and Correlation.

Note: Custom Step does not support this functionality, only Custom Vision Tool.

- `$int_ret = perlutil::CreateMaskBufs();` — Create the appropriate buffers needed to do masking. This is mandatory and must be done in

the PreRun subroutine of the Perl script before the masking agent is PreRun.

- `$int_ret = perlutil::FreeMaskBufs();` — Free the appropriate buffers needed to do masking. This is mandatory and must be done in the PostRun subroutine of the Perl script.
- `$int_ret = perlutil::GetMaskROI();` — Get the ROI from the embedded Mask datum in this step.
- `$int_ret = perlutil::MaskPostRun();` — Do necessary post processing for masking agent. This is mandatory and must be done in the PostRun subroutine of the Perl script.
- `$int_ret = perlutil::MaskPreRun();` — Prepare to run an agent that will use masking. This is mandatory and must be done in the PreRun subroutine of the Perl script before the masking agent is PreRun.
- `$int_ret = perlutil::MaskRun();` — Gather and combine masks for masking agent. This is mandatory and must be done in the Run subroutine of the Perl script before the masking agent is run.
- `$int_ret = perlutil::UpdateMaskBufs();` — Update internal mask information. This is done in the Run subroutine of the Perl script before the masking agent is run.

See “Masking Usage Example [maskable blob]” on page 6-122 and “Masking Usage Example (maskable sobel)” on page 6-125 to help further understand the use of the methods above).

Mask Generating Functions

These functions provide the ability to generate masks in Custom Vision Tools. Multiple masks can be combined into a resultant mask and saved in a Mask datum.

Note: Custom Step does not support this functionality, only Custom Vision Tool.

- `$int_ret = perlutil::CreateMaskList($int_numberOfMasks);` — Allocate a mask list for numberOfMasks masks. A pointer to the mask list is

returned. (Must call DeleteMaskList function later to prevent memory leaks.)

- `$int_ret = perlutil::CreateMaskResources($int_n, $Buffer*_buffer, int_l, int_t, int_r, int_b, int_filltype, int_lowThresh, int_highThresh, int_adjustType, int_numberOfAdjusts);` — Allocate and initialize resources needed for a given mask. A pointer to the resources is returned for use in a subsequent `CreateMask()` call. (Must call `DeleteMaskResources` function later to prevent memory leaks.)

Where:

`$int_n`: index in the mask list.

`$int_l`, `$int_t`, `$int_r`, `$int_b`: position and size of the mask.

`$int_buffer`: buffer holding the mask data.

`$int_filltype`: 0=fill, 1=thresholded.

`$int_lowThresh`, `$int_highThresh`: no longer used.

`$int_adjustType`: how binarized mask is adjusted 0=erode, 1=dilate.

`$int_numberOfAdjusts`: number of erodes or dilates that will be done to adjust the mask.

- `$int_ret = perlutil::CreateMask($int_n, $int_filltype, int_thresh, int_polarity, int_numberOfAdjusts, int_pResources);` — Create the mask by extracting the mask area from the specified buffer, binarizing this data based on the polarity and threshold, and performing all erode or dilate adjustments. A pointer to the pixel data for the mask is returned.

Where:

`$int_n`: index in the mask list.

`$int_filltype`: 0=fill, 1=thresholded.

`$int_thresh`: threshold value.

`$int_polarity`: 0=Light-on-dark, 1=Dark-on-light.

`$int_numberOfAdjusts`: number of erodes or dilates that will be done to adjust the mask.

`$int_pResources`: pointer to the resources set aside for this mask. This is the value that was returned from `CreateMaskResources(...)` when called with the same index in the mask list.

- `void perlutil::AddToMaskList($int_pMaskList, $int_n, $int_maskData, $int_maskOffsetLeft, $int_maskOffsetTop);` — Add the created mask to the mask list at the specified index.

Where:

`$int_pMaskList`: pointer to the mask list returned from `CreateMaskList(...)`.

`$int_n`: index in the mask list.

`$int_maskData`: pointer to the pixel data for this mask returned from `CreateMask(...)`.

`$int_maskOffsetLeft`: x offset for this mask from the upper left corner of the input ROI.

`$int_maskOffsetTop`: y offset for this mask from the upper left corner of the input ROI.

- `$int_ret = perlutil::CombineMasks($int_numMasks, $int_pMaskList, $int_pMaskDatum);` — Combine all masks in the input mask list and save the result in the specified Mask datum.

Where:

`$int_numMasks`: number of valid masks in the mask list.

`$int_pMaskList`: pointer to the mask list returned from `CreateMaskList(...)`.

`$int_pMaskDatum`: pointer to the output Mask datum for this Custom Vision Tool.

- `void perlutil::DeleteMaskResources($int_pResources);` — Free the resources for a given mask.

Where:

`$int_pResources`: pointer to the resources set aside for this mask. This is the value that was returned from `CreateMaskResources(...)` when called with the same index in the mask list.

- `void perlutil::DeleteMaskList($int_pMaskList);` — Delete the mask list that was allocated in the `CreateMaskList(...)` function.

`$int_pMaskList`: pointer to the mask list returned from `CreateMaskList(...)`.

See “Mask Generating Functions” on page 6-14 to help further understand the use of the above methods.

System Timers

The functions provide access to system timing. These functions can be used to determine the processing time of a custom step.

- `$double_time = perlutil::AcuElapsedTime($double startTime);` — Return the elapsed time in seconds as a double. (For example: 0.0023 = 2.3 milliseconds). This time is accurate to the frequency of the CPU clock.
- `$double_time = perlutil::AcuGetClock();` — Return the current time as a double.
- `$long_ret = perlutil::AcuElapsedSince(long startTick);` — Return the number of milliseconds since `startTick`.
- `$long_ret = perlutil::AcuTick();` — Return the number of milliseconds since boot time.
- `$void = perlutil::AcuMSecSleep(long millisecondsToSleep);` — Sleep (give up the CPU) for the indicated number of milliseconds. This is fairly accurate on the PC; however, on the target this function will wait to the nearest 16msec boundary of what you asked for. For instance if you request a 40 millisecond sleep, you sleep for 48 milliseconds on the target.

Composite-Datum-Step Packages (Common Functions)

Composite — Common Functions For Datums or Steps

The composite package contains many functions that are common to steps and datums. The functions in this package can be used whenever you have a pointer to a step or datum object, referred to as `$Composite_` below.

- `$bool_ret = Composite::CheckVisibility ($Composite_, $VisibilityMask_mask);` — Return the “mask and visibility” status of this composite.
- `$bool_ret = Composite::GetStatus ($Composite_);` — Return status of this composite or its owner.
- `$bool_ret = Composite::IsIncludedInResultsUpload ($Composite_);` — Return true if this composite is included as part of the current results upload.
- `$bool_ret = Composite::IsRef ($Composite_);` — Return true if this composite is a reference to another datum.
- `$bool_ret = Composite::IsScalar ($Composite_);` — Return true if this composite contains a scalar value.
- `$bool_ret = Composite::IsTrainable ($Composite_);` — Return true if this composite is trainable.
- `$bool_ret = Composite::IsTrained ($Composite_);` — Return true if this composite is trained.
- `$bool_ret = Composite::Passed ($Composite_);` — Return true if this composite passed; false if this composite failed.
- `$float_ret = Composite::GetScalar ($Composite_);` — Return the scalar value for this composite object.
- `$int_ret = Composite::FindFromName ($Composite_, $string_name);` — Return a pointer to the composite object with the given name.
- `$int_ret = Composite::FindFromSymName ($Composite_, $string_symbolicName);` — Return a pointer to the composite object with the given symbolic name.

- `$int_ret = Composite::FindParentInspectionStep ($Composite_);` — Return a pointer to the inspection step that contains this composite.
- `$int_ret = Composite::FindParentOfType ($Datum_, $int_typeIndex, $int_followRef);` — Return a pointer to the step that is the parent of this composite and of type `$int_typeIndex` (0=VisionSystemStep, 1=InspectionStep, 2=SnapshotStep). If `$int_followRef` is nonzero, then return a pointer to the step that is the parent to the composite that `$Datum` is referencing.
- `$int_ret = Composite::GetInspection($Composite_);` — Return a pointer to the Inspection step that contains this Custom Tool.
- `$int_ret = Composite::GetLastError ($Composite_);` — Return the last error code for this composite object.
- `$int_ret = Composite::GetSystem ($Composite_);` — Return a pointer to the target step (VisionSystemStep) of this composite. This is useful to access I/O, etc.
- `$int_ret = Composite::GetVisibility ($Composite_);` — Return the visibility status of this composite.
- `$int_ret = Composite::Name ($Composite_);` — Return the name of this composite as a pointer to a character string.
- `$int_ret = Composite::OwnerPart ($Composite_);` — Return a pointer to the owner part of this composite (located datums only, i.e., points, lines, areas, distances, and angles).
- `$int_ret = Composite::Parent ($Composite_);` — Return a pointer to the composite object that is the parent of this composite.
- `$int_ret = Composite::RefTo ($Composite_);` — Return a pointer to the composite that this composite is referencing.
- `$int_ret = Composite::FindInspectionStepN ($VisionSystemStep_, int n);` — Return a pointer to the nth inspection step that is contained in the given target (VisionSystemStep) step. Zero (null) is returned if the nth inspection step is not found.
- `$int_ret = Composite::FindSnapshotStepN ($VisionSystemStep_, int n);` — Return a pointer to the nth snapshot step that is contained in the

given target (VisionSystemStep) step. Zero (null) is returned if the nth snapshot step is not found.

- `Composite::NextHigherName ($Composite_, $string_name);` — Prepend the name of the next higher object from the Job Tree to the input string `$string_name`.
- `Composite::NextHigherSymName ($Composite_, $string_symbolicname);` — Prepend the symbolic name of the next higher object from the Job Tree to the input string `$string_name`.
- `Composite::SetLastError ($Composite_, $int_lasterr);` — Set internal error code for this composite.
- `Composite::SetName ($Composite_, "name");` — Set name of this composite to "name".
- `Composite::SetOwnerPart ($Composite_, $PartStep_p);` — Set owner part (system of coordinates) of this composite to `$PartStep_p`. (Located datums only, i.e., points, lines, areas, distances, and angles).
- `Composite::SetSymName ($Composite_, $string_symbolicName);` — Set symbolic name of this composite to "symbolicName".
- `Composite::SetVisibility ($Composite_, $VisibilityMask_mask);` — Set the visibility status of this composite.

Datum — Common Functions For Datums

The datum package contains functions that are common for all types of datums. The functions in this package can be used with a pointer to any datum type, referred to as `$Datum_` below.

- `$bool_ret = Datum::stringCreate(int size);` — Create a string of the given size and return a pointer to it.
- `$bool_ret = Datum::GetStatus ($Datum_);` — Return the status of the step that holds this datum, or true if there is no owner step.
- `$bool_ret = Datum::GetValueByString ($Datum_, "value");` — Return true if successful with the value of the datum returned in "value" as an ASCII string.

- `$bool_ret = Datum::IsDatum ($Datum_);` — Return true if this object is a datum.
- `$bool_ret = Datum::IsDatumValid ($Datum_);` — Return true if the step that holds this datum is valid, or if there is no owner step.
- `$bool_ret = Datum::Passed ($Datum_);` — Return status of this datum.
- `$bool_ret = Datum::SetValueByString ($Datum_, "value");` — Set the value of the datum from the ASCII string "value".
- `$int_ret = Datum::GetLastError ($Datum_);` — Get the last error code from the datum.
- `$int_ret = Datum::Owner ($Datum_);` — Return a pointer to the object that owns this datum.
- `Datum::GetValue ($Datum_, "fieldname", $double_val);` — Return the value of the specified fieldname into `$double_val`. Fieldname might be "x" for a point datum, for example. If the fieldname does not exist, then `$double_val` is not updated.
- `Datum::NameFromStep ($Datum_, name);` — Return the name of this object. The input name is a character array for holding the result string.
- `Datum::NextHigherName ($Datum_, $string_name);` — Prepend the name of the next higher object from the Job Tree to the input string `$string_name`.
- `Datum::NextHigherSymName ($Datum_, $string_symbolicName);` — Prepend the symbolic name of the next higher object from the Job Tree to the input string `$string_name`.
- `Datum::PrecheckOff ($Datum_);` — Disable prechecking for this datum.
- `Datum::PrecheckOn ($Datum_);` — Enable prechecking for this datum.
- `Datum::SetValue ($Datum_, "fieldname", $double_newval);` — Set the value of the specified fieldname with `$double_val`. Fieldname might be "x" for a point datum, for example. If the fieldname does not exist, then no action is taken.

- `Datum::SymNameFromStep ($Datum_, $symname);` — Return the symbolic name of this object. The input `symname` is a character array for holding the result string.
- `Datum::Train ($Datum_);` — Train this object.

Step — Common Functions For Steps

The step package contains functions that are common for all types of steps. The functions in this package can be used with a pointer to any step type, referred to as `$Step_` below.

- `$bool_ret = Step::CanMask ($Step_);` — Return true if this step can be masked, false otherwise.
- `$bool_ret = Step::CanRotate ($Step_);` — Return true if this step can be rotated, false otherwise.
- `$bool_ret = Step::GetStatus ($Step_);` — Return the status of this step.
- `$int_ret = Step::GetParentSnapshotPart(Step *pstep);` — Return pointer to PART (coordinate system) of the snapshot step which is a parent to the input step. The search is done using the Step Tree - i.e. follows the step containment. This function returns 0 if no snapshot is found. (A similar function exists in the Part2DStep package, `Part2DStep::GetParentSnapshotPart(Datum*pdatum)`).

Sample usage:

```
# Get pointer to the Custom Vision Tool
$psstep = perlutil::get_myStepPointer();
# Get PART of parent snapshot (via the STEP tree)
$spPart = Step::GetParentSnapshotPart($psstep);
```

- `$bool_ret = Step::IgnorePreprocFailures ($Step_, $Step_pStep);` — Return true if this step is configured to ignore failures in preprocessing steps. Return false otherwise.
- `$bool_ret = Step::IsReadyToRun ($Step_);` — Return true if this step is ready to run. Return false otherwise.
- `$bool_ret = Step::IsSetupStep ($Step_);` — Return true if this is a setup step. Return false otherwise.

- `$bool_ret = Step::NeedStatTrain ($Step_);` — Return true if the step needs statistical training, false otherwise.
- `$bool_ret = Step::Passed ($Step_);` — Return passed/failed status for this step. True=passed, false=failed.
- `$int_ret = Step::FindFromName ($Step_, $string_name);` — Search through the sub-steps and sub-datums of this step and return a pointer to the object (Composite*) that has the specified name. Return 0, if unsuccessful.
- `$int_ret = Step::FindFromSymName ($Step_, $string_symbolicName);` — Search through the sub-steps and sub-datums of this step and return a pointer to the object (Composite*) that has the specified symbolic name. Return 0, if unsuccessful.
- `$int_ret = Step::GetLastError ($Step_);` — Return the last error code generated internally by this step.
- `$int_ret = Step::GetRoot ($Step_);` — Returns a pointer to the root step.
- `$int_ret = Step::GetTarget ($Step_);` — Returns a pointer to the target (Vision System) step.
- `Step::NextHigherName ($Step_, $string_name);` — Prepend the name of the next higher object from the Job Tree to the input string `$string_name`.
- `Step::NextHigherSymName ($Step_, $string_name);` — Prepend the symbolic name of the next higher object from the Job Tree to the input string `$string_name`.

Datums

Datums pass information from step to step. The functions available for each datum package, the parameters and their type, and a brief description of that function are documented here.

Basic Datums

Buffer Datums

Buffer datums hold image and graphics information. The `get_datum_buffer` or `get_datum` functions in the `perlutil` package get a pointer to a buffer datum (shown as `$bufferdm_` below).

- `$int_ret = BufferDm::GetBuffer ($BufferDm_);` — Return a pointer to the buffer in this buffer datum.
- `$int_ret = BufferDm::GetFailBuf ($BufferDm_);` — Return a pointer to the buffer containing the image from the last inspection failure.
- `$int_ret = BufferDm::Height ($BufferDm_);` — Return the height in pixels of the buffer in this buffer datum.
- `$int_ret = BufferDm::OwnerPart ($BufferDm_);` — Return a pointer to the coordinate system of buffer datum. A pointer to `PartStep` is returned.
- `$int_ret = BufferDm::Parent ($BufferDm_);` — Return pointer to the parent of the buffer datum.
- `$unsigned char_ret = BufferDm::PixVal ($BufferDm_, $int_x, $int_y);` — Return the pixel value at the position (`$int_x`, `$int_y`) in the buffer datum.
- `BufferDm::CrossAt ($BufferDm_, $double_xd, $double_yd, $unsigned_int);` — Draw a cross located at (`$double_xd`, `$double_yd`) that is `$unsigned_int` pixels long.
- `BufferDm::Ellipse ($BufferDm_, $double_x1d, $double_y1d, $double_x2d, $double_y2d);` — Draw an ellipse specified by the x and y coordinates of two points.
- `BufferDm::LineTo ($BufferDm_, $double_xd, $double_yd);` — Draw a line in the buffer datum from the current location to the point (`$double_xd`, `$double_yd`).
- `BufferDm::MoveTo ($BufferDm_, $double_xd, $double_yd);` — Move to the location (`$double_xd`, `$double_yd`) in `bufferdm`; usually in preparation for a draw.

- `BufferDm::Rectangle ($BufferDm_, $double_x1d, $double_y1d, $double_x2d, $double_y2d);` — Draw a rectangle in the buffer datum with the top-left and bottom-right points specified.
- `BufferDm::SetBrushColor ($BufferDm_, $int_r, $int_g, $int_b);` — Set brush color for drawing to the RGB color specified.
- `BufferDm::SetBuffer ($BufferDm_, $Buffer_pBuf);` — Set the buffer in the buffer datum to `$Buffer_pBuf`.
- `BufferDm::SetDirty ($BufferDm_);` — `SetDirty` signals the UI that this `bufferDm` needs to be refreshed.
- `BufferDm::SetFailBuf ($BufferDm_, $Buffer_pBuf);` — Set the fail buffer of the buffer datum to `pBuf`.
- `BufferDm::SetOwnerPart ($BufferDm_, $PartStep_part);` — Set the coordinate system of this buffer.
- `BufferDm::SetPenColor($BufferDm_, $int_r, $int_g, $int_b);` — Set pen color for drawing to the RGB color specified.
- `BufferDm::SetPixel($BufferDm_, $int_x, $int_y, $int_r, $int_g, $int_b);` — Set pixel at location `($int_x, $int_y)` to the RGB color specified.
- `BufferDm::TextOut ($BufferDm_, $int_x, $int_y, "string", $int_count);` — Draw the specified string into the buffer at location `($int_x, $int_y)`. If `count=0`, then the entire string is written; otherwise, `count` characters are written.

DoubleDm

`DoubleDm` holds a floating point value. The `get_datum_double` and `get_datum` functions in the `perlutil` package get a pointer to a double datum (shown as `$doubledm_` below).

- `$bool_ret = DoubleDm::GetValueByString ($DoubleDm_, "value");` — Get value of datum as an ASCII string in "value"; return true if successful, false otherwise.
- `$bool_ret = DoubleDm::SetValueByString ($DoubleDm_, "value");` — Set value of datum to value specified in ASCII string. For example, "3.5" sets value to 3.5.

- `$double_ret = DoubleDm::GetDefaultValue ($DoubleDm_);` — Get default value for this datum.
- `$double_ret = DoubleDm::GetMaxValue ($DoubleDm_);` — Get maximum allowable value for this datum.
- `$double_ret = DoubleDm::GetMinValue ($DoubleDm_);` — Get minimum allowable value for this datum.
- `$double_ret = DoubleDm::GetValue ($DoubleDm_);` — Get value of datum.
- `$float_ret = DoubleDm::GetScalar ($DoubleDm_);` — Get scalar value from the datum.
- `DoubleDm::SetDefaultValue ($DoubleDm_, $double_m);` — Set default value for this datum.
- `DoubleDm::SetMaxValue ($DoubleDm_, $double_m);` — Set maximum allowable value for this datum.
- `DoubleDm::SetMinValue ($DoubleDm_, $double_m);` — Set minimum allowable value for this datum.
- `DoubleDm::SetRange ($DoubleDm_, $double_minval, $double_maxval);` — Set range of values allowable value for this datum.
- `DoubleDm::SetToDefault ($DoubleDm_);` — Set datum to its default value.
- `DoubleDm::SetValue ($DoubleDm_, $double_value);` — Set value of datum to `$double_value`.

EnumDm

EnumDm holds an enumerated type value. The selections for an EnumDm are specified by an array of strings. This allows the user to configure the EnumDm from a list of meaningful selections instead of an unrelated index. The `get_datum_enum` and `get_datum` functions in the `perlutil` package get a pointer to an enumerated datum (shown as `$enumdm_` below).

- `$bool_ret = EnumDm::GetValueByString ($EnumDm_, "value");` — Get the value of the enum datum as an ASCII string, into `value`. If

successful, true is returned. If the function fails, then false is returned and input string is unchanged.

- `$bool_ret = EnumDm::SetValueByString ($EnumDm_, "value");` — Set value of datum to value specified in ASCII string "value". If string does not match one of the selections within the EnumDm, then no action is taken. The return value is always true.
- `$int_ret EnumDm::GetValue ($EnumDm);` — Return the value of the enum datum. This is essentially the index of the current selection in the datum.
- `EnumDm::SetValue ($EnumDm_, $int_value);` — Set the value of the enum datum with \$int_value.

IntDm

IntDm holds an integer value. The `get_datum_int` and `get_datum` functions in the `perlutil` package get a pointer to an integer datum (shown as `$intdm_` below).

- `$bool_ret = IntDm::GetValueByString ($IntDm_, "value");` — Get value of integer datum as an ASCII string.
- `$bool_ret = IntDm::SetValueByString ($IntDm_, "value");` — Set the value of the integer datum by the string "value". For example, "67" sets datum to 67.
- `$int_ret = IntDm::GetDefaultValue ($IntDm_);` — Get the default value for this datum.
- `$int_ret = IntDm::GetMaxValue ($IntDm_);` — Get the maximum allowable value for this datum.
- `$int_ret = IntDm::GetMinValue ($IntDm_);` — Get the minimum allowable value for this datum.
- `$int_ret = IntDm::GetValue ($IntDm_);` — Get value of integer datum.
- `IntDm::SetDefaultValue ($IntDm_, $int_m);` — Set the default value for this datum.
- `IntDm::SetMaxValue ($IntDm_, $int_m);` — Set the maximum allowable value for this datum.

- `IntDm::SetMinValue ($IntDm_, $int_m);` — Set the minimum allowable value for this datum.
- `IntDm::SetRange ($IntDm_, $int_minval, $int_maxval);` — Set the allowable range of values for this datum.
- `IntDm::SetToDefault ($IntDm_);` — Set the datum to its default value.
- `IntDm::SetValue ($IntDm_, $int_value);` — Set the value of the integer datum to `$int_value`.

StringDm

String Dm holds a string. The `get_datum_string` and `get_datum` functions in the `perlutil` package get a pointer to a string datum (shown as `$stringdm_` below).

- `$bool_ret = StringDm::GetValueByString ($StringDm_, "value");` — Copy the string from the string datum to the input string "value".
- `$bool_ret = StringDm::SetValueByString ($StringDm_, "value");` — Set the string in the string datum to the string specified ("value").
- `$int_ret = StringDm::GetValue ($StringDm_);` — Return the string in the string datum.
- `StringDm::SetValue ($StringDm_, "str");` — Set the string in the string datum to the string specified ("str").

StatusDm

StatusDm holds a boolean status. The `get_datum_status` and `get_datum` functions in the `perlutil` package get a pointer to a status datum (shown as `$statusdm_` below).

- `$bool_ret = StatusDm::GetStatus ($StatusDm_);` — Return the value/status of the status datum.
- `$float_ret = StatusDm::GetScalar ($StatusDm_);` — Return the scalar value of the status datum.

Location Datums

Location datums have a coordinate system (part2dStep) associated with them.

AngleDm

AngleDm represents angle values. The angle can be calibrated. The `get_datum` function in the `perlutil` package gets a pointer to an angle datum (shown as `$angledm_` below).

- `$double_ret = AngleDm::GetAngleDeg ($AngleDm_);` — Get the angle in degrees.
- `$double_ret = AngleDm::GetAngleRad ($AngleDm_);` — Get the angle in radians.
- `$double_ret = AngleDm::GetSingularDirRad ($AngleDm_);` — Get singular direction.
- `$float_ret = AngleDm::GetScalar ($AngleDm_);` — Get the scalar value of this angle datum.
- `$int_ret = AngleDm::CoordPart ($AngleDm_);` — Return a pointer to the angle datum part coordinate system.
- `$int_ret = AngleDm::GetCoordPart ($AngleDm_);` — Return a pointer to the angle datum part coordinate system.
- `$int_ret = AngleDm::OwnerPart ($AngleDm_);` — Return the owner part coordinate system step for the angle datum.
- `AngleDm::SetAngleDeg ($AngleDm_, $double_angDeg);` — Set angle in degrees.
- `AngleDm::SetAngleRad ($AngleDm_, $double_angRad);` — Set angle in radians.
- `AngleDm::SetOwnerPart ($AngleDm_, $PartStep_part);` — Set the owner part coordinate system for the angle datum.
- `AngleDm::SetSingularDirRad ($AngleDm_, $double_singDirInDeg);` — Set singular direction.

AreaDm

AreaDm represents an area value. An area value is a double. The `get_datum_area` and `get_datum` functions in the `perlutil` package get a pointer to an area datum (shown as `$areadm_` below).

- `$double_ret = AreaDm::GetArea ($AreaDm_);` — Get the area value.

- `$float_ret = AreaDm::GetScalar ($AreaDm_);` — Get the scalar value from this datum.
- `$int_ret = AreaDm::CoordPart ($AreaDm_);` — Return a pointer to the area datum part coordinate system.
- `$int_ret = AreaDm::GetCoordPart ($AreaDm_);` — Return a pointer to the area datum part coordinate system.
- `$int_ret = AreaDm::OwnerPart ($AreaDm_);` — Return the owner part coordinate system step for the area datum.
- `AreaDm::SetArea ($AreaDm_, $double_area);` — Set the area value.
- `AreaDm::SetOwnerPart ($AreaDm_, $PartStep_part);` — Set the owner part coordinate system for the area datum.

DistanceDm

DistanceDm holds a distance value, which is a double value. The distance can be calibrated. The `get_datum_distance` and `get_datum` functions in the `perlutil` package get a pointer to a distance datum (shown as `$distancedm_` below).

- `$double_ret = DistanceDm::GetDistance ($DistanceDm_);` — Get distance value from the distance datum.
- `$float_ret = DistanceDm::GetScalar ($DistanceDm_);` — Get scalar value from the datum.
- `$int_ret = DistanceDm::CoordPart ($DistanceDm_);` — Return a pointer to the distance datum part coordinate system.
- `$int_ret = DistanceDm::GetCoordPart ($DistanceDm_);` — Return a pointer to the distance datum part coordinate system.
- `$int_ret = DistanceDm::OwnerPart ($DistanceDm_);` — Return the owner part coordinate system for the distance datum.
- `DistanceDm::SetDistance ($DistanceDm_, $double_distV);` — Set distance value for distance datum.
- `DistanceDm::SetDistancePt ($DistanceDm_, double distance, double x1, double y1, double x2, double y2);` — Set distance with two points

[(x1,y1), (x2,y2)]. DistanceDm uses these points when computing a ratio from pixel coordinates to world coordinates.

- DistanceDm::SetOwnerPart (\$DistanceDm_, \$PartStep_part); — Set the owner part coordinate system for the distance datum.
- DistanceDm::Transform (\$DistanceDm, \$double_distance, \$bool_forward) — If forward is true, then transform \$double_distance from pixel to world coordinate space. If forward is false, then transform \$double_distance from world to pixel coordinate space.

Note: This assumes the transform is non-skewed. Before a DistanceDm can be transformed, the coordinate system must be set up using the DistanceDm::SetOwnerPart() call. In the following example, the coordinate system (part) is taken from the input buffer datum of the Custom Vision Tool, where \$pDistanceDm is a pointer to the distance datum to transform:

```
$inbufdm = perlutil::get_input_bufferdm();
$calPart = BufferDm::OwnerPart ($inbufdm);
DistanceDm::SetOwnerPart ($pDistanceDm, $calPart);
$calDist = DistanceDm::Transform ($pDistanceDm, $double_dist,
$bool_forward);
```

Linedm

Linedm holds a line. The line can be calibrated. The get_datum_line and get_datum functions in the perlutil package get a pointer to a line datum (shown as \$linedm_ below).

- \$int_ret = LineDm::GetLinePtr (\$LineDm_); — Return a pointer to the line contained in the datum.
- \$int_ret = LineDm::OwnerPart (\$LineDm_); — Return the owner part coordinate system step for the line datum.
- LineDm::SetLine (\$LineDm_, \$Line_pline); — Set the line contained in the line datum to \$Line_pline.
- LineDm::SetOwnerPart (\$LineDm_, \$PartStep_part); — Set the owner part coordinate system for the line datum.

PointDm

PointDm holds a point. The point can be calibrated. The `get_datum_point` and `get_datum` functions in the `perlutil` package get a pointer to a point datum (shown as `$pointdm_` below).

- `$double_ret = PointDm::AngDeg ($PointDm_);` — Get the angle of a PointDm in degrees.
- `$double_ret = PointDm::AngRad ($PointDm_);` — Get the angle of a PointDm in radians.
- `$double_ret = PointDm::GetPoint ($PointDm_);` — Return a pointer to the embedded point<double> in this point datum.
- `$double_ret = PointDm::X ($PointDm_);` — Return the x component of the PointDm.
- `$double_ret = PointDm::Y ($PointDm_);` — Return the y component of the PointDm.
- `$int_ret = PointDm::CoordPart ($PointDm_);` — Get a pointer to the PointDm's part coordinate system.
- `$int_ret = PointDm::OwnerPart ($PointDm_);` — Get a pointer to the PointDm's owner part coordinate system.
- `$int_ret = PointDm::PointDm ("name");` — Construct a PointDm with the input name and get back a pointer to it.
- `PointDm::IncrAng ($PointDm_, $double_dang);` — Increment the angle of the PointDm by the input dang. The input must be in radians.
- `PointDm::IncrX ($PointDm_, $double_dx);` — Increment the x component of the PointDm by the input dx.
- `PointDm::IncrY ($PointDm_, $double_dy);` — Increment the y component of the PointDm by the input dy.
- `PointDm::SetAngRad ($PointDm_, $double_orient);` — Set the angle of a PointDm in radians.
- `PointDm::SetOwnerPart ($PointDm_, $PartStep_part);` — Set the owner part coordinate system for the point datum.

- `PointDm::SetPoint ($PointDm_, $Point<double>);` — Set the embedded point<double> in the point datum.
- `PointDm::SetXY ($PointDm_, $double_x, $double_y);` — Set the x and y components of the point datum.

Part2DStep

Part2DStep represents a coordinate system. Location datums have methods `OwnerPart()` and `SetOwnerPart()` that allow access to the part2dstep object.

- `$double_ret = Part2DStep::Angle ($Part2DStep_);` — Get the angle of a Part2DStep.
- `$int_ret = Part2DStep::FindCommonAncestor ($Part2DStep_, $PartStep_part);` — Get the common ancestor, or parent part of the input part and the Part2DStep.
- `$int_ret = Part2DStep::Part2DStep ();` — Construct a Part2DStep and get back a pointer to it.
- `Part2DStep::Part2DStep_Delete ($Part2DStep_);` — Delete a Part2DStep.
- `$int_ret = Part2DStep::GetParentSnapshotPart(Datum*pdatum);` — Return pointer to PART (coordinate system) of the snapshot step which is a parent to the input datum. The search is done using the Part Tree - i.e. follows the system of coordinates containment. This function returns 0 if no snapshot is found or if the input datum does not have an associated system of coordinates. (A similar function also exists in the Step package, `Step::GetParentSnapShotPart(Step *pStep)`)

Sample usage:

```
# Get pointer to a datum. Ex: use 0th 1st 2nd input datum to a
Custom Vision Tool
$pdatum = perlutil::get_datum(0, 2);
# Get PART of parent snapshot (via the PART tree)
$spPart = Part2Dstep::GetParentSnapshotPart($pdatum);
```

- `Part2DStep::ComposeTransform ($Part2DStep_, $Part2DStep_part2);` — Compose the transform needed to go to and

from this part coordinate system and `$Part2DStep_part2` coordinate system.

- `Part2DStep::InvTransformPoint ($Part2DStep_, $PointDm_ptdm);` — Convert the input point to this part coordinate system.
- `Part2DStep::SetCoord ($Part2DStep_, $PointDm_coord);` — Set the coordinates of the `Part2DStep` to the coordinates specified by `$PointDm_coord`.
- `Part2DStep::SetFwdandBkwdTransforms ($Part2DStep_, $Matrix_coord, $Matrix&_invcoord);` — Set the forward transform of the `Part2DStep` to `coord` and the backward transform to `invcoord`.
- `Part2DStep::SetIdentity ($Part2DStep_);` — Set up the `Part2DStep` transforms to the identity matrix, meaning that the part coordinate system will be the same as its parent.
- `Part2DStep::SetTransform ($Part2DStep_, $Matrix_coord);` — Set the `Part2DStep` transform to the input `coord`.
- `Part2DStep::TransformLine ($Part2DStep_, $LineDm_dmLineIn, $LineDm_dmLineOut, $PartStep_part);` — Transform a line datum to the specified part coordinate system. The part is assumed to be an ancestor of the current part or itself. The new datum is returned in `dmLineOut`.
- `Part2DStep::TransformLine2 ($Part2DStep_, $Line_lineInOut, $PartStep_part);` — Transform a line datum to the specified part coordinate system. The part is assumed to be an ancestor of the current part or itself. The new values are returned in the same input line datum, `lineInOut`.
- `Part2DStep::TransformPoint ($Part2DStep_, $point<double>*_inpoint, $Part2DStep_);` — Transform a point to the specified part coordinate system. The part is assumed to be an ancestor of the current part. The new values to the transform point overwrite the input.
- `Part2DStep::TransformPoint2 ($Part2DStep_, $PointDm_sptdm, $PointDm_dptdm);` — Use the coordinate transforms in the `Part2DStep` to transform the input point datum, `sptdm`. The output value is returned in the point datum `dptdm`.

- `Part2DStep::TransformPointList ($Part2DStep_, $PtListDm_dmPtListIn, $PtListDm_dmPtListOut, $PartStep_part);` — Transform a point list datum, `dmPtListIn`, to the specified part coordinate system. The part is assumed to be an ancestor of the current part or itself. The new values are returned in the point list datum `dmPtListOut`.
- `Part2DStep::Translate ($Part2DStep_, $double_tx, $double_ty);` — Move, or translate, the `Part2DStep` coordinate system horizontally by `tx` and vertically by `ty`.
- `Part2DStep::Update ($Part2DStep_, $PartStep_part);` — Copy the part transforms from `$PartStep_part` into `$Part2DStep`.
- `Part2DStep::UpdateFromInputROI ($Part2DStep_, $ShapeDm *inputRoi)` — Update the matrices of the part coordinate system with the scale, rotation, translation from the input ROI.
- `Part2DStep::XformLine ($Part2DStep_, $LineDm_inLineDm, $Line_xformedLine, $bool_forward);` — Use the coordinate transforms in the `Part2DStep` to transform the input line datum. Depending on the value of the input `forward`, do either a forward or backward transform. The output value is returned in the `Line_xformedLine`.

Advanced Datums

BlobDm

`BlobDm` holds a blob. The `get_datum` function in the `perlutil` package gets a pointer to a blob datum (shown as `$blobdm_` below).

- `$int_ret = BlobDm::GetBlob ($BlobDm_);` — Return a pointer to the blob contained in this blob datum.
- `BlobDm::SetBlob ($BlobDm_, $Blob_theblob);` — Set the blob contained in this blob datum to `theblob`. Input, `theblob`, is a pointer to a blob object.

BlobTreeDm

`BlobTreeDm` holds a `blobresult`. The `get_datum` function in the `perlutil` package gets a pointer to a `blobtree` datum (shown as `$blobtreedm_` below). A `blobtreedm` would be created by a blob step.

- `$int_ret = BlobTreeDm::GetBlobTree ($BlobTreeDm_);` — Returns a pointer to the BlobResult in this blobtree datum.
- `BlobTreeDm::SetBlobTree ($BlobTreeDm_, $BlobResult_pBlobRes);`
— Set the BlobResult in this blobtree datum.

PtListDm

PtListDm holds a point list, which is a list of points represented by double values (x,y). The `get_ptlistdm` and `get_datum` functions in the `perlutil` package get a pointer to a ptlist datum (shown as `$ptlistdm_` below). Tools that create a ptlistdm include the `flawtool`, `edgestep` and `bgastep`.

- `$double *_ret = PtListDm::GetPointListByArray ($PtListDm_);` — Allocates an array and fills it with the points from the point list datum. Must be paired with the `FreePointListByArray()` function to avoid memory leaks.
- `$double_ret = PtListDm::GetPointListX ($double_pArray, $int_index);`
— Returns the x coordinate of the indexth point in the array returned by the `GetPointListByArray` call.
- `$double_ret = PtListDm::GetPointListY ($double_pArray, $int_index);`
— Returns the y coordinate of the indexth point in the array returned by the `GetPointListByArray` call.
- `$int_ret = PtListDm::GetSize ($PtListDm_);` — Return the number of points in the pointlist datum.
- `PtListDm::AddPoint ($PtListDm_, $double_x, $double_y);` — Add a point created from x and y to the pointlist datum.
- `PtListDm::Draw ($PtListDm_, $BufferDm_dstBuf, $Matix*_pMat);` — Draw the points in the pointlist datum into the specified buffer datum. As an optional parameter, a matrix may be specified to transform the points to a given coordinate space.
- `PtListDm::Empty ($PtListDm_);` — Remove all points from the pointlist datum.
- `PtListDm::FreePointListByArray ($double* plist);` — Deallocates the array setup by the `GetPointListByArray()` call. These calls must be paired together to avoid memory leaks. The input to the function is the return value from `GetPointListByArray()`.

- `PtListDm::SetBoundingRect ($PtListDm_, $Rect&_rect);` — Set the bounding rectangle for the point list datum.
- `PtListDm::SetPointListByArray ($PtListDm_, $double_pArray, $int_size);` — Setup the pointlist inside the pointlist datum with the data in `pArray`. Size indicates the number of points represented by the data in `pArray`.

ShapeDm

The `get_datum_rectshape` and `get_datum` functions in the `perlutil` package get a pointer to a shape datum (shown as `$shapedm_` below).

- `$double_ret = ShapeDm::GetAnchorPointX ($ShapeDm);` — Return the x coordinate of the anchor point of the shape.
- `$double_ret = ShapeDm::GetAnchorPointY ($ShapeDm);` — Return the y coordinate of the anchor point of the shape.
- `$int_ret = ShapeDm::CalcTransform ($ShapeDm_, $BufferDm_dmbuf);` — Return a pointer to the partstep (coordinate system) that maps the shape to the specified buffer, if possible. Zero is returned if the operation was not successful.
- `$int_ret = ShapeDm::FindOutputBuffer ($ShapeDm_);` — Return a pointer to the buffer datum that originates from this shape.
- `$int_ret = ShapeDm::GetBoundingRectBottom ($ShapeDm);` — Return the y coordinate of the bottom side of the shape's bounding rectangle.
- `$int_ret = ShapeDm::GetBoundingRectLeft ($ShapeDm);` — Return the x coordinate of the left side of the shape's bounding rectangle.
- `$int_ret = ShapeDm::GetBoundingRectRight ($ShapeDm);` — Return the x coordinate of the right side of the shape's bounding rectangle.
- `$int_ret = ShapeDm::GetBoundingRectTop ($ShapeDm);` — Return the y coordinate of the top side of the shape's bounding rectangle.
- `void ShapeDm::SetBoundingRect2 ($ShapeDm, $int_l, $int_t, $int_r, $int_b);` — Set the shape's bounding rectangle to the rectangular area enclosed by `$l`, `$t`, `$r` and `$b`.

TemplateDm — Template Datum

- `$bool_ret = TemplateDm::TrainTemplate ($TemplateDm_);` — Train the template datum. This creates necessary internal templates. Return true if successful, false otherwise.
- `$bool_ret = TemplateDm::TrainTemplate2 ($TemplateDm_, $Buffer_, $int_l, $int_t, $int_r, $int_b);` — Train the template datum based on the image information contained in the specified rectangular area with the input buffer. This creates necessary internal templates. Return true if successful, false otherwise.
- `$int_ret TemplateDm::GetBuffer ($TemplateDm);` — Return a pointer to the template buffer, (i.e., trained image), in the specified template datum.
- `$int_ret TemplateDm::GetMatchTempl ($TemplateDm_);` — Return a pointer to the match template in the specified template datum.
- `$int_ret TemplateDm::GetSearchTempl ($TemplateDm);` — Return a pointer to the search template in the specified template datum.
- `TemplateDm::Reset ($TemplateDm_);` — Reset the template datum and free up internal memory resources. The template datum will need to be trained again after a reset is done.
- `TemplateDm::SetHotPt ($TemplateDm_, $double_x, $double_y);` — Set the hotspot for the template datum to x,y.
- `void TemplateDm::SetBoundRect ($TemplateDm,$int_l, $int_t, $int_r, $int_b);` — Set the bounding rectangle of the template.
- `void TemplateDm::SetBuffer ($TemplateDm, int pBuf);` — Set the internal template buffer to pBuf.
- `void TemplateDm::SetMatchTempl ($TemplateDm,$CorrMatchTempl_templ);` — Set the match template in the specified template datum.
- `void TemplateDm::SetSearchTempl ($TemplateDm,$CorrSearchTempl_templ);` — Set the search template in the specified template datum.

MaskDm — Mask Datum

Also see “Masking Functions” on page 6-13 and “Mask Generating Functions” on page 6-14 for additional perl masking capabilities. For example code, see “Usage Examples” on page 6-112.

- `$int_ret = MaskDm_CountONPixels ($MaskDm);` — Count the pixels that are ON in the specified mask datum and return that count.
- `$int_ret = MaskDm_GetBuffer ($MaskDm);` — Return a pointer to the buffer embedded in the mask datum.
- `$int_ret = MaskDm_GetONPixels ($MaskDm);` — Return the number of pixels ON in the specified mask datum. The ON pixels are not recounted in this method as they are in `CountOnPixel(...)`
- `$int_ret = MaskDm_GetSw1BitMask ($MaskDm);` — Return a pointer to the 1-bit software mask in the specified mask datum.
- `$int_ret = MaskDm_GetSw8BitMask ($MaskDm);` — Return a pointer to the 8-bit software mask in the specified mask datum.
- `$int_ret = MaskDm_Height ($MaskDm);` — Return the height for the specified mask datum.
- `$int_ret = MaskDm_RowBytes ($MaskDm);` — Return the rowbytes for the specified mask datum.
- `$int_ret = MaskDm_SetSizeAndAlloc8Bit ($MaskDm, $int_w, $int_h);` — Set the size and allocate buffers for the specified mask datum.
- `$int_ret = MaskDm_Width ($MaskDm);` — Return the width for the specified mask datum.
- `$int_ret = MaskDm_GetROI ($MaskDm);` — Return a pointer to the ROI contained in the specified mask datum.
- `unsigned char MaskDm_PixVal ($MaskDm, int x, int y);` — Return the value of the pixel in the mask datum at offset x, y from the upper left corner.
- `void MaskDm_Notify1BitChanged ($MaskDm);` — Called when the 1-bit version of the mask changes. This allows internal processing to be done as required.

- `void MaskDm_Notify8BitChanged ($MaskDm);` — Called when the 8-bit version of the mask changes; for example, after a `SetSizeAndAlloc8Bit()` call. This allows internal processing to be done as required.

IOListDm — IO List Datum

An IOList datum sets IO points. The IOList datum supports all types of IO: physical, virtual, strobes, analog, etc. The `write_digital_io` and `read_digital_io` custom steps use the IOList datum to perform IO. The methods available to the perl script to perform IO are detailed below.

- `$int_ret = GetIOIndex ($IOListDm)` — Return the index of the IOList datum. Index values start at "1" -- i.e., index zero does not exist.
- `$int_ret = GetIOType ($IOListDm)` — Return the type of the IOList datum. Type values are:

PHYSICAL	1
VIRTUAL	2
SENSOR	3
STROBE	4
ANALOG OUTPUT	5
SLAVE SENSOR	6
TTL INPUT	7
TTL OUTPUT	8
RS422 INPUT	9
RS422 OUTPUT	10

- `unsigned_int_ret = IOListDm::ReadMyIOPoint ($IOListDm);` — Read the IO point that is specified by the IOList datum and return the value.
- `void IOListDm::WriteMyIOPoint ($IOListDm, $int_value);` — Write `$int_value` to the I/O point that is specified by the IOList datum.
- `void IOListDm::IOListDm_Delete ($IOListDm);` — Delete the specified IOList datum. Note that the IOList datum in the Perl scripts for `write_digital_io` and `read_digital_io` will be automatically deleted by the framework and do not need to be deleted by the user.
- `void IOListDm::PrepareIO ($IOListDm);` — Prepare the IOList object to perform IO. This only needs to be done once and can be done in the `PreRun` subroutine of the Perl script.

Vision Library

The vision library completes a vision task at the lowest levels. This section is broken up into parts. First the packages for common objects such as `roi`, `buffer`, `point`, `fpoint`, etc., are described. These objects are used by virtually all vision agents.

Next, the basic vision agents are described. An agent performs some vision process on an input image, creating a modified output image, image statistics or both. Most agents are created through the use of a factory function. Although agents can be created directly, factories will create the appropriate agent type, software or vision monster, automatically. Calls to create agents return a pointer to the agent created, or zero to indicate failure. Packages describing specific factory function calls are listed with each agent.

Many agents also produce results information. Packages describing specific results for each agent are also listed with that agent.

Common Objects

Buffer

- `$int_ret = Buffer::AltBank ($Buffer_);` — Return the alternate monster bank number from the bank where this buffer is stored.
- `$int_ret = Buffer::Bank ($Buffer_);` — Return the monster bank number where this buffer is stored.
- `$int_ret = Buffer::BitsPerPixel ($Buffer_);` — Return the number of bits per pixel in this buffer.
- `$int_ret = Buffer::BufferCreate ($Buffer_likethis, $int_width, $int_height, $int_allocationType);` — Create a buffer like `$Buffer_likethis` with size `$int_width` by `$int_height`. This will create either a software buffer or a vision monster (ASIC) buffer depending on the buffer type of `$Buffer_likethis`. `$int_allocationType` is ignored for software buffers. For vision monster buffers, `$int_allocationType = 0` means the buffer is allocated in permanent memory. `$int_allocationType = 1` means the buffer is allocated in temporary memory.

- `$int_ret = Buffer::Buffer2 ($Buffer_likethis, $int_width, $int_height);` — Create a buffer like `$Buffer_likethis` with size `$int_width` by `$int_height`, and return a pointer to it.
- `$int_ret = Buffer::Buffer3 ($int_width, $int_height, $int_rowbytes, $PixelFormat_pixeldepth);` — Create a software buffer with the size and depth set as indicated by input parameters, and return a pointer to it.

Note: Agents using this buffer will not use the ASIC for processing.

- `$int_ret = Buffer::GetBitsPerPixel ($Buffer_);` — Get the number of bits per pixel in `$Buffer_`
- `$int_ret = Buffer::GetMonster ($Buffer_);` — Get a pointer to the vision monster object used by this buffer.
- `$int_ret = Buffer::Height ($Buffer_);` — Return the height, in pixels, of this buffer.
- `$int_ret = Buffer::IsSwBuf ($Buffer_);` — Return true if the buffer type is a software buffer.
- `$int_ret = Buffer::IsVgaBuf ($Buffer_);` — Return true if the buffer type is a vga buffer.
- `$int_ret = Buffer::IsVmBuf ($Buffer_);` — Return true if the buffer type is a vision monster buffer.
- `$int_ret = Buffer::RowBytes ($Buffer_);` — Return the rowbytes (width rounded up to next multiple of 4 pixels) for this buffer.
- `$int_ret = Buffer::Size ($Buffer_);` — Return the size of the buffer in pixels.
- `$int_ret = Buffer::Width ($Buffer_);` — Return the width, in pixels, of the buffer.
- `$unsigned char_ret = Buffer::PixVal ($Buffer_, $int_x, $int_y);` — Return the value of the pixel at location `($int_x, $int_y)` in the buffer.
- `Buffer::Buffer_Delete ($Buffer_);` — Delete buffer.

- `Buffer::CopyFrom ($Buffer_dst, $Buffer_src);` — Copy `$Buffer_src` to `$Buffer_dst`.
- `Buffer::CopyFrom2 ($Buffer_dst, $Buffer_src, $int_l, $int_t, $int_r, $int_b);` — Copy the rectangular area specified from `$Buffer_src` to `$Buffer_dst`.
- `Buffer::CopyFrom3 ($Buffer_dst, $Buffer_src, $int_l, $int_t, $int_r, $int_b, $int_destx, $int_desty);` — Copy the rectangular area specified from `$Buffer_src` to the position `($int_destx, $int_desty)` in `$Buffer_dst`.
- `Buffer::ReadTiff ($Buffer_, "filename");` — Read the image from the file “filename.tif” into `$Buffer_`.
- `Buffer::RenderDC ($Buffer)` — Render the graphics associated with this buffer.
- `Buffer::WriteTiff ($Buffer_, "filename");` — Write the image in the buffer to the file “filename.tif”.

Fpoint — Floating-Point Point Object

- `$int_fpoint = Fpoint::Fpoint ();` — Create a floating point object and return a pointer to it.
- `Fpoint::Fpoint_Delete ($int_fpoint);` — Delete the specified floating point object.
- `Fpoint::SetXY ($Fpoint_, $float_xin, $float_yin);` — Set x and y coordinates of floating point object.

Frect — Floating-Point Rectangle Object

- `$int_frect = Frect::Frect ();` — Create a float rectangle object and return a pointer to it.
- `$float_ret = Frect::Height ($Frect_);` — Return height of float rectangle.
- `$float_ret = Frect::Width ($Frect_);` — Return width of float rectangle.
- `Frect::Frect_Delete ($int_frect);` — Delete the specified float rectangle object.

- `Frect::Inflate ($Frect_, $float_x, $float_y);` — Expand the rectangle by `$float_x` in x and `$float_y` in y.

Line — Line Object

- `$double_ret = Line::GetLineA ($Line_);` — Get the A coefficient of the line, where $Ax+By+C=0$
- `$double_ret = Line::GetLineB ($Line_);` — Get the B coefficient of the line, where $Ax+By+C=0$
- `$double_ret = Line::GetLineC ($Line_);` — Get the C coefficient of the line, where $Ax+By+C=0$
- `$double_ret = Line::IncludedAngle ($Line_, $Line_line2);` — Return the included angle, in radians, between this line (`$Line_`) and the line `$Line_line2`.
- `$double_ret = Line::IncludedAngleAbsAcuteDegrees ($Line_, $Line_line2);` — Return the absolute value of the acute included angle, in degrees, between this line (`$Line_`) and the line `$Line_line2`.
- `$int_ret = Line::LinePtDist ($Line_, $point<double>* _pt, $DistanceDm_distancedm);` — Measure the distance from this line (`$Line_`) to the input point (pt). Store the computed distance in `$DistanceDm_distancedm`. Return 0 on success, and nonzero if an error occurs.
- `$int_ret = Line::NormalPt ($Line_, $point<double>* _inpoint, $point<double>* _outpoint);` — Compute a point on the input line which, together with the input point, defines a line normal to the input line. The input point cannot be on the input line. The computed point is stored in `outpoint`. Return 0 for success, nonzero if a failure occurs.
- `$int_ret = Line::Intersect ($Line_, $Line_line2, $point<double>* _retpoint);` — Compute the point of intersection between this line (`$Line_`) and `$Line_line2`. The point of intersection is stored in `retpoint`. Returns 0 on success, and nonzero if an error occurs.
- `$int_ret = Line::Line ();` — Create a line object and return a pointer to it.
- `Line::Line_Delete ($Line_);` — Delete the line.

- `Line::SetLine ($Line_, $double_A, $double_B, $double_C);` — Setup the line with the coefficients specified, where $Ax+By+C=0$
- `Line::TwoPtLine ($Line_, $point<double>_pt1, $point<double>_pt2);` — Compute the line formed by pt1 and pt2 and store it in this line (`$Line_`).

Monster (Vision Monster) — Hardware Accelerator Object

- `$int_ret = Monster::Size ($Monster_);` — Returns the size of each bank, in bytes, in the monster. The monster has 2 banks, 0 and 1.
- `Monster::PermScratch ($Monster_, $BankType_aBank);` — Make all permanent monster memory in aBank available. Permanent memory holds buffers/data that are allocated once and used over many inspections. `$BankType_aBank = 0` or `1`.
- `Monster::TempScratch ($Monster_, $BankType_aBank);` — Make all temporary monster memory in aBank available. Temporary memory holds buffers/data that are allocated for each inspection run. `$BankType_aBank = 0` or `1`.

Point — Point Object

A point can be based on double values or integer values.

- `$int_ret = Point::Point ($double_x, $double_y);` — Construct a point<double> with the input x and y values and return a pointer to it.
- `$double_ret Point = X ($point<double>);` — Return the x coordinate of this point<double>.
- `$double_ret Point = Y ($point<double>);` — Return the y coordinate of this point<double>.
- `Point::Point_Delete ($point<double>_);` — Delete a point<double>.
- `$int_ret = Point::PointInt ($int_x, $int_y);` — Construct an integer based point with the input x and y values and return a pointer to it.
- `$int_ret = Point::intX ($point<int>);` — Return the x coordinate of this integer based point.
- `$int_ret = Point::intY ($point<int>);` — Return the y coordinate of this integer based point.

- `Point::PointInt_Delete ($point<int>_);` — Delete an integer based point.

Quad — Quadrilateral Object

- `$int_ret = Quad::Quad ();` — Create a quadrilateral object and return a pointer to it.
- `$int_ret = Quad::Quad2 ($double_x0, $double_y0, $double_x1, $double_y1, $double_x2, $double_y2, $double_x3, $double_y3);` — Create a quadrilateral object with the given corner points and return a pointer to it.
- `Quad::Quad_Delete ($Quad_);` — Delete the quadrilateral object.

Rect — Rectangle Object

- `$int_ret = Rect::Rect ();` — Create a rectangle object and return a pointer to it.
- `Rect::Rect_Delete ($Rect_);` — Delete a rectangle object.
- `$int_ret = Rect::Height ($Rect_);` — Return the height of the rectangle object.
- `$int_ret = Rect::Width ($Rect_);` — Return the width of the rectangle object.
- `Rect::Inflate ($Rect_, $int_x, $int_y);` — Increase the width of the rectangle by x and the height of the rectangle by y.
- `Rect::Shift ($Rect_, $int_x, $int_y);` — Shift the location of the rectangle right by x and down by y.

Rectshape

Although there is not a Rectshape package, there are some methods in the `perlutil` package that deal with Rectshapes (see “Rectshape Support” on page 6-7).

RRect — Rotated Rectangle Object

`$int = perlutil::CreateRotatedRectFromInputSearchArea()` creates a RRect object from the input ROI information. A pointer to the rotated rect is returned.

- `$double_ret = RRect::Height ($RRect_);` — Return the height.
- `$double_ret = RRect::Width ($RRect_);` — Return the width.
- `$double_ret = RRect::Angle ($RRect_);` — Return the angle of rotation in radians.
- `$double_ret = RRect::CenterX ($RRect_);` — Return the x coordinate of the center point.
- `$double_ret = RRect::CenterY ($RRect_);` — Return the y coordinate of the center point.
- `$int_ret = RRect::RRect ($double_centerX, $double_centerY, $double_angle, $double_width, $double_height);` — Create a rotated rectangle object and return a pointer to it.
- `$void = RRect::Configure ($double_centerX, $double_centerY, $double_angle, $double_width, $double_height);` — Set all parameters of the rotated rectangle.
- `void RRect::Inflate ($RRect_, $double_w, $double_h);` — Increase the width of the rectangle by `$double_w` and the height of the rectangle by `$double_h`.
- `void RRect::Rotate ($RRect_, $double_angle);` — Rotate the rectangle by `$double_angle` which is specified in radians.
- `void RRect::RRect_Delete ($RRect_);` — Delete a rotated rectangle object.
- `void RRect::SetAngle ($RRect_, $double_angle);` — Set the angle of rotation to `$double_angle` which is specified in radians.
- `void RRect::SetCenterPointX ($RRect_, $double_x);` — Set the x coordinate of the center point to `$double_x`.
- `void RRect::SetCenterPointY ($RRect_, $double_y);` — Set the y coordinate of the center point to `$double_y`.

- `void RRect::SetHeight ($RRect_, $double_height);` — Set the height of this rotated rectangle to `$double_height`.
- `void RRect::SetWidth ($RRect_, $double_width);` — Set the width of this rotated rectangle to `$double_width`.

ROI — Region-of-Interest

- `$int_ret = ROI::ROI ();` — Create an ROI object and return a pointer to it.
- `ROI::ROI_Delete ($ROI_);` — Delete the ROI object.
- `$int_ret = ROI::Height ($ROI_);` — Return the height of the ROI.
- `$int_ret = ROI::Width ($ROI_);` — Return the width of the ROI.
- `ROI::SetExtent ($ROI_, $int_width, $int_height);` — Set the width and height of the ROI.
- `ROI::SetMaskbits ($ROI_, $Buffer_newbits);` — Set the maskbits of the ROI with the data in the newbits buffer.

Basic Agents

Arithmetic Agent

Usage Example — Create the arithmetic agent using the `arithfact` package. Do a binarization of a 50x50 area located from (20,30) to (70,80) in the input buffer. Configure the agent to do binarization with min and max thresholds of 50 and 200 respectively. Run the arithmetic agent.

```
$pInbuf = perlutil::get_input_buf;           #get pointer to input
buffer
$POutbuf = Buffer::BufferCreate($pInbuf, 50, 50, 1);#create 50x50 output
buffer
#create arithmetic agent to binarize
$PArithAgent = ArithFact::ArithFact($pInbuf, 20, 30, 70, 80, $pOutbuf, 3,
1.0, 0, 0) ;
ArithAgent::SetThreshold ($pArithAgent, 50, 200);#configure thresholds
ArithAgent::Go($pArithAgent);               #run arithmetic agent
```

Arithfact

Create

- `$int_ret = ArithFact::ArithFact ($InBuffer_, $int_l, $int_t, $int_r, $int_b, $OutBuffer_, $enum_ArithOp, $float_gain, $int_offset, $InBuffer2_);`
— Create an arithmetic agent and return a pointer to it.

The arithmetic opcodes are:

```
enum ArithOp {
  _ARITH_NOT = 0,
  _ARITH_NEGATE=1,
  _ARITH_COPY=2,
  _ARITH_BINARIZE=3,
  _ARITH_MASKSHIFT=4,
  _ARITH_ABS=5,
  _ARITH_PASSRANGE=6,
  _ARITH_ADD=7,
  _ARITH_SUB=8,
  _ARITH_DIFF=9,
  _ARITH_AND=10,
  _ARITH_OR=11,
  _ARITH_XOR=12,
  _ARITH_MIN=13,
  _ARITH_MAX=14};
```

Arithagent

Delete

- `ArithAgent::ArithAgent_Delete ($ArithAgent_);` — Delete the arithmetic agent.

Pre Go / Go

- `$int_ret = ArithAgent::Go ($ArithAgent_);` — Run the arithmetic agent.

Results

- `$int_ret = ArithAgent::GetCountOutput ($ArithAgent_);` — Get the number of nonzero output pixels generated.

- `$int_ret = ArithAgent::GetSumOutput $ArithAgent_;` — Get the sum of the output pixels generated.

Configuration/Information

- `ArithAgent::SetCornerB ($ArithAgent_, $int_xB, $int_yB);` — Set the top-left point of the B (second) operand to (`$int_xB`, `$int_yB`). Used with 2-operand functions like subtraction, for example.
- `ArithAgent::SetNorm ($ArithAgent_, $double_gain, $int_offset);` — Set the gain and offset for the arithmetic agent.
- `ArithAgent::SetSrcB ($ArithAgent_, $Buffer_);` — Set the source buffer pointer for the B (second) operand.
- `ArithAgent::SetThreshold ($ArithAgent_, $int_loThresh, $int_hiThresh);` — Set the lower and upper pixel threshold values for the arithmetic agent.

Common To All Agents

- `$int_ret = ArithAgent::SrcBuf ($ArithAgent_);` — Get the source buffer pointer for the arithmetic agent.
- `$int_ret = ArithAgent::SrcPtX ($ArithAgent_);` — Get x coordinate of source point (top-left point within source buffer) for agent.
- `$int_ret = ArithAgent::SrcPtY ($ArithAgent_);` — Get y coordinate of source point (top-left point within source buffer) for agent.
- `$int_ret = ArithAgent::SrcROI ($ArithAgent_);` — Get pointer to current source ROI for agent.
- `ArithAgent::SetSrcBuf ($ArithAgent_, $Buffer_srcBuf);` — Set the source buffer pointer for the arithmetic agent.
- `ArithAgent::SetSrcPt ($ArithAgent_, $Point_sPt);` — Set top-left point within source buffer where agent processing will begin.
- `ArithAgent::SetSrcROI ($ArithAgent_, $ROI_sRoi);` — Set the region within the source buffer that the agent will process.

AutoFocus

AutoFocusFact — AutoFocus Agent Factory

Create

- `$int_ret = AutoFocusFact::AutoFocusFact($Buffer_src,$int_l, $int_t, $int_r, $int_b);` — Create an AutoFocus agent and return a pointer to it. The agent will process the designated rectangular area with the source buffer.
- `$int_ret = AutoFocusFact::AutoFocusFact2($Buffer_src);` — Create an AutoFocus agent and return a pointer to it. The agent will process the entire source buffer.

AutoFocusAgent — AutoFocusAgent

Delete

- `AutoFocusAgent::AutoFocusAgent_Delete($AutoFocusAgent);` — Delete the AutoFocusAgent.

Pre Go / Go

- `$int_ret = AutoFocusAgent::Go($AutoFocusAgent);` — Run the AutoFocus agent.

Configuration/Information

- `$double_ret = AutoFocusAgent::GetFocusValue ($AutoFocusAgent);` — Return the focus value computed by the agent from the last Go().
- `$double_ret = AutoFocusAgent::GetHistoValue($AutoFocusAgent,$int_index);` — Return the number of pixels in the histogram bin specified by index. Index is valid from 0 to n-1, where n is the number of histogram bins in the agent.
- `$int_ret = AutoFocusAgent::GetHorizontalSparseness ($AutoFocusAgent);` — Return the horizontal sparseness. This is the spacing used for measuring horizontal pixel intensity differences.

- `$double_ret = AutoFocusAgent::GetMean ($AutoFocusAgent);` — Return the mean value of histogrammed values.
- `$int_ret = AutoFocusAgent::GetNumHistoBins ($AutoFocusAgent);` — Return the number of histogram bins being used by the agent.
- `$double_ret = AutoFocusAgent::GetPixelCount ($AutoFocusAgent);` — Return the number of pixels used to compute the focus value.
- `$double_ret = AutoFocusAgent::GetPixelSum ($AutoFocusAgent);` — Return the sum of pixels used to compute the focus value.
- `$double_ret = AutoFocusAgent::GetStdDev ($AutoFocusAgent);` — Return the standard deviation of the histogrammed values.
- `$double_ret = AutoFocusAgent::GetSumPixelDiff ($AutoFocusAgent);` — Return the sum of the pixel differences.
- `$int_ret = AutoFocusAgent::GetVerticalSparseness ($AutoFocusAgent);` — Return the vertical sparseness. This is the spacing used for measuring vertical pixel intensity differences.
- `$int_ret = AutoFocusAgent::SetNumHistoBins ($AutoFocusAgent,$int_n);` — Set the number of histogram bins. Valid values are: 16,32,64,128,256. The default for the agent is 256.
- `void AutoFocusAgent::SetHorizontalSparseness ($AutoFocusAgent,$int_x);` — Set the horizontal sparseness. This is the horizontal spacing between pixels used when computing horizontal pixel intensity differences.
- `void AutoFocusAgent::SetVerticalSparseness ($AutoFocusAgent,$int_y);` — Set the vertical sparseness. This is the vertical spacing between pixels used when computing vertical pixel intensity differences.

Common To All Agents

- `$int_ret = AutoFocusAgent::SrcBuf ($AutoFocusAgent_);` — Get pointer to current source buffer for agent.
- `$int_ret = AutoFocusAgent::SrcPtX ($AutoFocusAgent_);` — Get x coordinate of source point (top-left point within source buffer) for agent.

- `$int_ret = AutoFocusAgent::SrcPtY ($AutoFocusAgent_);` — Get y coordinate of source point (top-left point within source buffer) for agent.
- `$int_ret = AutoFocusAgent::SrcROI ($AutoFocusAgent_);` — Get pointer to current source ROI for agent.
- `AutoFocusAgent::SetSrcBuf ($AutoFocusAgent_, $Buffer_srcBuf);` — Set source buffer for agent for next `Go()`.
- `AutoFocusAgent::SetSrcPt ($AutoFocusAgent_, $Point_sPt);` — Set top-left point within source buffer where agent processing will begin.
- `AutoFocusAgent::SetSrcROI ($AutoFocusAgent_, $ROI_sRoi);` — Set region within source buffer that agent will process.

Morphology

Binary Morphology

Usage Example — Create the binary morphology agent using the `binmorphfact` package. Do two iterations of full dilatation on a 50x50 area located from (20,30) to (70,80) in the input buffer. Get the number of pixels On in the result.

```
$pInbuf = perlutil::get_input_buf;           #get pointer to input
buffer
$pOutbuf = Buffer::BufferCreate($pInbuf, 50, 50, 1);#create 50x50 output
buffer
#create binary morph operator for full dilate
$pBMO = BinMorphOperator::BinMorphOperator (6);
#create binary morphology agent (full dilate, 2 iterations)
$pBinmorphAgent = BinMorphFact::BinMorphFact($pInbuf, $pOutbuf, 20,
30, 70, 80, 0, $pBMO, 2);
BinMorphAgent::Go($pBinMorphAgent);         #run binmorph agent
$count = BinMorphAgent::GetCount($pBinMorphAgent);#get count of ON
pixels
```

BinMorphFact — Binary Morphology Factory

Create

- `$int_ret = BinMorphFact::BinMorphFact ($Buffer_, $Buffer_, $int_l, $int_t, $int_r, $int_b, $BMBORDER_theBorder, $PBMO_pBMO, $int_iter);` — Create a binary morphology agent and return a pointer to it.

BinMorphAgent — Binary Morphology Agent

Delete

- `BinMorphAgent::BinMorphAgent_Delete ($BinMorphAgent_);` — Delete the binary morphology agent.

Pre Go / Go

- `$int_ret = BinMorphAgent::Go ($BinMorphAgent_);` — Run the Agent.

Results

- `$int_ret = BinMorphAgent::GetChange ($BinMorphAgent_);` — Returns zero if no pixels were changed (from source to destination) when the agent last ran; return one if any pixels were changed.
- `$int_ret = BinMorphAgent::GetCount ($BinMorphAgent_);` — Return the number of pixels that are On (nonzero) in the destination/result.
- `$int_ret = BinMorphAgent::GetError ($BinMorphAgent_);` — Return the error from the last run of the agent.

Configuration/Information

- `BinMorphAgent::SetIterations ($BinMorphAgent_, $int_iter);` — Set iteration count, the number of times the morphological operation is performed on the source buffer.

Common To All Agents

- `$int_ret = BinMorphAgent::SrcBuf ($BinMorphAgent_);` — Get pointer to current source buffer for agent.

- `$int_ret = BinMorphAgent::SrcPtX ($BinMorphAgent_);` — Get x coordinate of source point (top-left point within source buffer) for agent.
- `$int_ret = BinMorphAgent::SrcPtY ($BinMorphAgent_);` — Get y coordinate of source point (top-left point within source buffer) for agent.
- `$int_ret = BinMorphAgent::SrcROI ($BinMorphAgent_);` — Get pointer to current source ROI for agent.
- `BinMorphAgent::SetSrcBuf ($BinMorphAgent_, $Buffer_srcBuf);` — Set source buffer for agent for next `Go()`.
- `BinMorphAgent::SetSrcPt ($BinMorphAgent_, $Point_sPt);` — Set top-left point within source buffer where agent processing will begin.
- `BinMorphAgent::SetSrcROI ($BinMorphAgent_, $ROI_sRoi);` — Set region within source buffer that agent will process.

BinMorphOperator — Binary Morphology Operator

The binary morph operator is used by the binary morph agent and determines the morphological operation that is performed.

Create

- `BinMorphOperator::BinMorphOperator ($BinMorphOperator_, $enum_BinMorphOpCode);` — Create a binary morph operator object to do the specified function.

```
enum BinMorphOpCode {
    SEMIERODE = 0,      semi-strong erosion
    FULLERODE = 1,      full erosion
    SEMIDILATE = 2,      semi-strong dilation
    NOP = 3,             no operation, output = input
    NOISEERODE = 4,      erode single isolated pixels
    SHAPEERODE = 5,      erosion along shape or contour
    FULLDILATE = 6,      full dilation
    SHAPEDILATE = 7 }    dilation along shape or contour
```

- `BinMorphOperator::BinMorphOperator ($BinMorphOperator_, $pBMLut);` — Create a binary morph operator object using data

pointed to by \$pBMLut. Data consists of sixteen 32bit unsigned integers specifying a look-up table.

Delete

- `BinMorphOperator::BinMorphOperator_Delete ($BinMorphOperator_);` — Delete a binary morph operator object.

Configuration/Information

- `$unsigned_ret = BinMorphOperator::GetLutWord ($BinMorphOperator_, $int_which);` — Return the word in the look-up table at index \$int_which (index is zero based).

Grayscale Morphology

Graymorphfact — Grayscale Morphology Factory

Create

- `$int_ret = GrayMorphFact::GrayMorphFact ($Buffer_, $Buffer_, $int_l, $int_t, $int_r, $int_b, $enum_GRAYMORPHOP, $enum_GMPOLARITY, $int_iterations, $GrayMorphElem_);` — Create a gray morphology agent and return a pointer to it.

Graymorphagent — Grayscale Morphology Agent

Delete

- `GrayMorphAgent::GrayMorphAgent_Delete ($GrayMorphAgent_);` — Delete a gray morphology agent.

Pre Go / Go / Results

- `$int_ret = GrayMorphAgent::GetError ($GrayMorphAgent_);` — Return the error from the last run of the agent.
- `$int_ret = GrayMorphAgent::Go ($GrayMorphAgent_);` — Run the agent.

Configuration

- `$GMPOLARITY_ret = GrayMorphAgent::GetPolarity ($GrayMorphAgent_);` — Return the polarity for this agent. 0 = Light, 1 = Dark. Light Polarity means that erode will shrink light pixels and dilate will grow light pixels. Dark Polarity means that erode will shrink dark pixels and dilate will grow dark pixels.
- `$GRAYMORPHOP_ret = GrayMorphAgent::GetMorphFcn ($GrayMorphAgent_);` — Return the function that the agent is configured to run.

The gray morph functions are:

enumGrayMorphOp{	
erode = 0	erosion/min function
dilate = 1	dilation/max function
open = 2	erosion followed by dilation
close = 3	dilation followed by erosion
gradient = 4	max - min function
tophat = 5	original image - open image
well = 6}	close image - original image

- `$sint_ret = GrayMorphAgent::GetIterations ($GrayMorphAgent_);` — Return the number of iterations this agent is configured to run.

Common To All Agents

- `$sint_ret = GrayMorphAgent::SrcBuf ($GrayMorphAgent_);` — Get pointer to current source buffer for agent.
- `$sint_ret = GrayMorphAgent::SrcPtX ($GrayMorphAgent_);` — Get x coordinate of source point (top-left point within source buffer) for agent.
- `$sint_ret = GrayMorphAgent::SrcPtY ($GrayMorphAgent_);` — Get y coordinate of source point (top-left point within source buffer) for agent.
- `$sint_ret = GrayMorphAgent::SrcROI ($GrayMorphAgent_);` — Get pointer to current source ROI for agent.
- `GrayMorphAgent::SetSrcBuf ($GrayMorphAgent_, $Buffer_srcBuf);` — Set source buffer for agent for next Go().

- `GrayMorphAgent::SetSrcPt ($GrayMorphAgent_, $Point_sPt);` — Set top-left point within source buffer where agent processing will begin.
- `GrayMorphAgent::SetSrcROI ($GrayMorphAgent_, $ROI_sRoi);` — Set region within source buffer that agent will process.

Graymorphelem — Grayscale Morphology Structuring Element

The structuring element determines which pixels are processed as the structuring element is moved over the ROI. If a bit is Off in the structuring element, any pixel beneath it is ignored.

Create/Delete

- `$int_ret = GrayMorphElem::GrayMorphElem ();` — Create a gray morph structuring element and return a pointer to it.
- `$int_ret = GrayMorphElemCustom ($int_xhot,$ int_yhot,$ int_xsize,$int_ysize, $unsigned char_elemTbl);` — Create a custom gray morph structuring element of size (xsize,ysize) with hot spot at (xhot,yhot) and containing the elements in elemTbl. The elements in elemTbl are delimited by either spaces or commas. For example, create this custom 5 x 5 element:

TABLE 6–5.

0	1	1	1	1
1	0	1	1	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

`$elemTbl = "0 1 1 1 1 1 0 1 1 1 1 0 1 1 1 1 0";` #
delimit with commas or spaces.

`$pElem = GrayMorphElem::GrayMorphElemCustom (2, 2, 5, 5,
$elemTbl);`

- `GrayMorphElem::Delete ($GrayMorphElem_);` — Delete a gray morph structuring element.

Configuration/Information

- `$bool_ret = GrayMorphElem::ElementsUpdated ($GrayMorphElem_);`
— This function needs to be called when a structuring is changed to update its internal representation.
- `$bool_ret = GrayMorphElem::SetElementHotSpot ($GrayMorphElem_, $int_x, $int_y);` — Set hot-spot of the structuring element. Returns false if `$int_x` or `$int_y` are out of range, true otherwise.
- `$bool_ret = GrayMorphElem::SetElementWeight ($GrayMorphElem_, $int_x, $int_y, $unsigned_char);` — Set the weight at the specified `$int_x`, `$int_y` offsets within the structuring element. Return false if `$int_x` or `$int_y` are out of range, true otherwise.
- `$int_ret = GrayMorphElem::GetElementHeight ($GrayMorphElem_);`
— Return the height of the structuring element.
- `$int_ret = GrayMorphElem::GetElementHotX ($GrayMorphElem_);` — Return the x coordinate of the hot-spot of the structuring element.
- `$int_ret = GrayMorphElem::GetElementHotY ($GrayMorphElem_);` — Return the y coordinate of the hot-spot of the structuring element.
- `$int_ret = GrayMorphElem::GetElementWidth ($GrayMorphElem_);`
— Return the width of the structuring element.
- `$unsigned char_ret = GrayMorphElem::GetElementWeight ($GrayMorphElem_, $int_x, $int_y);` — Return the weight at the specified `$int_x`, `$int_y` offsets within the structuring element.

Blob

Usage Example — Create the blob agent using the blobfact package. Process a 50x50 area located from (20,30) to (70,80) in the input buffer. Extract the (x,y) location of the center of each blob found.

```
$pInbuf = perlutil::get_input_buf;          #get pointer to input buffer
#create a blob result for blob agent
$BlobResult = BlobResult::BlobResult(6);
#create blob agent
$BlobAgent = BlobFact::BlobFact($pInbuf, 20, 30, 70, 80, 0,
$BlobResult, 0, 100) ;
```

```
BlobAgent::Go($pBlobAgent);          #run blob agent
# extract the (x,y) location of each blob found
$NBlobs = BlobResult::NBlobs($BlobResult_);
for ($i=0; $i<$NBlobs; $i++) {
    $pBlob = BlobResult::GetBlob($BlobResult_, $i);
    $x = Blob::CentroidX($pBlob);
    $y = Blob::CentroidY($pBlob);
    printf ("\nBlob Location = (%6.2f, %6.2f)", $x, $y);
}
```

Blob — Methods For Blob Statistics and Configuration

Create/Delete

- `$int_ret = Blob::Blob ($BlobResult*__res);` — Create a blob object. The input `res` is an optional parameter and is a pointer to a `BlobResult` object.
- `Blob::Blob_Delete ($Blob_);` — Delete a blob object.

Statistics/Configuration

- `$double_ret = Blob::Angle ($Blob_);` — Returns the major axis angle in radians from horizontal (+/- 90 degrees).
- `$double_ret = Blob::Area ($Blob_);` — Returns the area in pixels.
- `$double_ret = Blob::AreaRatio ($Blob_);` — Returns the ratio: area-of-blob/area-of-search-region (ROI).
- `$double_ret = Blob::AvgRad ($Blob_);` — Returns the average distance from center to perimeter points.
- `$double_ret = Blob::AxRatio ($Blob_);` — Returns the ratio of minor axis length to major axis length.
- `$double_ret = Blob::BoxArea ($Blob_);` — Returns the area in pixels of bounding (extent) box.
- `$double_ret = Blob::BoxAreaRatio ($Blob_);` — Returns the ratio of total area to box area.

- `$double_ret = Blob::CGDist ($Blob_);` — Returns the distance between center point and origin (usually top left corner of buffer)
- `$double_ret = Blob::HoleArea ($Blob_);` — Returns the area of all the holes in a blob.
- `$double_ret = Blob::HoleRatio ($Blob_);` — Returns the ratio of hole area to total blob area (≤ 1.0).
- `$double_ret = Blob::LenDiff ($Blob_);` — Returns the asymmetry along major axis. This is the distance between center of rotated enclosing box and center of blob area.
- `$double_ret = Blob::Length ($Blob_);` — Returns the length of blob along major axis.
- `$double_ret = Blob::LenRatio ($Blob_);` — Returns the ratio of width to length.
- `$double_ret = Blob::Major ($Blob_);` — Returns the equivalent ellipse major axis length.
- `$double_ret = Blob::MaxMinRad ($Blob_);` — Returns the angle range in radians (+/- 180 degrees).
- `$double_ret = Blob::Minor ($Blob_);` — Returns the equivalent ellipse minor axis length.
- `$double_ret = Blob::Perimeter ($Blob_);` — Returns the blob perimeter length including change in direction correction.
- `$double_ret = Blob::Peround ($Blob_);` — Returns the roundness (≤ 1.0), a perfect circle has a peround of 1.0.
- `$double_ret = Blob::PPDA ($Blob_);` — Returns the perimeter invariant, calculated as perimeter squared over total area.
- `$double_ret = Blob::RadRatio ($Blob_);` — Returns the ratio of minimum radius to maximum radius.
- `$double_ret = Blob::RMax ($Blob_);` — Returns the maximum distance from center to perimeter of blob.
- `$double_ret = Blob::RMaxAng ($Blob_);` — Returns the angle of radius of maximum distance perimeter point.

- `$double_ret = Blob::RMin ($Blob_);` — Returns the minimum distance from center to perimeter of blob.
- `$double_ret = Blob::RMinAng ($Blob_);` — Returns the angle of radius of minimum distance perimeter point.
- `$double_ret = Blob::SigX ($Blob_);` — Returns the 1st moment of area in x.
- `$double_ret = Blob::SigXX ($Blob_);` — Returns the 2nd moment of area in x.
- `$double_ret = Blob::SigXY ($Blob_);` — Returns the 2nd moment of area in xy.
- `$double_ret = Blob::SigY ($Blob_);` — Returns the 1st moment of area in y.
- `$double_ret = Blob::SigYY ($Blob_);` — Returns the 2nd moment of area in y.
- `$double_ret = Blob::TotArea ($Blob_);` — Returns the total blob area including holes.
- `$double_ret = Blob::WidDiff ($Blob_);` — Returns the asymmetry along minor axis, distance between center of rotated enclosing box and center of area.
- `$double_ret = Blob::Width ($Blob_);` — Returns the length of blob along minor axis.
- `$double_ret = Blob::XDiff ($Blob_);` — Returns the horizontal extent, i.e., right - left.
- `$double_ret = Blob::XPerim ($Blob_);` — Returns the x perimeter length in pixels.
- `$double_ret = Blob::YDiff ($Blob_);` — Returns the vertical extent, i.e., bottom - top.
- `$double_ret = Blob::YPerim ($Blob_);` — Returns the y perimeter length in pixels.
- `$float_ret = Blob::BottomPointX ($Blob_);` — Returns the x coordinate of the bottom extent of the blob.

- `$float_ret = Blob::BottomPointY ($Blob_);` — Returns the y coordinate of the bottom extent of the blob.
- `$float_ret = Blob::CentroidX ($Blob_);` — Returns the subpixel x center of blob.
- `$float_ret = Blob::CentroidY ($Blob_);` — Returns the subpixel y center of blob.
- `$float_ret = Blob::LeftPointX ($Blob_);` — Returns the x coordinate of the left extent of the blob.
- `$float_ret = Blob::LeftPointY ($Blob_);` — Returns the y coordinate of the left extent of the blob.
- `$float_ret = Blob::RightPointX ($Blob_);` — Returns the x coordinate of the right extent of the blob.
- `$float_ret = Blob::RightPointY ($Blob_);` — Returns the y coordinate of the right extent of the blob.
- `$float_ret = Blob::TopPointX ($Blob_);` — Returns the x coordinate of the top extent of the blob.
- `$float_ret = Blob::TopPointY ($Blob_);` — Returns the y coordinate of the top extent of the blob.
- `$int_ret = Blob::CentroidX2 ($Blob_);` — Returns the whole pixel x center of blob.
- `$int_ret = Blob::CentroidY2 ($Blob_);` — Returns the whole pixel y center of blob.
- `$int_ret = Blob::Color ($Blob_);` — Returns the color of the blob. Where `PARTCOLOR = 0`, `HOLECOLOR = -1`.
- `$int_ret = Blob::NHoles ($Blob_);` — Returns the number of holes.
- `$int_ret = Blob::Parent ($Blob_);` — Returns a pointer to the parent blob.
- `$int_ret = Blob::ParentOf ($Blob_);` — Returns the parent blob's handle.

- `$long_ret = Blob::GrayAverage ($Blob_);` — Returns the gray average of blob, for software blob processing only.
- `$long_ret = Blob::GrayTotal ($Blob_);` — Returns the gray sum of blob, for software blob processing only.
- `$long_ret = Blob::NCells ($Blob_);` — Returns the area in pixels, excluding holes.
- `$long_ret = Blob::TotCells ($Blob_);` — Returns the area in pixels, including holes.
- `Blob::Axes ($Blob_);` — Calculates the angle of rotation, major and minor axes from the second moments.
- `Blob::BoundingBox ($Blob_, $int_l, $int_t, $int_r, $int_b);` — Returns the bounding box of the blob. The parameters l, t, r, b are pointers to integers, and the bounding box coordinates are stored there.
- `Blob::BoundingBox2 ($Blob_, $Rect&_r);` — Returns the bounding box of the blob into the specified rectangle.
- `Blob_HighlightBlob ($Blob_, $pAcuityMetaDC, $int_cornerx, $int_cornerx);` — Highlight the given blob.
- `$int_ret = Blob::HighlightBlobCustom ($blob, $int_r, $int_g, $int_b, $mystep, $mybufdm);` — Highlight a blob by drawing a crosshair at the blob's centroid location and drawing the blob boundary in the RGB color indicated. Pointers to the step and input buffer datum for the owner CustomVisionTool are needed for drawing graphics. Get these as follows:

```
$mystep = perlutil::get_myStepPointer();
$mybufdm = perlutil::get_input_bufferdm ();
```

A nonzero return value from HighlightBlobCustom indicates an error.

Blobfact — Blob Agent Factory

Create

- `$int_ret = BlobFact::BlobFact ($Buffer_src, $int_l, $int_t, $int_r, $int_b, $BlobResult_res, $int_lowThr, $int_highThr);` — Create a blob agent and return a pointer to it.

- `$int_ret = BlobFact::BlobFactMask ($Buffer_src, $int_l, $int_t, $roi_ROI, $BlobResult_res, $int_lowThr, $int_highThr);` — Create a maskable blob agent and return a pointer to it.

Blobagent — Blob Agent

Delete

- `BlobAgent::BlobAgent_Delete ($BlobAgent_);` — Delete the blob agent.

Pre Go / Go / Results

- `$int_ret = BlobAgent::GetResult ($BlobAgent_);` — Returns the result from the run of the agent. A pointer to a `BlobResult` is returned. Refer to the `blobResult` package for more information.
- `$int_ret = BlobAgent::Go ($BlobAgent_);` — Run the blob agent.

Configuration

- `BlobAgent::SetFlags ($BlobAgent_, $int_cFlags);` — Set various filtering flags as specified by `cFlags`. The filtering flags are:


```
CNOFILTER=0,    // reset builtin filtering flags
CMINPARTS=1,    // only apply minblob to parts
CMAXPARTS=2,    // only apply maxblob to parts
CTOUCHROI=4,   // ignore blobs that touch the rect edges of the roi
CDOGRAYAVG=8,  // compute gray average for each run length
                (swBlob only)
CDISCARDCHILDREN=16, // discard children when discarding blobs
CFILTAREA=32    // filter blobs based on Area or TotalArea(default)
```
- `BlobAgent::SetMaxBlob ($BlobAgent_, $int_maxblob);` — Sets the maximum blob size (total area).
- `BlobAgent::SetMaxSegments ($BlobAgent_, $int_maxsegments);` — Sets the maximum number of segments per line when doing blob analysis.
- `BlobAgent::SetMaxStride ($BlobAgent_, $int_maxstride);` — Sets the maximum ROI width.

- `BlobAgent::SetMinBlob ($BlobAgent_, $int_minblob);` — Sets the minimum blob size.
- `BlobAgent::SetProcessSwitches ($BlobAgent_, $int_switchflags);` — Set various processing flags as specified by switchflags. The processing flags are:

```
DO1MOM=1,      // compute first moments
DO2MOM=2,      // compute second moments
DOPERIM=4,      // accumulate boundary description
DOHOLEMOM=8,   // calculate hole moments
DORADII=16,    // calculate perimeter features
DOLENWID=32    // calculate symmetry feature
```
- `BlobAgent::SetResult ($BlobAgent_, $BlobResult);` — The blob agent will output results into the specified blob result object.
- `BlobAgent::SetThresholds ($BlobAgent_, $int_lowThr, $int_highThr);` — Set the low and high thresholds.

Common To All Agents

- `$int_ret = BlobAgent::SrcBuf ($BlobAgent_);` — Get pointer to current source buffer for agent.
- `$int_ret = BlobAgent::SrcPtX ($BlobAgent_);` — Get x coordinate of source point (top-left point within source buffer) for agent.
- `$int_ret = BlobAgent::SrcPtY ($BlobAgent_);` — Get y coordinate of source point (top-left point within source buffer) for agent.
- `$int_ret = BlobAgent::SrcROI ($BlobAgent_);` — Get pointer to current source ROI for agent.
- `BlobAgent::SetSrcBuf ($BlobAgent_, $Buffer_srcBuf);` — Set source buffer for agent for next `Go()`.
- `BlobAgent::SetSrcPt ($BlobAgent_, $Point_sPt);` — Set top-left point within source buffer where agent processing will begin.
- `BlobAgent::SetSrcROI ($BlobAgent_, $ROI_sRoi);` — Set region within source buffer that agent will process.

Blobresult — Blob Agent Results (Blobtree)

Create/Delete

- `int BlobResult_BlobResult (double LDiffMin, double WDiffMin);` — Create a blob result and return a pointer to it. Need to create a `BlobResult` for a blob agent to use.
- `void BlobResult_CleanUp ($BlobResult_);` — Free some static allocations for the blob result.

Statistics/Drawing

- `double BlobResult_LDiffMin ($BlobResult_);` — Return the minimum length differential (asymmetry along major axis).
- `double BlobResult_WDiffMin ($BlobResult_);` — Return the minimum width differential (asymmetry along minor axis).
- `BlobResult_HighlightBlobTree ($BlobResult_, $pAcuityMetaDC, $int_cornerx, $int_cornerx);` — Highlight the given blob result (blobtree).
- `$int_ret = BlobResult::HighlightTreeCustom ($blobtree, $int_r, $int_g, $int_b, $mystep, $mybufdm);` — Highlight a blobresult (blobtree). For each blob in the blob tree draw a crosshair at the blob's centroid location and draw the blob boundary in the RGB color indicated. Pointers to the step and input buffer datum for the owner `CustomVisionTool` are needed for drawing graphics. Get these as follows:

```
$mystep = perlutil::get_myStepPointer();
$mybufdm =perlutil::get_input_bufferdm ();
```

A nonzero return value from `HighlightTreeCustom` indicates an error.

- `int BlobResult_GetBlob ($BlobResult_, int index);` — Returns a pointer to the indexth blob in the blob result.
- `int BlobResult_NBlobs ($BlobResult_);` — Return the number of blob objects in the blob result.
- `int BlobResult_NParts ($BlobResult_);` — Returns the number of parts in the blob result.

- `long BlobResult_ROIArea ($BlobResult_);` — Returns the number of pixels in the ROI that blob agent processed.
- `void BlobResult_SetLWDiffMin ($BlobResult_, double LDiffMin, double WDiffMin);` — Set the minimum length and width differentials (asymmetry along major axis and minor axis).

Convolution

Convfact

Create

- `$int_ret = ConvFact::ConvFact ($Buffer_, $Buffer_, $int_l, $int_t, $int_r, $int_b, $CorrTemplate_, $_int, $int_tSigned, $int_xStep, $int_yStep);` — Create a convolution agent and return a pointer to it.

Convagent — Convolution Agent

Delete

- `ConvAgent::ConvAgent_Delete ($ConvAgent_);` — Delete the convolution agent.

Pre Go / Go

- `$int_ret = ConvAgent::Go ($ConvAgent_);` — Run the agent.

Information

- `$int_ret = ConvAgent::ClipNegValues ($ConvAgent_);` — Return the state of the ClipNegValues flag. 0=false, 1=true.
- `$int_ret = ConvAgent::NormValue ($ConvAgent_);` — Return the state of the NormValue flag. 0=false, 1=true.
- `$int_ret = ConvAgent::SrcSigned ($ConvAgent_);` — Return the state of the SrcSigned flag. 0=false, 1=true.
- `$int_ret = ConvAgent::TemplateSigned ($ConvAgent_);` — Return the state of the TemplateSigned flag. 0=false, 1=true.

- `$int_ret = ConvAgent::UseAbsSum ($ConvAgent_);` — Return the state of the `UseAbsSum` flag. 0=false, 1=true.
- `$int_ret = ConvAgent::UseNormValue ($ConvAgent_);` — Return the state of the `UseNormValue` flag. 0=false, 1=true.
- `$int_ret = ConvAgent::UseSum ($ConvAgent_);` — Return the state of the `UseSum` flag. 0=false, 1=true.
- `$int_ret = ConvAgent::UseSumSqr ($ConvAgent_);` — Return the state of the `UseSumSqr` flag. 0=false, 1=true.

Configuration

- `ConvAgent::ClipNegValues ($ConvAgent_, $int yesNo);` — Set the state of the `ClipNegValues` flag. 0=do not clip negative results, 1=clip negative results to zero.
- `ConvAgent::NormValue ($ConvAgent_, $int_val);` — Set the normalization value.
- `ConvAgent::SetSrcSigned ($ConvAgent_, $int yesNo);` — Set the state of the `SrcSigned` flag. 0=src (image) data is unsigned, 1=src (image) data is signed.
- `ConvAgent::SetTemplateSigned ($ConvAgent_, $int yesNo);` — Set the state of the `TemplateSigned` flag. 0=template data is unsigned, 1=template data is signed.
- `ConvAgent::SetUseAbsSum ($ConvAgent_, $int yesNo);` — Set the state of the `UseAbsSum` flag. 0=disabled, 1=enabled: use the absolute value of the template's sum of pixels (`abs(ST)`) to compute right-shift of the sum of the `[template*image]` result. One of `UseSumSqr`, `UseSum`, `UseAbsSum` or `UseNormValue` must be selected/enabled.
- `ConvAgent::SetUseNormValue ($ConvAgent_, $int yesNo);` — Set the state of the `UseNormValue` flag. 0=disabled, 1=enabled: use the normalization value to compute right-shift of the sum of the `template*image` result. One of `UseSumSqr`, `UseSum`, `UseAbsSum` or `UseNormValue` must be selected/enabled.
- `ConvAgent::SetUseSum ($ConvAgent_, $int yesNo);` — Set the state of the `UseSum` flag. 0=disabled, 1=enabled: use the template's sum

of pixels (ST) to compute right-shift of the sum of the [template*image] result. One of UseSumSqr, UseSum, UseAbsSum or UseNormValue must be selected/enabled.

- `ConvAgent::SetUseSumSqr ($ConvAgent_, $int yesNo);` — Set the state of the UseSumSqr flag. 0=disabled, 1=enabled: use the squareroot of the template's sum of pixels squared value ($\sqrt{ST2}$) to compute right-shift of the sum of the [template*image] result. One of UseSumSqr, UseSum, UseAbsSum or UseNormValue must be selected/enabled.

Common To All Agents

- `$int_ret = ConvAgent::SrcBuf ($ConvAgent_);` — Get pointer to current source buffer for agent.
- `$int_ret = ConvAgent::SrcPtX ($ConvAgent_);` — Get x coordinate of source point (top-left point within source buffer) for agent.
- `$int_ret = ConvAgent::SrcPtY ($ConvAgent_);` — Get y coordinate of source point (top-left point within source buffer) for agent.
- `$int_ret = ConvAgent::SrcROI ($ConvAgent_);` — Get pointer to current source ROI for agent.
- `ConvAgent::SetSrcBuf ($ConvAgent_, $Buffer_srcBuf);` — Set source buffer for agent for next Go().
- `ConvAgent::SetSrcPt ($ConvAgent_, $Point_sPt);` — Set top-left point within source buffer where agent processing will begin.
- `ConvAgent::SetSrcROI ($ConvAgent_, $ROI_sRoi);` — Set region within source buffer that agent will process.

Correlation (Match, Search)

Corrmatchfact — Correlation Match Agent Factory

- `$int_ret = CorrMatchFact::CorrMatchFact ($Buffer_, $int_l, $int_t, $int_r, $int_b, $CorrTemplate_, $CorrResult_, $int_sSigned, $int_tSigned, $int_xStep, $int_yStep);` — Create a correlation match agent and return a pointer to it.

Corrmatchagent — Correlation Match Agent

Delete

- `CorrMatchAgent::CorrMatchAgent_Delete ($CorrMatchAgent_);` — Delete a correlation match agent.

Pre Go / Go / Results

- `$int_ret = CorrMatchAgent::Go ($CorrMatchAgent_);` — Run correlation match agent.
- `$int_ret = CorrMatchAgent::PreGo ($CorrMatchAgent_);` — Prepare correlation match agent to run.

See `corresult` package for results information.

Configuration

- `$int_ret = CorrMatchAgent::MaskEnable ($CorrMatchAgent_);` — Enable masking for correlation match agent.
- `CorrMatchAgent::SetCorrResults ($CorrMatchAgent_, $CorrResult_aResultObject);` — Set the correlation result object in this agent to `$CorrResult_aResultObject`.
- `CorrMatchAgent::SetMaskEnable ($CorrMatchAgent_, $int_enable);` — Enable/disable use of mask during correlation. Enable = 1, Disable = 0.
- `CorrMatchAgent::SetPositionOnly ($CorrMatchAgent_, $int_flag);` — If `$int_flag` is true, then configure the correlation agent to compute the match position only (for better speed); if `$int_flag` is false, then both score and match position are computed.
- `CorrMatchAgent::SetTemplate ($CorrMatchAgent_, $CorrTemplate_aTemplate);` — Set the correlation template for this agent to `$CorrTemplate_aTemplate`.

Common To All Agents

- `CorrMatchAgent::SetSrcBuf ($CorrMatchAgent_, $Buffer_pBuffer);` — Set source buffer for agent for next `Go()`.

Corrpoint — Correlation Point Object

Create/Delete

- `$int_ret = CorrPoint::CorrPoint ();` — Create a correlation point object and return a pointer to it.
- `CorrPoint::CorrPoint _Delete ($CorrPoint_);` — Delete a correlation point object.

Results

- `$double_ret = CorrPoint::Score ($CorrPoint_);` — Return the correlation score of correlation point.
- `$int_ret = CorrPoint::X ($CorrPoint_);` — Return the x coordinate of correlation point.
- `$int_ret = CorrPoint::Y ($CorrPoint_);` — Return the y coordinate of correlation point.

Corresult — Correlation Result Object

Create/Delete

- `$int_ret = Corresult::Corresult ($int_size);` — Create a correlation result object with `$int_size` results (results themselves are not setup).
- `Corresult::Delete ($Corresult_);` — Delete a correlation result object.

Results

- `$double_ret = Corresult::SumImage ($Corresult_, $int_n);` — Returns the sum of image pixels for correlation point number `$int_n`.
- `$double_ret = Corresult::SumImageTemp ($Corresult_, $int_n);` — Returns the sum of image*template pixels for correlation point number `$int_n`.
- `$double_ret = Corresult::SumSqrImage ($Corresult_, $int_n);` — Returns the sum of image*image pixels for correlation point number `$int_n`.

- `$double_ret = CorrResult::Value ($CorrResult_, $int_n);` — Returns the score of correlation point number `$int_n`.
- `$int_ret = CorrResult::Size ($CorrResult_);` — Returns the size of the correlation result object.
- `$int_ret = CorrResult::X ($CorrResult_, $int_n);` — Returns the x coordinate of correlation point number `$int_n`.
- `$int_ret = CorrResult::Y ($CorrResult_, $int_n);` — Returns the y coordinate of correlation point number `$int_n`.

Corrsrchfact — Correlation Search Factory

Create

- `$int_ret = CorrSrchFact::CorrSrchFact ($Buffer_aBuf, $int_l, $int_t, $int_r, $int_b, $CorrSearchTempl_aTempl);` — Create a correlation search agent and return a pointer to it.

Corrsrchagent — Correlation Search Agent

Delete

- `CorrSrchAgent::CorrSrchAgent_Delete ($CorrSrchAgent_);` — Delete a correlation search agent.

Pre Go / Go / Results

- `$float_ret = CorrSrchAgent::GetScore ($vector<CorrMax>_, $int_index);` — Returns the score of correlation result number `$int_index`.
- `$float_ret = CorrSrchAgent::GetX ($vector<CorrMax>_, $int_index);` — Returns the x coordinate of correlation result number `$int_index`.
- `$float_ret = CorrSrchAgent::GetY ($vector<CorrMax>_, $int_index);` — Returns the y coordinate of correlation result number `$int_index`.
- `$int_ret = CorrSrchAgent::GetResults2 ($CorrSrchAgent_);` — Returns a pointer to a vector of correlation results `$vector<CorrMax>`. (GetResults2 and DeleteResults2 must be paired together to avoid memory leaks).

- `$int_ret = CorrSrchAgent::Go ($CorrSrchAgent_);` — Run correlation search agent.
- `CorrSrchAgent::DeleteResults2($CorrSrchAgent_, $vector<CorrMax>_results);` — Deletes a vector of correlation results. (`GetResults2` and `DeleteResults2` must be paired together to avoid memory leaks).
- `CorrSrchAgent::PreGo ($CorrSrchAgent_);` — Prepare correlation search agent to run.

Configuration/Information

- `$int_ret = CorrSrchAgent::TimesDecimated ($CorrSrchAgent_);` — Return the number of decimations agent is configured to do when run.
- `CorrSrchAgent::SetAcceptThresh ($CorrSrchAgent_, $float_thresh, $int_nmatch);` — Set minimum correlation score for allowable match; set maximum number of matches returned.
- `CorrSrchAgent::SetRobustness ($CorrSrchAgent_, $float_level);` — Set robustness level. Robust level is between 0.0 and 1.0 with greater values indicating greater levels of robustness.

Common To All Agents

- `$int_ret = CorrSrchAgent::SrcBuf ($CorrSrchAgent_);` — Get pointer to current source buffer for agent.
- `$int_ret = CorrSrchAgent::SrcPtX ($CorrSrchAgent_);` — Get x coordinate of source point (top-left point within source buffer) for agent.
- `$int_ret = CorrSrchAgent::SrcPtY ($CorrSrchAgent_);` — Get y coordinate of source point (top-left point within source buffer) for agent.
- `$int_ret = CorrSrchAgent::SrcROI ($CorrSrchAgent_);` — Get pointer to current source ROI for agent.
- `CorrSrchAgent::SetSrcBuf ($CorrSrchAgent_, $Buffer_);` — Set source buffer for agent for next `Go()`.

- `CorrSrchAgent::SetSrcPt ($CorrSrchAgent_, $Point_sPt);` — Set top-left point within source buffer where agent processing will begin.
- `CorrSrchAgent::SetSrcROI ($CorrSrchAgent_, $ROI_sRoi);` — Set region within source buffer that agent will process.

Corrtempl — Correlation Template (For use w/corrsrchagent)

- `$bool_ret = CorrTempl::Signed ($CorrTempl_);` — Return true if the template is a signed image, return false if the template is an unsigned image.
- `$corrTempl = CorrTempl_CorrSearchTemplMask (Buffer* Tbuf,$roi_ROI, int left, int top, float hotx, float hoty)` — Create a new maskable `CorrSearchTempl` object and return a pointer to it. The template buffer for the `CorrSearchTempl` object is the area within the input buffer (`Tbuf`) with upper left boundary of (`left,top`) and extents that equal the width and height of the mask contained in the input ROI. The template is trained as part of the creation of the object.
- `$corrTempl = CorrTempl_CorrTempl ();` — Create a new `CorrTempl` object and return a pointer to it. The object created will have no template data.
- `$double_ret = CorrTempl::Num ($CorrTempl_);` — Return the number of pixels On in the template.
- `$double_ret = CorrTempl::Sum ($CorrTempl_);` — Return the sum of the pixel values in the template.
- `$double_ret = CorrTempl::SumSqr ($CorrTempl_);` — Return the sum of the squares of the pixel values in the template.
- `$int_ret = CorrTempl::GetErrOfConstr ($CorrTempl_);` — Return the error code when the correlation template was constructed. If this value is nonzero, then do no try to run correlation with this template.
- `$int_ret = CorrTempl::NmbDecimated ($CorrTempl_);` — Return the number of times the template is decimated.

Corrtemplate — Correlation Template (For use w/corrMatchAgent)

Create/Delete

- `$int_ret = CorrTemplate::CorrTemplate ($Buffer_buffer, $int_mask);` — Create a correlation template and return a pointer to it. `$Buffer` specifies a pointer to a buffer holding the image for the template. Mask indicates whether or not pixels with value zero are included in statistics computations: 0=exclude zero pixels from statistics computations, 1=include zero pixels in statistics computations.
- `$int_ret = CorrTemplate::CorrTemplate2 ($Buffer_, $int_left, $int_top, $int_right, $int_bottom, $int_mask);` — Create a correlation template and return a pointer to it. `$Buffer` specifies a pointer to a buffer holding the image for the template. Left, top, right, and bottom specify the region within `$Buffer`. Mask indicates whether or not pixels with value zero are included in statistics computations: 0=exclude zero pixels from statistics computations, 1=include zero pixels in statistics computations.
- `CorrTemplate::CorrTemplate_Delete ($CorrTemplate_);` — Delete a correlation template.

Results

- `$double_ret = CorrTemplate::Sum ($CorrTemplate_);` — Return the sum of the pixel values in the template.
- `$double_ret = CorrTemplate::SumSqr ($CorrTemplate_);` — Return the sum of the squares of the pixel values in the template.
- `$int_ret = CorrTemplate::Num ($CorrTemplate_);` — Return the number of pixels on in the template.

Configuration/Information

- `$int_ret = CorrTemplate::GetBuffer ($CorrTemplate_);` — Returns a pointer to the buffer that holds the image for the correlation template.
- `CorrTemplate::SetHotSpot ($CorrTemplate_, $int_x, $int_y);` — Set the hotspot for the correlation template to (x,y).

- `CorrTemplate::SetMaskEnable ($CorrTemplate_, $int_mask);` — Set the mask enable for the correlation template. Mask indicates whether or not pixels with value zero are included in statistics computations: 0=exclude zero pixels from statistics computations, 1=include zero pixels in statistics computations.

Fcorrpoint — Correlation Point Object

Results

- `float FCorrPoint_GetAngle ($FCorrPoint_);` — Return the angle of the correlation point.
- `float FCorrPoint_GetCorrValue ($FCorrPoint_);` — Return the correlation value (score) of the correlation point.
- `float FCorrPoint_GetLocX ($FCorrPoint_);` — Return the X coordinate of the correlation point location.
- `float FCorrPoint_GetLocY ($FCorrPoint_);` — Return the Y coordinate of the correlation point location.
- `float FCorrPoint_GetScale ($FCorrPoint_);` — Return the scale of the correlation point.

Decimation

Currently, there are no perl access functions for decimate agent. The `SWDecimateAgent` specifically needs an input buffer that is not on the vision monster, i.e., a software buffer.

A `Buffer::CopyFrom (vmBuffer, SWBuffer)` can be done to copy a buffer from the vision monster into a software buffer for `swDecimateAgent` to use.

Decimatefact — Decimate Agent Factory

Create

- `$int_ret = DecimateFact::DecimateFact ($Buffer_, $_Buffer, $_int, $int_t, $int_r, $int_b, $PixelStep_step);` — Create a decimation agent and return a pointer to it.

Swdecimateagent — Software Decimate Agent

Delete

- `swDecimateAgent::swDecimateAgent_Delete ($DecimateAgent_);` — Delete a software decimation agent.

Configuration/Information

- `$sint_ret = swDecimateAgent::DstBuf ($DecimateAgent_);` — Return a pointer to the destination buffer for software decimation agent.

Pre Go / Go

- `$sint_ret = swDecimateAgent::Go ($DecimateAgent_);` — Run the software decimation agent.

DMR

DMRFact — DMR Agent Factory

Create

- `$sint_ret = DMRFact::DMRFact($Buffer_src,$sint_l, $sint_t, $sint_r, $sint_b);` — Create an DMR agent and return a pointer to it. The agent will process the designated rectangular area with the source buffer.
- `$sint_ret = DMRFact::DMRFact2($Buffer_src);` — Create an DMR agent and return a pointer to it. The agent will process the entire source buffer.

DMRAgent — DMRAgent

Delete

- `DMRAgent::DMRAgent_Delete($DMRAgent);` — Delete the DMRAgent.

Pre Go / Go

Configuration/Information

Common To All Agents

- `DMRAgent::SetSrcBuf ($DMRAgent_, $Buffer_srcBuf);` — Set source buffer for agent for next `Go()`.
- `DMRAgent::SetSrcPt ($DMRAgent_, $Point_sPt);` — Set top-left point within source buffer where agent processing will begin.
- `DMRAgent::SetSrcROI ($DMRAgent_, $ROI_sRoi);` — Set region within source buffer that agent will process.

EdgeFast

EdgeFast Agent

- `$EdgeFastAgent = EdgeFast::EdgeFast ($Buffer, $int_numPrims);` — Create an edge fast agent and return a pointer to it. `$int_numPrims` is the number of edge primitives that the agent contains. Each edge primitive processes a certain area within the input buffer for edges and is uniquely configured.
- `EdgeFast::SetPrim ($EdgefastAgent, $int_n,$int_l,$int_t,$int_r,$int_b,$int_direction,$int_color,$int_option,$int_threshold)` — Configure the `n`th edge primitive.

Where:

`$int_n`: Index of this edge primitive. `SetPrim` must be called for each edge primitive that the `EdgeFast` agent is going to process.

`$int_l`, `$int_t`, `$int_r`, `$int_b`: Defines the rectangular region within the input buffer to process for edges.

`$int_direction`: (look for edges going):

- 0 = from left -> right
- 1 = from top -> bottom
- 2 = from right -> left
- 3 = from bottom to top

\$int_color: (edge polarity):

0 = light -> dark edge
1 = dark -> light edge
2 = any edge

\$int_option: (which edge to detect):

0 = first edge
1 = best edge
2 = last edge

\$int_threshold: minimum edge threshold

Delete

- `EdgeFast::EdgeFast_Delete ($EdgeFastAgent _);` — Delete an EdgeFast agent.

Pre Go / Go

- `$int_ret = EdgeFast::Go ($EdgeFastAgent_);` — Run the EdgeFast agent.

Results

- `$EdgePrim_ret = EdgeFast::GetPrim ($EdgeFastAgent, $int_n);` — Return a pointer to the net edge primitive for the specified EdgeFast agent. The actual points where the edge is found and the line equation for the edge are contained in the EdgePrim object.

EdgePrim

Configure

- `$void SetGradSpace ($EdgePrim, $int_gradSpace)` — Set the gradient space. The difference between pixels gradient space apart determines the strength of the gradient.
- `$void SetGradThr ($EdgePrim, $int_gradThreshold)` — Set the gradient threshold to \$int_gradThreshold.

Results

- `EdgePrim::ComputeRobustClassic ($EdgePrim);` — Computes a line to fit the edge points in this `EdgePrim` using a robust method.
- `EdgePrim::ComputeLine ($EdgePrim);` — Computes a line to fit the edge points in this `EdgePrim` using a least squares method.
- `EdgePrim::ComputeMoreRobust ($EdgePrim);` — Computes a line to fit the edge points such that all points used fall within a given distance (0.75 pixels) from the resulting line.
- `$float_ret = $EdgePrim::A ($EdgePrim)` — Return the A coefficient of the line equation for the edge in this edge primitive. Line equation: $Ax+By+C=0$.
- `$float_ret = $EdgePrim::B ($EdgePrim)` — Return the B coefficient of the line equation for the edge in this edge primitive. Line equation: $Ax+By+C=0$.
- `$float_ret = $EdgePrim::C ($EdgePrim)` — Return the C coefficient of the line equation for the edge in this edge primitive. Line equation: $Ax+By+C=0$.
- `$float_ret = $EdgePrim::GetPointX ($EdgePrim, $int_n)` — Return the x coordinate of the nth edge point found in this edge primitive.
- `$float_ret = $EdgePrim::GetPointY ($EdgePrim, $int_n)` — Return the y coordinate of the nth edge point found in this edge primitive.
- `$int_ret = $EdgePrim::PointCount ($EdgePrim)` — Return the number of edge points in this edge primitive.

Gradscan

Gradscanfact — Gradscan Agent Factory

Create

- `$int_ret = GradScanFact::GradScanFact ($Buffer_src, $int_l, $int_t, $int_r, $int_b, $SignMode_mode, $GradSign_sign, $GradSelect_select, $int_minG, $GradMode_gmode);` — Create a gradient scan agent and return a pointer to it.

\$SignMode_mode = 0 = unsigned
\$SignMode_mode = 1<<24 = signed
\$GradSign_sign = 0 = Positive (Dark to Light)
\$GradSign_sign = 1<<25 = Negative (Light to Dark)
\$GradSelect_select = 0 = report all grads
\$GradSelect_select = 1<<26 = report max gradient only

Gradscanagent — Gradient Scan Agent

Delete

- `GradScanAgent::GradScanAgent_Delete ($GradScanAgent_);` — Delete a gradient scan agent.

Pre Go / Go

- `$int_ret_pGradPoints=GradScanAgent::AllocGradPoints ($int_number);` — Allocate a vector of \$int_number gradient points, and fill with default gradient points. Must be paired with `FreeGradPoints` function to prevent memory leaks.
- `$int_ret = GradScanAgent::Go ($GradScanAgent_);` — Run a gradient scan agent.
- `GradScanAgent::FreeGradPoints ($int pGradPoints);` — Deallocate a vector of gradient points. Must be paired with `AllocGradPoints` function to prevent memory leaks.
- `GradScanAgent::PreGo ($GradScanAgent_);` — Prepare a gradient scan agent to run.

Results

- `$float_ret = GradScanAgent::GetQualifiedPointX ($int pGradPoints, $int_index);` — Get the x coordinate of qualified gradient point number \$int_index. (pGradPoints comes from a previous call to `AllocGradPoints` function).
- `$float_ret = GradScanAgent::GetQualifiedPointY ($int pGradPoints, $int_index);` — Get the y coordinate of qualified gradient point number \$int_index. (pGradPoints comes from a previous call to `AllocGradPoints` function).

- `$int_ret = GradScanAgent::GetQualifiedPointGrad ($int pGradPoints, $int_index);` — Get the gradient value of qualified gradient point number `$int_index`. (`pGradPoints` comes from a previous call to `AllocGradPoints` function).
- `$int_ret = GradScanAgent::QualifiedPoints ($GradScanAgent_, $int pGradPoints);` — Get the qualified gradient points from the agent and put into `pGradPoints`. (`pGradPoints` comes from a previous call to `AllocGradPoints` function). If return value = 0 ==> OK, else the `pGradPoints` is bad or no gradient points were found.

```
$int_ret=GradScanAgent::ComputeMinCenter($vector<GradPoint>* _
pGradPoints1, $vector<GradPoint>* _pGradPoints2,
$int_minGradThresh, $int_direction, $int_left, $int_top, $int_right,
$int_bottom, $int_graphics, BufferDm*_pbufferdm, $int_debug,
$float_minDiff, $float_maxDiff, $PtListDm*_pdmPtList);
```

Computes the minimum center point when taking the center points of two sets of point data. If each set of points represents a line or edge, then this is useful to get the lowest midpoint between the lines/edges.

Where:

`pGradPoints1`, `pGradPoints2` — come from previous calls to `AllocGradPoints` function.

`$int_direction` — 0=compute center based on y coordinates
 1=compute center based on x coordinates

`$int_left`, `$int_top`, `$int_right`, `$int_bottom`: bounding area of step/tool used when grad points computed

`$int_graphics`: 0=no graphics, 1=show all computed points

`$BufferDm*_pbufferdm`: pointer to buffer datum that has buffer for graphics output

`$int_debug`: 0=no debug messages, 1=debug messages printed

`$float_minDiff`: minimum allowable difference between points (in x or y, depending on direction); if difference is less, then point is not used in computation

`$float_maxDiff`: maximum allowable difference between points (in x or y, depending on direction); if difference is more, then point is not used in computation

`$PtListDm*_pdmPtList`: pointer to a point list datum to hold all points used in computation. If 0, no points are output.

```
$int_ret=GradScanAgent::ComputeMaxCenter($vector<GradPoint>*
_pGradPoints1, $vector<GradPoint>*_pGradPoints2,
$int_minGradThresh, $int_direction, $int_left, $int_top, $int_right,
$int_bottom, $int_graphics, $BufferDm*_pbufferdm, $int_debug,
$float_minDiff, $float_maxDiff, $PtListDm*_pdmPtList);
```

Computes the maximum center point when taking the center points of two sets of point data. If each set of points represents a line or edge, then this is useful to get the highest midpoint between the lines/edges.

Where:

pGradPoints1, pGradPoints2: come from previous calls to AllocGradPoints function. \$int_direction: 0=compute center based on y coordinates, 1=compute center based on x coordinates
\$int_left, \$int_top, \$int_right, \$int_bottom: bounding area of step/tool used when grad points computed
\$int_graphics: 0=no graphics, 1=show all computed points
\$BufferDm*_pbufferdm: pointer to buffer datum that has buffer for graphics output
\$int_debug: 0=no debug messages, 1=debug messages printed
\$float_minDiff: minimum allowable difference between points (in x or y, depending on direction); if difference is less, then point is not used in computation
\$float_maxDiff: maximum allowable difference between points (in x or y, depending on direction); if difference is more, then point is not used in computation
\$PtListDm*_pdmPtList: pointer to a point list datum to hold all points used in computation. If 0, no points are output.

```
$int_ret=GradScanAgent::ComputeAvgCenter($vector<GradPoint>*
_pGradPoints1, $vector<GradPoint>*_pGradPoints2,
$int_minGradThresh, $int_direction, $int_left, $int_top, $int_right,
$int_bottom, $int_graphics, $BufferDm*_pbufferdm, $int_debug,
$float_minDiff, $float_maxDiff);
```

Computes the average center point when taking the center points of two sets of point data. If each set of points represents a line or edge, then this is useful to get the average midpoint between the lines/edges.

Where:

pGradPoints1, pGradPoints2: come from previous calls to AllocGradPoints function.

\$int_direction: 0=compute center based on y coordinates, 1=compute center based on x coordinates

\$int_left, \$int_top, \$int_right, \$int_bottom: bounding area of step/tool used when grad points computed

\$int_graphics: 0=no graphics, 1=show all computed points

\$BufferDm*_pbufferdm: pointer to buffer datum that has buffer for graphics output

\$int_debug: 0=no debug messages, 1=debug messages printed

\$float_minDiff: minimum allowable difference between points (in x or y, depending on direction); if difference is less, then point is not used in computation

\$float_maxDiff: maximum allowable difference between points (in x or y, depending on direction); if difference is more, then point is not used in computation

\$PtListDm*_pdmPtList: pointer to a point list datum to hold all points used in computation. If 0, no points are output.

- \$float_ret = GradScanAgent::GetCenterX (); — Return the center X coordinate computed in one of the methods (ComputeMinCenter, ComputeMaxCenter, ComputeAvgCenter)
- \$float_ret = GradScanAgent::GetCenterY (); — Return the center Y coordinate computed in one of the methods (ComputeMinCenter, ComputeMaxCenter, ComputeAvgCenter)

Configuration/Information

- GradScanAgent::SetMinGradient (\$GradScanAgent_, \$int_minimum); — Set minimum gradient threshold for gradscan agent.

Histogram

Histogramfact — Histogram Agent Factory

Create Agent

- \$int_ret = HistogramFact::HistogramFact (\$Buffer_src, \$HistRes_resolution, \$int_l, \$int_t, \$int_r, \$int_b,

`$vector<int>_results);` — Create a histogram agent and return a pointer to it.

Create/Delete Results

- `$int_ret = HistogramFact::CreateResults ();` — Create result objects used by histogram agent, and return a pointer to it.
- `HistogramFact::DeleteResults ($vector<int>_results);` — Delete result objects used by histogram agent.

Histogramagent — Histogram Agent

Delete (Agent)

- `HistogramAgent::HistogramAgent_Delete ($HistogramAgent_);` — Delete a histogram agent.

Pre Go / Go

- `$int_ret = HistogramAgent::Go ($HistogramAgent_);` — Run the agent.

Results

- `$int_ret = HistogramAgent::Bucket ($HistogramAgent_, $int_i);` — Return the results for histogram bucket `$int_i`.
- `$int_ret = HistogramAgent::GetPixelCount ($HistogramAgent_);` — Return the number of pixels On in the result buffer.
- `$int_ret = HistogramAgent::GetPixelSum ($HistogramAgent_);` — Return the sum of the pixels in the result buffer.
- `$int_ret = HistogramAgent::Overflow ($HistogramAgent_);` — Return the current overflow value.
- `$int_ret = HistogramAgent::Underflow ($HistogramAgent_);` — Return the current underflow value.

Configuration/Information

- `$int_ret = HistogramAgent::NumBuckets ($HistogramAgent_);` — Return the number of buckets setup in the agent.
- `$int_ret = HistogramAgent::Offset ($HistogramAgent_);` — Return the offset value setup in the agent. The offset value is added to each pixel in the processed area before assigning that pixel to a bin-bucket in the histogram result.

Common To All Agents

- `$int_ret = HistogramAgent::SrcBuf ($HistogramAgent_);` — Get pointer to current source buffer for agent.
- `$int_ret = HistogramAgent::SrcPtX ($HistogramAgent_);` — Get x coordinate of source point (top-left point within source buffer) for agent.
- `$int_ret = HistogramAgent::SrcPtY ($HistogramAgent_);` — Get y coordinate of source point (top-left point within source buffer) for agent.
- `$int_ret = HistogramAgent::SrcROI ($HistogramAgent_);` — Get pointer to current source ROI for agent.
- `HistogramAgent::SetSrcBuf ($HistogramAgent_, $Buffer_srcBuf);` — Set source buffer for agent for next `go()`.
- `HistogramAgent::SetSrcPt ($HistogramAgent_, $Point_sPt);` — Set top-left point within source buffer where agent processing will begin.
- `HistogramAgent::SetSrcROI ($HistogramAgent_, $ROI_sRoi);` — Set region within source buffer that agent will process.

Hough Processing

Houghfact — Hough Agent Factory

Create

- `$int_ret = HoughFact::HoughFact ($Buffer_, $int_l, $int_t, $int_r, $int_b,`

`$int_threshold, $int_sparseness,);` — Create a hough (circle find) agent and return a pointer to it.

- `$int_ret = HoughFact::HoughFact2 ($Buffer_, $int_l, $int_t, $int_r, $int_b);` — Create a hough (circle find) agent and return a pointer to it.

HoughAgent — Hough Agent

Create/Delete

- `$int_ret = HoughAgent::HoughAgent ($Buffer_srcbuf, $int_threshold, $int_sparseness, $int_pts_per_kernal, $float_c1_center, $float_c2_center, $int_c1_minpts, $int_c2_minpts, $int_circle_maxpts, $float_c1_radius_search, $float_nominal_radius);` — Create the hough agent and return a pointer to it.
- `HoughAgent::HoughAgent_Delete ($HoughAgent_);` — Delete hough agent.

Pre Go / Go

- `$int_ret = HoughAgent::Go ($HoughAgent_);` — Run the agent.
- `HoughAgent::PostGo ($HoughAgent_);` — Post Go (post run) of agent.
- `HoughAgent::PreGo ($HoughAgent_);` — Prepare agent to run.

Results

- `$double_ret = HoughAgent::GetResultRadius ($HoughAgent_, $int_index);` — Return radius of circle number `$int_index` (index is zero based).
- `$double_ret = HoughAgent::GetResultX ($HoughAgent_, $int_index);` — Return x-coordinate of circle number `$int_index` (index is zero based).
- `$double_ret = HoughAgent::GetResultY ($HoughAgent_, $int_index);` — Return y-coordinate of circle number `$int_index` (index is zero based).
- `$int_ret = HoughAgent::GetNumCircles ($HoughAgent_);` — Return number of circles found by agent.

- `$int_ret = HoughAgent::GetResultVotes ($HoughAgent_, $int_index);` — Return number of votes circle number `$int_index` received (index is zero based).

Configuration/Information

- `$int_ret = HoughAgent::GetGradientMaxX ($HoughAgent_, $int_index);` — Return the maximum X gradient setting of the hough agent.
- `$int_ret = HoughAgent::GetGradientMaxY ($HoughAgent_, $int_index);` — Return the maximum Y gradient setting of the hough agent.
- `$int_ret = HoughAgent::GetGradientMinX ($HoughAgent_, $int_index);` — Return the minimum X gradient setting of the hough agent.
- `$int_ret = HoughAgent::GetGradientMinY ($HoughAgent_, $int_index);` — Return the minimum Y gradient setting of the hough agent.

Common To All Agents

- `$int_ret = HoughAgent::SrcBuf ($HoughAgent_);` — Get pointer to current source buffer for agent.
- `$int_ret = HoughAgent::SrcPtX ($HoughAgent_);` — Get x coordinate of source point (top-left point within source buffer) for agent.
- `$int_ret = HoughAgent::SrcPtY ($HoughAgent_);` — Get y coordinate of source point (top-left point within source buffer) for agent.
- `$int_ret = HoughAgent::SrcROI ($HoughAgent_);` — Get pointer to current source ROI for agent.
- `HoughAgent::SetSrcBuf ($HoughAgent_, $Buffer_srcBuf);` — Set source buffer for agent for next go().
- `HoughAgent::SetSrcPt ($HoughAgent_, $Point_sPt);` — Set top-left point within source buffer where agent processing will begin.
- `HoughAgent::SetSrcROI ($HoughAgent_, $ROI_sRoi);` — Set region within source buffer that agent will process.

Mask Processing

Maskfact — Mask Agent Factory

Create

- `$int_ret = MaskFact::MaskFact ($Buffer_src, $Buffer_dst, $int_x, $int_y, $ROI_, $int_lowThr, $int_highThr);` — Create a masking agent and return a pointer to it.

Maskagent — Mask Agent

Delete

- `MaskAgent::MaskAgent_Delete ($MaskAgent_);` — Delete the mask agent.

Pre Go / Go

- `$int_ret = MaskAgent::Go ($MaskAgent_);` — Run the agent.

Configuration/Information

- `$int_ret = MaskAgent::DestBufSize ($MaskAgent_);` — Return the size of the destination buffer for agent (rounded up to the next multiple of 32 bits/pixels).
- `$int_ret = MaskAgent::DstBuf ($MaskAgent_);` — Return a pointer to the destination buffer for agent.
- `MaskAgent::SetDestBuf ($MaskAgent_, $Buffer_);` — Set the destination buffer for agent to `$Buffer_`.

Common To All Agents

- `$int_ret = MaskAgent::SrcBuf ($MaskAgent_);` — Get pointer to current source buffer for agent.
- `$int_ret = MaskAgent::SrcPtX ($MaskAgent_);` — Get x coordinate of source point (top-left point within source buffer) for agent.
- `$int_ret = MaskAgent::SrcPtY ($MaskAgent_);` — Get y coordinate of source point (top-left point within source buffer) for agent.

- `$int_ret = MaskAgent::SrcROI ($MaskAgent_);` — Get pointer to current source ROI for agent.
- `MaskAgent::SetSrcBuf ($MaskAgent_, $Buffer_srcBuf);` — Set the source buffer pointer for the agent.
- `MaskAgent::SetSrcPt ($MaskAgent_, $Point_sPt);` — Set top-left point within source buffer where agent processing will begin.
- `MaskAgent::SetSrcROI ($MaskAgent_, $ROI_sRoi);` — Set region within source buffer that agent will process.

Projection

ProjectFact

- `$int_ret = ProjectFact_ProjectFact ($int_src,$ int_dst,$ int_l,$int_t,$int_r,$int_b, $int_type);` — Create a projection agent and return a pointer to it. The projection agent will process the rectangular area bound by \$l, \$t, \$r, \$b in the input buffer.
- `$int_ret = ProjectFact_ProjectFact2 (int src, int dst, int type);` — Create a projection agent and return a pointer to it. The projection agent will process the entire input buffer.

Valid values for `$int_type` are:

0 = vertical
1 = horizontal

Projectagent — Projection Agent

Delete

- `ProjectAgent::ProjectAgent_Delete ($ProjectAgent_);` — Delete the projection agent.

Pre Go / Go

- `$int_ret = ProjectAgent::Go ($ProjectAgent_);` — Run the agent.

Configuration/Information

- `$bool_ret = ProjectAgent::AutoShift ($ProjectAgent_);` — Return true if auto shift is enabled, false otherwise.
- `$int_ret = ProjectAgent::Shift ($ProjectAgent_);` — Return the number of times the data is shifted right when projection is done.
- `$int_ret = ProjectAgent::Type ($ProjectAgent_);` — Return the projection type for the agent. 0=vertical, 1=horizontal.
- `ProjectAgent::SetAutoShift ($ProjectAgent_, $bool_autoShift);` — If `autoshift=true`, then right-shift count is calculated automatically by agent. Otherwise, use `SetShiftRight()`.
- `ProjectAgent::SetDstBuf ($ProjectAgent_, $Buffer_);` — Set the destination buffer for agent to `$Buffer_`.
- `ProjectAgent::SetDstRow ($ProjectAgent_, $int_row);` — Set the destination row for agent, this is where the projection will be stored in the destination buffer.
- `ProjectAgent::SetShiftRight ($ProjectAgent_, $int_shift);` — Set the number of times the data is shifted right when projection is done.
- `ProjectAgent::SetType ($ProjectAgent_, $int_type);` — Set the projection type 0=vertical, 1=horizontal.

Common To All Agents

- `$int_ret = ProjectAgent::SrcBuf ($ProjectAgent_);` — Get pointer to current source buffer for agent.
- `$int_ret = ProjectAgent::SrcPtX ($ProjectAgent_);` — Get x coordinate of source point (top-left point within source buffer) for agent.
- `$int_ret = ProjectAgent::SrcPtY ($ProjectAgent_);` — Get y coordinate of source point (top-left point within source buffer) for agent.
- `$int_ret = ProjectAgent::SrcROI ($ProjectAgent_);` — Get pointer to current source ROI for agent.

- `ProjectAgent::SetSrcBuf ($ProjectAgent_, $Buffer_srcBuf);` — Set the source buffer pointer for the agent.
- `ProjectAgent::SetSrcPt ($ProjectAgent_, $Point_sPt);` — Set top-left point within source buffer where agent processing will begin.
- `ProjectAgent::SetSrcROI ($ProjectAgent_, $ROI_sRoi);` — Set region within source buffer that agent will process.

Sobel

Sobelfact — Sobel Agent Factory

Create

- `$int_ret = SobelFact::SobelFact ($Buffer_src, $Buffer_dst, $int_l, $int_t, $int_r, $int_b, $enum_SobelOut, $_enum, $int_QThresh);` — Create a sobel agent and return a pointer to it.
- `$int_ret = SobelFact::SobelFactMask ($Buffer_src, $Buffer_, $int_l, $int_t, $roi_ROI, $enum_SobelOut, $_enum, $int_QThresh);` — Create a maskable sobel agent and return a pointer to it.

The selectable outputs, `$enum_SobelOut` are:

```
SOBEL_DX = 0
SOBEL_DY = 1
SOBEL_MAG = 2
SOBEL_ANGLE = 3
SOBEL_QANGLE = 4
SOBEL_ABSGRADX = 5
SOBEL_ABSGRADY = 6
SOBEL_3M5A = 7
SOBEL_4GX4GY = 8
```

`$int_QThresh` is the threshold for qualified Sobel results. The default threshold is 20.

Sobelagent — Sobel Agent

Delete

- `SobelAgent::SobelAgent_Delete ($SobelAgent_);` — Delete the sobel agent.

Pre Go / Go

- `$int_ret = SobelAgent::Go ($SobelAgent_);` — Run the sobel agent.

Configuration/Information

- `$int_ret = SobelAgent::GetOffset ($SobelAgent_);` — Get the offset the sobel agent is configured with. The offset offsets input values into the histogrammer internally in the agent.

Return the right shift the sobel agent is configured with:

```
SOBEL_DIV1 = 0,  
SOBEL_DIV2 = 1,  
SOBEL_DIV4 = 2,  
SOBEL_DIV8 = 3,
```

- `$int_ret = SobelAgent::GetRightShift ($SobelAgent_);` — The right shift selects the gain on input values for the histogrammer internally in the agent.
- `SobelAgent::NoHistogram ($SobelAgent_);` — Configures the sobel agent to NOT use the histogrammer.
- `SobelAgent::SetMaskOutput ($SobelAgent_, $int_newmask);` — Set the mask for the sobel agent output to newmask.
- `SobelAgent::SetupQualifySobel ($SobelAgent_, $int_qualify, $int_sobelThresh);` — Configure the sobel agent to generate sobel-qualified-sobel statistics if qualify=1. The sobel output is qualified on sobelThresh. All sobel output less than the sobelThresh is ignored.

Common To All Agents

- `$int_ret = SobelAgent::SrcBuf ($SobelAgent_);` — Get pointer to current source buffer for agent.

- `$int_ret = SobelAgent::SrcPtX ($SobelAgent_);` — Get x coordinate of source point (top-left point within source buffer) for agent.
- `$int_ret = SobelAgent::SrcPtY ($SobelAgent_);` — Get y coordinate of source point (top-left point within source buffer) for agent.
- `$int_ret = SobelAgent::SrcROI ($SobelAgent_);` — Get pointer to current source ROI for agent.
- `SobelAgent::SetSrcBuf ($SobelAgent_, $Buffer_srcBuf);` — Set source buffer for agent for next Go().
- `SobelAgent::SetSrcPt ($SobelAgent_, $Point_sPt);` — Set top-left point within source buffer where agent processing will begin.
- `SobelAgent::SetSrcROI ($SobelAgent_, $ROI_sRoi);` — Set region within source buffer that agent will process.

Transpose

Note: The size of the area being processed must be a multiple of 4 pixels.

See “Usage Examples” on page 6-112 for an example using the transpose factory and transpose agent.

TransposeFact — Transpose Agent Factory

Create

- `$int_ret = TransposeFact::TransposeFact ($Buffer_src, $Buffer_dst, $int_left, $int_top, $int_right, $int_bottom, $int_opcode);` — Create a transpose agent and return a pointer to it. The transpose agent created will process the area enclosed by (`$int_left`,`$int_top`) to (`$int_right`,`$int_bottom`) in the source buffer, `$Buffer_src`. `$Buffer_dst` will contain the resultant image.
- `$int_ret = TransposeFact::TransposeFact2 ($Buffer_src, $Buffer_dst, $int_opcode);` — Create a transpose agent and return a pointer to it. The transpose agent created will process the entire source buffer, `$Buffer_src`. `$Buffer_dst` will contain the resultant image.

Valid `$int_opcode` values are:

0 = rotate 90 degrees left
1 = rotate 90 degrees right
2 = rotate 180 degrees
3 = vertical flip
4 = horizontal flip

Transpose Agent

Delete

- `TransposeAgent::TransposeAgent_Delete ($TransposeAgent_);` — Delete a transpose agent.

Pre Go / Go

- `$int_ret = TransposeAgent::Go ($TransposeAgent_);` — Run a transpose agent.

Configuration/Information

- `$int_ret = TransposeAgent::DstBuf ($TransposeAgent_);` — Return a pointer to the destination buffer for agent.
- `$TransOpType_ret = TransposeAgent::Type ($TransposeAgent_);` — Return the type of transposition the agent is configured to do, where: { ROT90LEFT=0, ROT90RIGHT=1, ROT180=2, VFLIP=3, HFLIP=4 }.
- `$TransposeAgent::SetType ($TransposeAgent_, $TransOpType);` — Set the type of transposition for the agent, where: { ROT90LEFT=0, ROT90RIGHT=1, ROT180=2, VFLIP=3, HFLIP=4 }.

Common To All Agents

- `$int_ret = TransposeAgent::SrcBuf ($TransposeAgent_);` — Get pointer to current source buffer for agent.
- `$int_ret = TransposeAgent::SrcPtX ($TransposeAgent_);` — Get x coordinate of source point (top-left point within source buffer) for agent.

- `$int_ret = TransposeAgent::SrcPtY ($TransposeAgent_);` — Get y coordinate of source point (top-left point within source buffer) for agent.
- `$int_ret = TransposeAgent::SrcROI ($TransposeAgent_);` — Get pointer to current source ROI for agent.
- `TransposeAgent::SetSrcBuf ($TransposeAgent_, $Buffer_srcBuf);` — Set the source buffer pointer for the agent.
- `TransposeAgent::SetSrcPt ($TransposeAgent_, $Point_sPt);` — Set top-left point within source buffer where agent processing will begin.
- `TransposeAgent::SetSrcROI ($TransposeAgent_, $ROI_sRoi);` — Set region within source buffer that agent will process.

Warp

Warpfact — Warp Agent Factory

Create

- `$int_ret = WarpFact::WarpFact ($Buffer_, $Buffer_, $Transform_transform);` — Create a warp agent and return a pointer to it.

Warpagent — Warp Agent

Delete

- `WarpAgent::WarpAgent_Delete ($WarpAgent_);` — Delete a warp agent.

Pre Go / Go

- `$int_ret = WarpAgent::Go ($WarpAgent_);` — Run a warp agent.

Configuration/Information

- `$int_ret = WarpAgent::GetTransform ($WarpAgent_);` — Return a pointer to the transform being used by this warp agent.

- `$int_ret = WarpAgent::SetDstBuf ($WarpAgent_, $Buffer_);` — Set the destination buffer for agent to `$Buffer_`.
- `WarpAgent::SetTransform ($WarpAgent_, $Transform_transform);` — Set the transform being used by this warp agent to `$Transform_transform`. The types of transforms that warp can use, and methods to create and delete them, are given below.

Matrix3x3 - 3x3 Transform Matrix

- `$int_ret = Matrix3x3::Matrix3x3 ($double_a, $double_b, $double_c, $double_d, $double_e, $double_f, $double_g, $double_h, $double_i);` — Create a 3x3 matrix with the specified elements (going left to right, top to bottom).
- `Matrix3x3::Matrix3x3_Delete ($Matrix3x3_);` — Delete a 3x3 matrix.

Transform — Generic Transform Object

Transform Object

- `$int_ret = Transform::Transform_Transform ($int_l, $int_t, $int_r, $int_b);` — Create a transform object and return a pointer to it.
- `Transform::Transform_Delete ($Transform_);` — Delete a transform object.

Generic Transform Object

- `$int_ret = Transform::Generic_AllocatePoints ($int_number);` — Allocate memory for start and end points for a generic transform object. This allocates space for future calls to `AddPoints`. This must be paired with `FreePoints` to avoid memory leaks.
- `$int_ret = Transform::GenericTransform ($Fpoint_pPoints, $int_numSegs, $int_l, $int_t, $int_r, $int_b);` — Create a generic transform and return a pointer to it. The input `pPoints` is a pointer to an array of start and end points and `numSegs` indicates how many start or end points are in the array. The bounding rectangle for the transform is specified by `l`, `t`, `r`, `b`.
- `Transform::Generic_AddPoints ($Fpoint_, $int_index, $int_number, $double_startx, $double_starty, $double_endx, $double_endy);` —

Add a start and end point to the generic transform. The start point is (startx, starty), and the endpoint is (endx, endy). Index indicates that this point will be the indexth point in the transform. Number indicates how many points are allocated in the transform (from the `AllocatePoints()` call).

- `Transform::Generic_Delete ($GenericTransform_);` — Delete a generic transform object.
- `Transform::Generic_FreePoints ($Fpoint_);` — Free memory allocated in `AllocatePoints()`.

Four-Point Transform Object

- `$int_ret = Transform::FourPoint1 ($Quad_, $int_l, $int_t, $int_r, $int_b);` — Create a four point transform and return a pointer to it. The transform will transform from the input quadrilateral area (`$Quad` is a pointer to a `Quad` object) to a destination rectangle (`l, t, r, b`).
- `$int_ret = Transform::FourPoint2 ($int_in_l, $int_in_t, $int_in_r, $int_in_b, $int_dst_l, $int_dst_t, $int_dst_r, $int_dst_b, $double_angle);` — Create a four point transform and return a pointer to it. The transform will transform from the input rectangle (`in_l, in_t, in_r, in_b`) to the destination rectangle (`dst_l, dst_t, dst_r, dst_b`) with rotation specified by `angle`.
- `Transform::FourPoint_Delete ($FourPoint_);` — Delete a four point transform.

Annular Transform Object

- `$int_ret = Transform::Annular ($Annulus_, $int_l, $int_t, $int_r, $int_b);` — Create an annular transform and return a pointer to it. The transform will transform from the input annulus (`$Annulus`) to the destination rectangle (`l, t, r, b`).
- `Transform::Annular_Delete ($Annular_);` — Delete an annular transform.

3X3 Transform Object

- `$int_ret = Transform::Transform3x3 ($Matrix3x3_, $int_l, $int_t, $int_r, $int_b);` — Create a 3x3 transform and return a pointer to it.

- `Transform::Transform3x3_Delete ($Transform3x3_);` — Delete a 3x3 transform.

Utility Packages

Localize — Create Local Variables

- `Localize::makelocal (\$variable);` — Make `$variable` local to this instance of the script. This means `$variable` is unique for each Custom Step even if the steps are configured to run the same script. Put this call inside a package, but outside of all the subroutines in the package.

Kernal — Access to Target OS/Driver

- `$int_ret = Kernal::getDate ();` — Extract the date from the last call to `readTime()`. Range: 1-31
- `$int_ret = Kernal::getDay ();` — Extract the day from the last call to `readTime()`.
- `$int_ret = Kernal::getHour ();` — Extract the hour from the last call to `readTime()`. Range: 0-23.
- `$int_ret = Kernal::getMinutes ();` — Extract the number of minutes from the last call to `readTime()`. Range: 0-59.
- `$int_ret = Kernal::getMonth ();` — Extract the month from the last call to `readTime()`. Range: 1-12
- `$int_ret = Kernal::getSeconds ();` — Extract the number of seconds from the last call to `readTime()`. Range: 0-59.
- `$int_ret = Kernal::getYear ();` — Extract the year from the last call to `readTime()`.
- `$int_ret = Kernal::isNaN (double num);` — Return true (nonzero value) if `num` is not a number. Return false (zero) otherwise.
- `$int_ret = Kernal::memAvail ();` — Returns the number of bytes of memory available on the target.

- `$int_ret = Kernal::memFrag ()`; — Return the number of memory fragments in target memory.
- `$int_ret = Kernal::readTime ()`; — Read the current system time.

Note: One call to `readTime ()` is needed and then the information is extracted with calls to `getSeconds ()`, `getMinutes ()`, etc.

- `$int_ret = Kernal::taskIsSuspended ($int_taskId)`; — Return true if the specified `taskId` is suspended. Return false otherwise.
- `$int_ret = Kernal::taskNameTold ("pTaskName")`; — Return the Id of the task named "pTaskName".
- `Kernal::getTimeDateAsString (char* buffer)`; — Read the time/date as a string and put the resulting string into the character buffer specified. The character buffer should be long enough to hold the string created.

Kernal — Access to Target Illumination

- `void Kernal::setGreenFlash ($int_milliSeconds)`; — Turns on the green illumination LEDs for the specified number of milliseconds.
- `void Kernal::setLaserFlash ($int_milliSeconds)`; — Turns on the red targeting LEDs for the specified number of milliseconds.

Serial — Serial Port

- `$char*_ret = serial::ReadPort ($int_hComm, $int_number, $char_OnError, $bool_ErrorMsgs)`; — Read `$int_number` characters from the specified serial port. A pointer to the buffer containing these characters is returned in `ret`. If an error occurs, the character `$char_OnError` is written into the buffer. If `$bool_ErrorMsgs` is true, then any error messages are sent to FrontRunner's Debug Output Window.
- `$char_ret = serial::GetChar ($Char_Buf, $int_index)`; — Return the `indexth` character in the character buffer.

- `$int_hComm = serial::OpenPort ($int_portnumber, $int_flags, $int_mode);` — Open the serial port `$int_portnumber`. The `$int_flags` parameter indicates the type of access to the port: 0=read only, 1=write only, 2=read and write. This parameter is not used on the Visionscape GigE Camera. The `$int_mode` parameter is not used.
- `$int_ret = SetTimeouts ($int_hComm,$int_readIntervalTimeout, $int_readTimeoutPerChar, $int_readTimeoutTotal, $int_writeTimeoutPerChar, $int_writeTimeoutTotal)`

Note: This is only supported in Visionscape GigE Cameras.

The parameters are (all times are in milliseconds):

- `ReadIntervalTimeout` = max time between chars received
- `ReadTotalTimeoutMultiplier` = timeout for each char read
- `readTotalTimeoutConstant` = total timeout for read operation
- `writeTotalTimeoutMultiplier` = timeout for each char written
- `writeTotalTimeoutConstant` = total timeout for write operation
- `serial::ClosePort ($int_fd);` — Close the specified serial port.
- `serial::IoctlPort ($int_hComm, $int_baud, char*_parity, $int_databits, $double_stopbits);` — Configure the serial port. `$int_baud` sets the baud rate, `char*_parity` is a string indicating parity: "NONE", "ODD", "EVEN", "MARK", or "SPACE". `$int_databits` sets the number of data bits, `$double_stopbits` sets the stop bits: 1.0, 1.5 or 2.0.
- `serial::WritePort ($int_hComm, "pChar", $int_length, $bool_ErrorMsgs);` — Write the input string, `pChar`, to the specified serial port. Length is the number of characters in the string. If `$bool_ErrorMsgs` is true, then any error messages are sent to FrontRunner's Debug Output Window.

Video

- `$int_ret = AddButton (ulx, uly, width, height, text, bk_color, fg_color, text_color)` — Add a button to the display with upper left coordinate

(*ulx,uly*) that is *width*x*height* in size. “*text*” specifies the text on the button. *Bk_color*, *fg_color* and *text_color* specify the background color of the button, the foreground color of the button and the color of the button text, respectively. The number of buttons present, including this new button, is returned.

- `$int_ret = CreateRefMovButtons (bk_color, int fg_color, int text_color)` — Create four buttons for moving the reference point by +1 or -1. The buttons are labeled “X+”, “Y+”, “-X”, “-Y”. *Bk_color*, *fg_color* and *text_color* specify the background color of the button, the foreground color of the button and the color of the button text, respectively. The number of buttons created is returned. This value will be 4 unless one of the buttons was unable to be created. The buttons are drawn when the reference point is enabled for moving.
- `$int_ret = DragBoxBottom ()` — Return the bottom extent of the box (after dragging/resizing).
- `$int_ret = DragBoxLeft ()` — Return the left extent of the box (after dragging/resizing).
- `$int_ret = DragBoxRight ()` — Return the right extent of the box (after dragging/resizing).
- `$int_ret = DragBoxTop ()` — Return the top extent of the box (after dragging/resizing).
- `$int_ret = DragOctSizeH ()` — Return the height of the octagon (after dragging/resizing). The height here is really the length of the right or left edge of the octagon.
- `$int_ret = DragOctSizeW ()` — Return the width of the octagon (after dragging/resizing). The width here is really the length of the top or bottom edge of the octagon.
- `$int_ret = DragOctX ()` — Return the x coordinate of the octagon location (after dragging/resizing).
- `$int_ret = DragOctY ()` — Return the y coordinate of the octagon location (after dragging/resizing).
- `$int_ret = GetCurrentBuffer ()`; — Return a buffer pointer to the currently selected video buffer.

- `$int_ret = GetSnapBuffer ()` — Return a pointer to the buffer holding the current snap image.
- `$int_ret = GetUSecTick ()` — Return the current reading of an internal clock. Elapsed time in milliseconds is the number of ticks between two events divided by 50000.0.
- `$int_ret = RunningOnVxWorks ()`; — Return true if running on the target (on vxworks), false otherwise.
- `$int_ret = StartButtons ()` — Return the number of the button that was pushed. Zero is returned if no button is pushed.
- `$int_status = TakePicture (chan, bright, contrast, iters, strobe)` — Take a picture on the specified video channel using the specified strobe. Contrast and bright indicate the gain and offset used on the digitizer when the image is acquired. Iters is the number of times that the images is either dilated (iters > 0) or eroded (iters < 0) after acquisition. A nonzero status indicates failure.
- `AddRefObjExtent (start, left, top, right, bottom)`; — This function must be called with the extents of all objects being referenced in order to prevent these objects from being relocated offscreen. The start flag should be set when the first extents are input.
- `ClearOverlay (pl)` — Clear the overlay graphics. Optional parameter `pl` specifies which display buffer to draw into 0 or 1.
- `DeleteButtons ()` — Delete the buttons currently in use.
- `DragBox (l, t, r, b, color)` — Draw a box with extents `l, r, t, b` in the specified color and enable dragging/resizing of the box with the mouse.
- `DragOct (x, y, sizeW, sizeH, color)` — Draw an octagon at (x,y) of the specified size and color. Enable dragging/resizing of the octagon with the mouse.
- `DrawBlob ($blob, offsetX, offsetY, color,pl)` — Draw the input blob offset from it's location by (offsetX, offsetY) in the specified color. Optional parameter `pl` specifies which display buffer to draw into 0 or 1.

- `DrawBox (l, r, t, b, color, pl)` — Draw a box with extents l, r, t, b in the specified color. Optional parameter pl specifies which display buffer to draw into 0 or 1.
- `DrawCircle (x, y, size, color, pl)` — Draw a circle at (x,y) in the specified color. Size specifies the radius of the circle in pixels. Optional parameter pl specifies which display buffer to draw into 0 or 1.
- `DrawCrosshairs (x, y, color, length, pl)` — Draw crosshairs at (x,y) in the specified color. Length is the size of the crosshairs in pixels. Optional parameter pl specifies which display buffer to draw into 0 or 1.
- `DrawLine (x0, y0, x1, y1, color, pl)` — Draw a line from (x0,y0) to (x1,y1) in the specified color. Optional parameter pl specifies which display buffer to draw into 0 or 1.
- `DrawOct (x, y, sizeW, sizeH, color, pl)` — Draw an octagon at (x,y) in the specified color. SizeW specifies the length of the top or bottom side of the octagon in pixels. SizeH specifies the length of the right or left side of the octagon in pixels. Optional parameter pl specifies which display buffer to draw into 0 or 1.
- `DrawText (x0, y0, str, color, opaque, pl)` — Draw the text string str at (x0,y0) in the specified color. Optional parameter pl specifies which display buffer to draw into 0 or 1.
- `DumpMonster ()` — Print memory information (permanent and temporary memory available on each bank) for monster.
- `Freeze ()` — Change the video display from live to frozen.
- `LoadSnapTiff (fname)` — Load image from the specified file into the current snap buffer.
- `LockMonster ()` — Lock the monster resource.
- `position_event (int x, int y)` — Echo the (x,y) location of the mouse and the gray value of the pixel there.
- `SaveSnapTiff (fname)` — Save the image in the current snap buffer in the specified file.

- **SetButtonText** (buttonno, text) — Set the text for button number, buttonno, to the specified text.
- **SetCurrentBuffer** (bufno) — Set the current video buffer to the specified buffer. Bufno is a pointer to a Buffer object.
- **SetMorphlters** (iters) — Set number of time that images captured in live mode are either dilated (iters > 0) or eroded (iters < 0)
- **SetRefBoundingArea** (left, top, right, bottom); — This function enables you to restrict where objects can be moved by the reference point. An object can be moved anywhere within the bounding area. By default, the bounding area is full-screen. If this call is never made, the bounding area is the full screen.
- **ShowBuffer** (bufno); — Show the specified buffer on the display.
- **SizeBoxRestrict** (width, height); — Enable size restriction on the next box that is resized. Width and height specify the minimum width and height allowed for the box.
- **SizeBoxUnrestrict** (); — Disable size restriction for boxes.
- **StartLiveVideo** (cameraNo, bright, contrast, strobe) — Start live video on cameraNo using the specified strobe.
- **StopLiveVideo** (\$int_freeBuffers) — Stop live video. If freeBuffers is nonzero, then free all video buffers.
- **Thaw** () — Change the video display from frozen to live.
- **UnlockMonster** () — Unlock the monster resource.
- **UpdateVGA** () — Redraw the currently selected VGA buffer and associated graphics.
- **void DeleteRefMovButtons** () — Delete the buttons used for moving the reference point.

Line-by-line Analysis of a Script

The functionality of a Custom Step / Custom Vision Tool is determined by the perl script it runs. Below is an example perl script with a line-by-line analysis.

Line numbers are shown in () for reference.

Simple Example: Add Two Numbers

```
(01) use perlutil;
(02) package add_two_numbers;
(03) #
(04) # REGISTER this package so that custom steps know this package
    is available
(05) #
(06) perlutil::register("add_two_numbers");
(07)
(08) sub PreRun { }
(09) sub Update { }
(10) sub PostRun { }
(11) sub Draw { }
(12) sub Apply
(13) {
(14)     perlutil::clear_datum_lists;
(15)     perlutil::add_datum_int (2, 1, "FirstNum\nFirst Number", 1);
(16)     perlutil::add_datum_int (2, 1, "SecondNum\nSecond Number", 1);
(17)     perlutil::add_datum_int (1, 1, "Sum\nSum of Numbers", 1);
(18) }
(19) sub Run
(20) {
(21)     my $FirstNumber = perlutil::get_datum_int (0, 0);
(22)     my $SecondNumber = perlutil::get_datum_int (0, 1);
(23)     $Sum = $FirstNumber + $SecondNumber;
(24)     perlutil::set_datum_int (1, 0, $Sum);
(25)     $outstr = "Sum of numbers is " . $Sum;
(26)     printf $outstr;
(27)     perlutil::draw_text_out (95, 10, $outstr);
(28) }
(29) return 1;
```

Line-by-line Analysis

(01) This line indicates that the “perlutil” library module will be used. This library module is required for all custom steps.

(02) This line initiates the definition of a new “package”. Each custom step must be defined in a unique package.

(3-7) Register the package so that it is available to any custom step.

(8-11) The subroutines “PreRun”, “Update”, “PostRun”, and “Draw” contain empty definitions for this simple example.

(12) The subroutine “Apply” will be called when the user modifies the Datum page for the custom step. The first modification is likely to be the user selecting the name of the perl package defined on line (02). The Perl datum page presents a list of the available packages for a perl step. Once the user selects the desired package, the “Apply” subroutine (for that package) is called. It is responsible for defining the input and output datums that will be used and generated by the custom step.

(13) Perl uses curly braces like “C” to define blocks.

(14) This line removes all entries from the custom step’s input and output datum lists.

(15) This line adds the first input datum. The first argument specifies type. Type=2 indicates a user-editable entity (a resource datum). The second argument specifies that the datum should show up on the datum page for user editing. The third argument has two components separated by a “\n”. The first portion contains the symbolic name, and the second is the user name of the datum as it will appear on the datum page. Finally, the last argument specifies the default value for the datum.

(16) This line adds the second input datum.

(17) This line adds the first output datum (type is 1 for an output datum)

(18) Closing curly brace for the subroutine “Apply”

(19) Start of the subroutine “Run”, which will be used at run-time to implement the step.

(20) Perl uses curly braces like “C” to define blocks.

(21) Assigns the local variable \$FirstNumber with the value of input datum (indicated by first parameter = 0) number zero (indicated by the second parameter = 0).

Note: The input and output datum lists count from zero.

(22) Assigns the local variable `$SecondNumber` with the value of input datum (indicated by first parameter = 0) number one (indicated by second parameter = 1).

Note: The input and output datum lists count from zero.

(23) Assigns the global variable `$Sum` to the value of `$FirstNumber + $SecondNumber`.

(24) Sets the value of output datum (indicated by first parameter = 1) number zero (indicated by second parameter = 0) to the `$Sum`.

Note: The input and output datum lists count from zero.

(25) Setup a debug string (`$outstr`) as the concatenation of the literal string and the variable `$Sum` expressed in a string.

(26) Prints value of `$outstr` to the debug window.

(27) Call a perl utility to draw the given string at the given x,y location onto the display.

(28) Perl uses curly braces like “C” to define blocks.

(29) This line should be the last line in a script package file.

Notes

- To make a script available for use in a custom step or custom vision tool requires:
 - The file containing the script (filename) is included by the master script file **perlscr**. This is done by: `require filename`;
 - The file containing the package script is in the `perlmod` directory created at installation time. This is also where the master script file **perlscr** is located.

- The package must be registered. This is done by calling `perlutil::register("packagename")`, within the script and outside of all subroutines in the script.
- Support/library files are provided in the subdirectory `perlmod\support` of the installation directory. These are provided by Microscan and help provide access to internal vision functions.
- A file may contain many package scripts. Each package that the user wants available to a custom step must be registered separately. Also, a package within a file may have the same name as the file itself.
- The package **none** is included by default. This package shows the basic structure in a package script.
 - To “include” any of the library/support modules that are needed by a package script, type:

```
use supportfile;
```

For example:

```
use perlutil;
```

provides access to the `perlutil` package.

- Each script must contain the subroutines `Init`, `Cleanup`, `Start`, `Stop`, `Apply`, `Update`, `PreRun`, `PostRun`, `Run`, `Draw` within the package declaration. These subroutines can be stubs if they do not have processing to do. If a custom step is made trainable, then the `Train` subroutine must be in the script.
 - A call to the **register** function is included within the package but outside of any subroutines.
 - `Return 1;` is always the last statement in a package file. This statement occurs only once in the file regardless of how many package scripts are contained in the file.
- Comment lines may be in the master script file **pelscr** and any package file (*.pm). A comment is designated by a `#` character in the first column of a line.
- There are three types of datums available:

- An input datum (type=0) can be linked to any other similar-type datum in the Job.
- An output datum (type=1) is created an output by the custom step.
- A resource datum (type=2) is an input datum that you can edit.

The perl utility functions `add_datum_xxxx`, `set_datum_xxxx` and `get_datum_xxxx` are provided as part of the perlutil support package. These functions classify datums as either input (category = 0), which includes input datums and resource datums, or output (category = 1) for output datums.

- The default package script selection is **none**. If a Custom step is run with the **none** script selected, it will **always** fail.
- If changes are made to an existing script that involve internal processing only (no changes to input or output datums), then pressing the Re-Parse Package Script button will incorporate the changes. If any input/output datums are changed, then the Recreate Step Datums button must be pressed, which will cause the entire list of input and output datums to be recreated. This means that all datums must be reconnected. For example, if another step is using an output buffer from the custom step as its input, then the datum connection will need to be made again after datums are recreated.
- If a Job is loaded that contains a package script that is no longer available, then the package script **none** will automatically be selected. In this case, the Outputs information in the train window will indicate the problem, and/or a message will be sent to the Debug Output Window. Then, if this custom step is run, it will **always** fail until a valid script is selected.
- The master script **perlscr** is read either when the Re-Parse Package Script button is pressed, or when a custom step/custom vision tool is inserted, or when a Job containing a custom step is opened. Changes in **perlscr** will not take effect until one of these actions is taken.
- If this package script used `$FirstNumber` and `$SecondNumber` datums as inputs (type=0), then the inputs could be linked to another steps integer datums.

Usage Examples

For more examples, see “Basic Agents” on page 6-48.

TransposeFact Usage Example

```
use perlutil;
use TransposeAgent;
use TransposeFact;
use BufferDm;

package test_transpose;
#
# REGISTER this package so that perl steps know this package is available
#
perlutil::register("test_transpose");

sub Init    { }
sub Cleanup { }
sub PreRun  { }
sub Update  { }
sub PostRun { }
sub Draw    { }
sub Start   { }
sub Stop    { }
sub Apply
{
    perlutil::clear_datum_lists;
    perlutil::add_datum_int (2, 1, "Opcode (0,1,2,3,4)", 1);
    perlutil::add_datum_buffer (1, 1, "TransposeOutput\nTranspose Output");
}

sub Run
{
    # Valid values for "op" = { ROT90LEFT=0, ROT90RIGHT=1, ROT180=2, VFLIP=3, HFLIP=4 };
    # Note that for ROT90LEFT and ROT90RIGHT the width of the source = height of the dest
    # and the height of the source = width of the dest
    #
    $op = perlutil::get_datum_int (0, 0);
    $outBufferDm = perlutil::get_datum_buffer(1, 0);

    # get input buffer and ROI
    my $inbuf = perlutil::get_input_buf;
    my $l = perlutil::get_input_left();
    my $t = perlutil::get_input_top();
    my $r = perlutil::get_input_right();
    my $b = perlutil::get_input_bottom();

    # Transpose requires that size of area being processed is a multiple of 4 pixels.
    # Process less if necessary to enforce this rule, with the right and bottom areas
    # being trimmed.
    $width = ($r - $l) & ~3;# multiple of 4 width
    $height = ($b - $t) & ~3;# multiple of 4 height
    $r = $l + $width;# trim right if necessary
    $b = $t + $height;# trim bottom if necessary

    # Create output buffer for transposed image. !! NOTE SIZE OF OUTPUT BUFFER FOR ROTATE
    # 90 LEFT OR RIGHT !!
    if ($outbuf) {
        Buffer::Buffer_Delete($outbuf);
        $outbuf=0;
    }
}
```

```
}
if ($op==0 || $op==1) { # ROT90LEFT or ROT90RIGHT
    $outbuf = Buffer::Buffer2 ($inbuf, $height, $width);
} else {
    $outbuf = Buffer::Buffer2 ($inbuf, $width, $height);
}

$pTA = TransposeFact::TransposeFact ($inbuf, $outbuf, $l, $t, $r, $b, $op);
if ($pTA) {
    TransposeAgent::Go($pTA);
    TransposeAgent::TransposeAgent_Delete($pTA);
} else {
    printf "ERROR: Transpose agent not created!\n";
}

# This buffer goes into the output buffer datum.
BufferDm::SetBuffer($outBufferDm, $outbuf);
}

return 1;
```

HighlightBlobCustom and HighlightBlobTree Usage Examples

```

use perlutil;
use Buffer;
use BufferDm;
use Blob;
use BlobResult;
use BlobAgent;
use BlobFact;
use BlobTreeDm;
use Point;

#-----
#
#          ##### drawblob #####
#
# DO BLOB ANALYSIS ON THE INPUT ROI AND HIGHLIGHT EACH BLOB FOUND.
# BLOB WITH MAX AREA IS HIGHLIGHTED IN GREEN.
# BLOB WITH MIN AREA IS HIGHLIGHTED IN RED.
# REST OF BLOBS ARE HIGHLIGHTED IN YELLOW.
#-----

package drawblob;

perlutil::register("drawblob");

sub Init { }
sub Cleanup { }
sub Start { }
sub Stop { }
sub Update { }

sub Apply
{
    perlutil::clear_datum_lists;
    perlutil::add_datum_int (2, 1, "LowThreshold", 0); # INPUT: low threshold used in
        blob analysis
    perlutil::add_datum_int (2, 1, "HighThreshold", 128);# INPUT: high threshold used
        in blob analysis
    perlutil::add_datum_int (1, 1, "NumberOfBlobs", 1);# OUTPUT: number of blobs found
}

sub PreRun
{
    $res=0;
    $blobagent=0;
}

sub Run
{
    # PARAMETERS FOR BLOB
    my $lowThresh = perlutil::get_datum_int (0, 0);
    my $highThresh = perlutil::get_datum_int (0, 1);

    # GET INPUT BUFFER AND ROI
    my $mybuf = perlutil::get_input_buf();
    my $l = perlutil::get_input_left();
    my $t = perlutil::get_input_top();
    my $r = perlutil::get_input_right();
    my $b = perlutil::get_input_bottom();

```

```

# CREATE A BLOB AGENT AND BLOB RESULT (...AND DELETE THESE OBJECTS FROM PREVIOUS RUN)
if ($res) {
    BlobResult::BlobResult_Delete($res);
    $res = 0;
}
$res = BlobResult::BlobResult(0.25, 0.25);
if ($blobagent) {
    BlobAgent::BlobAgent_Delete($blobagent);
    $blobagent = 0;
}
$blobagent = BlobFact::BlobFact($mybuf, $l, $t, $r, $b, $res,
    $lowThresh, $highThresh);
BlobAgent::SetMinBlob($blobagent,10);
BlobAgent::SetMaxBlob($blobagent,1000000);
BlobAgent::SetProcessSwitches($blobagent,7);

# RUN BLOB AGENT
$status = BlobAgent::Go($blobagent);
}

sub PostRun { }

sub Draw
{
    # GET BUFFERDM POINTER FOR DRAW ROUTINE TO USE
    my $mybuf = perlutil::get_input_buf();
    my $mybufdm = perlutil::get_input_bufferdm();

    # IF BLOB AGENT RAN SUCCESSFULLY, EXTRACT EACH BLOB FROM THE BLOB RESULTS
    # PRINT SOME BLOB FEATURES IF DEBUG IS ENABLED
    my $nblobs = 0;
    if ($status==0 && $mybufdm) {
        $nblobs = BlobResult::NBlobs($res);
        perlutil::set_datum_int (1, 0, $nblobs); # SET OUTPUT DATUM

        # GET POINTER TO META FILE USED FOR DRAWING. CanDraw() WILL CREATE METAFILE
        #IF NECESSARY.
        BufferDm::CanDraw($mybufdm);
        $metaDC = BufferDm::GetMetaDC($mybufdm);

        # NEED POINTER TO THIS CUSTOM STEP. USED BY HighlightBlobCustom() FOR DRAWING
        # BLOBS INTO PARENT SNAPSHOT.
        $pThisStep = perlutil::get_myStepPointer();

        # FOR EACH BLOB FOUND OPTIONALLY OUTPUT SOME FEATURE INFORMATION
        # CHECK FOR BLOBS WITH MAX AND MIN AREAS FOR LATER
        $maxarea = 0;
        $maxblob = 0;
        $minarea = 99999;
        $minblob = 0;
        for ($i=0; $i<$nblobs; $i++) {
            $pBlob = BlobResult::GetBlob($res,$i);
            if ($pBlob) {
                $area = Blob::TotArea($pBlob);
                if ($area > $maxarea) {
                    $maxblob = $pBlob;
                    $maxarea = $area;
                }
                if ($area < $minarea) {
                    $minblob = $pBlob;
                    $minarea = $area;
                }
            }
        }
    }
}

```

```

    }

    # HIGHLIGHT THE BLOB IN YELLOW (i.e., draw the blob boundary)
    if ($metaDC) { Blob::HighlightBlobCustom($pBlob, 255, 255, 0,
        $pThisStep, $mybufdm); }
    }
}
if ($metaDC) {
    # HIGHLIGHT THE BLOB WITH MAX AREA IN GREEN
    Blob::HighlightBlobCustom($maxblob, 0, 255, 0, $pThisStep, $mybufdm);

    # HIGHLIGHT THE BLOB WITH MIN AREA IN RED
    Blob::HighlightBlobCustom($minblob, 255, 0, 0, $pThisStep, $mybufdm);
}
}
}

#-----
#
# ##### drawblobsimple #####
#
# DO BLOB ANALYSIS ON THE INPUT ROI & HILIGHT EACH BLOB FOUND. THIS SCRIPT DOES NOT CREATE
# A NEW BLOBAGENT AND BLOBRESULT EVERY RUN WHICH MAY SAVE TIME. IF DRAWING BLOB BOUNDARIES
# HOWEVER THE AMOUNT OF TIME SAVED WILL BE NEGLIGIBLE.
#
# ALL BLOBS ARE HIGHLIGHTED IN YELLOW.
#
# THIS SCRIPT SHOWS ONE WAY TO HANDLE MULTIPLE INSTANCES OF CUSTOM STEPS USING SAME SCRIPT.
#-----

package drawblobsimple;

perlutil::register("drawblobsimple");

# GLOBAL ARRAYS TO HOLD POINTERS TO BLOBAGENT AND BLOBRESULT
# USEFUL WHEN MULTIPLE INSTANCES OF THIS SCRIPT ARE IN A JOB, THIS INSURES THAT VARIABLES
# ARE UNIQUE FOR EACH INSTANCE OF THIS SCRIPT.
# (EX: UP TO 4 CUSTOM VISION TOOLS WITH "drawblobsimple" AS THE SELECTED SCRIPT CAN BE USED
@blobagent = (0,0,0,0); # expand array if >4 instances are needed
@blobresult = (0,0,0,0); # expand array if >4 instances are needed
@status = (0,0,0,0); # expand array if >4 instances are needed

sub Init { }
sub Cleanup { }
sub Start { }
sub Stop { }
sub Update { }

sub Apply
{
    perlutil::clear_datum_lists;
    perlutil::add_datum_int (2, 1, "LowThreshold", 0);# INPUT: low threshold used in
        blob analysis
    perlutil::add_datum_int (2, 1, "HighThreshold", 128);# INPUT: high threshold used
        in blob analysis
    perlutil::add_datum_int (2, 1, "Instance", 0);# which number "drawblobsimple" tool
        in job
        # ex: if there are 3 drawblobsimple tools in job:
        #     1st instance = 0, 2nd instance = 1, and 3rd instance = 2;

}

sub PreRun

```

```

{
    # PARAMETERS FOR BLOB
    my $lowThresh = perlutil::get_datum_int (0, 0);
    my $highThresh = perlutil::get_datum_int (0, 1);
    my $index = perlutil::get_datum_int (0, 2);# which tool (of multiple ones) are we running

    # GET INPUT BUFFER AND ROI
    my $mybuf = perlutil::get_input_buf();
    my $l = perlutil::get_input_left();
    my $t = perlutil::get_input_top();
    my $r = perlutil::get_input_right();
    my $b = perlutil::get_input_bottom();

    # CREATE A BLOB RESULT.
    $blobresult[$index] = BlobResult::BlobResult(0.25, 0.25);

    # CREATE AND CONFIGURE A BLOB AGENT.
    $blobagent[$index] = BlobFact::BlobFact($mybuf, $l, $t, $r, $b,
        $blobresult[$index], $lowThresh, $highThresh);
    BlobAgent::SetMinBlob($blobagent[$index],10);
    BlobAgent::SetMaxBlob($blobagent[$index],1000000);
    BlobAgent::SetProcessSwitches($blobagent[$index],7);
}

sub Run
{
    # WHICH TOOL (OF MULTIPLE ONES) ARE WE RUNNING?
    my $index = perlutil::get_datum_int (0, 2);

    # UPDATE INPUT BUFFER AND INPUT POINT
    $inbuf = perlutil::get_input_buf();
    $l = perlutil::get_input_left();
    $t = perlutil::get_input_top();
    $inpoint = Point::PointInt($l, $t);

    # CLEANUP PREVIOUS RESULTS
    BlobResult::CleanUp($blobresult[$index]);

    # RUN BLOB AGENT
    BlobAgent::SetSrcBuf($blobagent[$index], $inbuf);
    BlobAgent::SetSrcPt($blobagent[$index], $inpoint);
    BlobAgent::SetResult($blobagent[$index], $blobresult[$index]);
    $status[$index] = BlobAgent::Go($blobagent[$index]);

    # IF BLOB AGENT RUN FAILED FOR SOME REASON, CLEANUP BLOB RESULTS GENERATED
    if ($status[$index]) { BlobResult::CleanUp($blobresult[$index]); }

    Point::Point_Delete($inpoint);
}

sub PostRun
{
    # WHICH TOOL (OF MULTIPLE ONES) ARE WE RUNNING?
    my $index = perlutil::get_datum_int (0, 2);

    # DELETE RESOURCES CREATED IN PRE-RUN
    if ($blobresult[$index]) {
        BlobResult::BlobResult_Delete($blobresult[$index]);
        $blobresult[$index] = 0;
    }
}

```

```

    if ($blobagent[$index]) {
        BlobAgent::BlobAgent_Delete($blobagent[$index]);
        $blobagent[$index] = 0;
    }
}

sub Draw
{
    # WHICH TOOL (OF MULTIPLE ONES) ARE WE RUNNING?
    my $index = perlutil::get_datum_int (0, 2);

    # GET BUFFERDM POINTER FOR DRAW ROUTINE TO USE
    my $mybuf = perlutil::get_input_buf();
    my $mybufdm = perlutil::get_input_bufferdm();

    # IF BLOB AGENT RAN SUCCESSFULLY, EXTRACT EACH BLOB FROM THE BLOB RESULTS
    if ($status[$index]==0 && $mybufdm) {
        $nblobs = BlobResult::NBlobs($blobresult[$index]);

        # GET POINTER TO META FILE USED FOR DRAWING. CanDraw() WILL CREATE METAFILE
        # IF NECESSARY.
        BufferDm::CanDraw($mybufdm);
        $metaDC = BufferDm::GetMetaDC($mybufdm);

        # NEED POINTER TO THIS CUSTOM STEP. USED BY HighlightBlobCustom() FOR DRAWING
        # BLOBS INTO PARENT SNAPSHOT.
        $pThisStep = perlutil::get_myStepPointer();

        # HIGHLIGHT EACH BLOB FOUND IN YELLOW
        for ($i=0; $i<$nblobs; $i++) {
            $pBlob = BlobResult::GetBlob($blobresult[$index],$i);
            if ($pBlob) {
                if ($metaDC) { Blob::HighlightBlobCustom($pBlob, 255, 255, 0,
                    $pThisStep, $mybufdm); }
            }
        }
    }
}

#-----
#
# ##### drawblobtree #####
#
# DO BLOB ANALYSIS ON THE INPUT ROI AND HIGHLIGHT ENTIRE BLOB TREE.
# ALL BLOBS ARE HIGHLIGHTED IN YELLOW.
#
# IF ALL BLOBS FOUND ARE TO BE HIGHLIGHTED, THEN HIGHLIGHTING THE ENTIRE TREE AT ONE TIME
# IS MORE EFFICIENT. IF HIGHLIGHTING SELECTED BLOBS, THEN SCRIPTS SIMILAR TO drawblob OR
# drawblobsimple WOULD BE MORE APPROPRIATE.
#-----

package drawblobtree;

perlutil::register("drawblobtree");

sub Init { }
sub Cleanup { }
sub Start { }
sub Stop { }
sub Update { }

```



```

sub Apply
{
    perlutil::clear_datum_lists;
    perlutil::add_datum_int (2, 1, "LowThreshold", 0);# INPUT: low threshold used in
        blob analysis
    perlutil::add_datum_int (2, 1, "HighThreshold", 128);# INPUT: high threshold used in
        blob analysis
}

sub PreRun { }

sub Run
{
    # PARAMETERS FOR BLOB
    my $lowThresh = perlutil::get_datum_int (0, 0);
    my $highThresh = perlutil::get_datum_int (0, 1);

    # GET INPUT BUFFER AND ROI
    my $mybuf = perlutil::get_input_buf;
    my $l = perlutil::get_input_left();
    my $t = perlutil::get_input_top();
    my $r = perlutil::get_input_right();
    my $b = perlutil::get_input_bottom();

    # CREATE A BLOB AGENT AND BLOB RESULT (...AND DELETE THESE OBJECTS FROM PREVIOUS RUN)
    if ($res) {
        BlobResult::BlobResult_Delete($res);
        $res = 0;
    }
    $res = BlobResult::BlobResult(0.25, 0.25);
    if ($blobagent) {
        BlobAgent::BlobAgent_Delete($blobagent);
        $blobagent = 0;
    }
    $blobagent = BlobFact::BlobFact($mybuf, $l, $t, $r, $b, $res, $lowThresh, $highThresh);
    BlobAgent::SetMinBlob($blobagent,10);
    BlobAgent::SetMaxBlob($blobagent,1000000);
    BlobAgent::SetProcessSwitches($blobagent,7);

    # RUN BLOB AGENT
    $status = BlobAgent::Go($blobagent);
}

sub PostRun { }

sub Draw
{
    # GET BUFFERDM POINTER FOR DRAW ROUTINE TO USE
    $mybufdm = perlutil::get_input_bufferdm();

    # IF BLOB AGENT RAN SUCCESSFULLY, EXTRACT EACH BLOB FROM THE BLOB RESULTS
    if ($status==0 && $mybufdm) {
        # GET POINTER TO META FILE USED FOR DRAWING. CanDraw() WILL CREATE METAFIELD IF
        # NECESSARY.
        $metaDC = 0;
        BufferDm::CanDraw($mybufdm);
        $metaDC = BufferDm::GetMetaDC($mybufdm);

        # NEED POINTER TO THIS CUSTOM STEP. USED BY HighlightBlobCustom() FOR DRAWING BLOBS
        # INTO PARENT SNAPSHOT.
    }
}

```

```

        $pThisStep = perlutil::get_myStepPointer();

        # HIGHLIGHT BLOB TREE IN YELLOW
        if ($metaDC) { BlobResult::HighlightTreeCustom($res, 0, 255, 255,
            $pThisStep, $mybufdm); }
    }
}

#-----
#
# ##### drawblobtree_input #####
#
# BLOB TREE (result) IS INPUT TO THIS STEP. HIGHLIGHT INPUT BLOBTREE.
#-----

package drawblobtree_input;

perlutil::register("drawblobtree_input");

sub Init { }
sub Cleanup { }
sub Start { }
sub Stop { }
sub Update { }

sub Apply
{
    perlutil::clear_datum_lists;
    perlutil::add_datum_blobtree (0, 1, "Input Blob Tree", 1);
}

sub PreRun { }
sub Run { }
sub PostRun { }

sub Draw
{
    # GET BUFFERDM POINTER FOR DRAW ROUTINE TO USE
    $mybufdm = perlutil::get_input_bufferdm();

    # GET BLOB TREE FROM INPUT BLOB TREE DATUM
    $blobtreedm = perlutil::get_datum (0, 0);
    $res = BlobTreeDm::GetBlobTree ($blobtreedm);

    # HIGHLIGHT INPUT BLOB TREE IN YELLOW
    if ($mybufdm) {
        # GET POINTER TO META FILE USED FOR DRAWING. CanDraw() WILL CREATE METAFILE IF
        # NECESSARY.
        BufferDm::CanDraw($mybufdm);
        $metaDC = BufferDm::GetMetaDC($mybufdm);

        # NEED POINTER TO THIS CUSTOM STEP. USED BY HighlightBlobCustom() FOR DRAWING
        # BLOBS INTO PARENT SNAPSHOT.
        $pThisStep = perlutil::get_myStepPointer();

        # HIGHLIGHT BLOB TREE IN YELLOW
        if ($metaDC) { BlobResult::HighlightTreeCustom($res, 0, 255, 255,
            $pThisStep, $mybufdm); }
    }
}

return 1;

```

Gray Morphology Usage Example

```

use perlutil;
use GrayMorphAgent;
use GrayMorphFact;
use GrayMorphElem;
use BufferDm;

#####
# Below is an example script that can be used to perform various gray morphology operations.
# The user has 3 inputs available: opcode, polarity and iterations.
#
#   opcode values can be: 0=ERODE, 1=DILATE, 2=OPEN, 3=CLOSE, 4=GRADIENT, 5=TOPHAT, 6=WELL.
#   polarity values can be: 0=NORMAL, 1=REVERSE.
#   Iterations indicate how many passes of this morphological operation are performed.
#####

package test_GrayMorph;
#
# REGISTER this package so that perl steps know this package is available
#
perlutil::register("test_GrayMorph");

sub Init      { }
sub Cleanup   { }
sub Start     { }
sub Stop      { }
sub Update    { }

sub Apply
{
    perlutil::clear_datum_lists;
    perlutil::add_datum_int (2, 1, "Opcode", 1);
    perlutil::add_datum_int (2, 1, "Polarity", 1);
    perlutil::add_datum_int (2, 1, "Iterations", 1);
    perlutil::add_datum_buffer (1, 1, "GrayMorphOutput\nGray Morph Output");
}

sub PreRun
{
    # get input buffer and ROI
    my $inbuf = perlutil::get_input_buf;
    my $l = perlutil::get_input_left();
    my $t = perlutil::get_input_top();
    my $r = perlutil::get_input_right();
    my $b = perlutil::get_input_bottom();

    # Create output buffer for morphed image. Size will remain the same for all runs.
    $width = $r - $l;
    $height = $b - $t;
    $outbuf = Buffer::Buffer2 ($inbuf, $width, $height);

    # Create this custom 5x5 element table for the gray morph agent
    0  1  1  1  1
    1  0  1  1  1
    1  1  0  1  1
    1  1  1  0  1
    1  1  1  1  0
    $elemTbl = "0 1 1 1 1 1 0 1 1 1 1 0 1 1 1 1 0 1 1 1 1 0"; # delimit with
        commas or spaces
    $pElem = GrayMorphElem::GrayMorphElemCustom(2, 2, 5, 5, $elemTbl);
    # Create the gray morph agent

```

```

    $op = perlutil::get_datum_int (0, 0);
    $polarity = perlutil::get_datum_int (0, 1);
    $iterations = perlutil::get_datum_int (0, 2);
    $outBufferDm = perlutil::get_datum_buffer(1, 0);
    $pGMA = GrayMorphFact::GrayMorphFact ($inbuf, $outbuf, $l, $t, $r, $b, $op,
        $polarity, $iterations, $pElem);
}

sub Run
{
    # get input buffer and ROI
    $inbuf = perlutil::get_input_buf;
    $l = perlutil::get_input_left();
    $t = perlutil::get_input_top();
    $r = perlutil::get_input_right();
    $b = perlutil::get_input_bottom();

    # Update Gray Morph agent inputs and run
    $pt = Point::Point($l,$t);
    GrayMorphAgent::SetSrcBuf($pGMA, $inbuf);
    GrayMorphAgent::SetSrcPt($pGMA, $pt);
    GrayMorphAgent::Go($pGMA);

    # This buffer goes into the output buffer datum.
    BufferDm::SetBuffer($outBufferDm, $outbuf);
}

sub PostRun
{
    # Delete resources
    if ($outbuf) {
        Buffer::Buffer_Delete($outbuf);
        $outbuf=0;
    }
    if ($pElem) {
        GrayMorphElem::GrayMorphElem_Delete($pElem);
        $pElem=0;
    }
    if ($pGMA) {
        GrayMorphAgent::GrayMorphAgent_Delete($pGMA);
        $pGMA=0;
    }
}

return 1;

```

Masking Usage Example [maskable blob]

```

use perlutil;
use Buffer;
use Point;
use BlobFact;
use BlobAgent;
use BlobResult;
#use MaskDm;

##### MASKING BLOB PACKAGE #####

#
# This script does a masked blob analysis when used within a Custom Vision Tool. To perform
masked blob, the Custom # Vision Tool is inserted into a Dynamic Mask Tool. The Dynamic Mask
Tool creates a mask (from other mask producing

```

```

# step(s), ex: OCVTool) and makes it available to the Custom Vision Tool. The mask prohibits
certain areas from being
# processed by the blob agent created in this step.
#
#####

package maskblob;

#
# REGISTER this package so that perl steps know this package is available
#
perlutil::register("maskblob");

sub Init
{
    $res=0;
    $blobagent=0;
}

sub Cleanup
{
}

sub Apply
{
    perlutil::clear_datum_lists;
}

sub Update
{
}

sub Start
{
}

sub Stop
{
}

sub PreRun
{
    # MASK PREPARATIONS: MANDATORY AND ALWAYS IN PreRun
    perlutil::MaskPreRun();
    perlutil::CreateMaskBufs();

    # GET INPUT BUFFER AND ROI
    my $inbuf = perlutil::get_input_buf();
    my $l = perlutil::get_input_left();
    my $t = perlutil::get_input_top();
    my $r = perlutil::get_input_right();
    my $b = perlutil::get_input_bottom();

    # CREATE A BLOB RESULT
    $res = BlobResult::BlobResult(0.25, 0.25);

    # GET THE ROI FROM THE EMBEDDED MASKDM
    $roi = perlutil::GetMaskROI();

    # CREATE AND CONFIGURE NEW MASKABLE BLOB AGENT. AN ROI (from the
    # embedded maskdm to this custom step) NEEDS TO BE USED AS AN INPUT
    # PARAMETER WHEN DOING MASKING.
    $lowThresh=0;

```

```

    $highThresh=128;
    $blobagent = BlobFact::BlobFactMask($inbuf, $l, $t, $roi, $res, $lowThresh, $highThresh);
    BlobAgent::SetMinBlob($blobagent,10);
    BlobAgent::SetMaxBlob($blobagent,1000000);
    BlobAgent::SetProcessSwitches($blobagent,7);
}

sub Run
{
    # MASK PREPARATIONS: MANDATORY AND ALWAYS IN Run
    perlutil::MaskRun();

    # GET INPUT BUFFER. ROI IS ASSUMED TO BE THE SAME AS IT WAS AT PRERUN
    my $inbuf = perlutil::get_input_buf();

    # GET ROI FROM THE EMBEDDED MASKDM AND MAKE SURE BLOB IS USING THIS # UP-TO-DATE ROI
    $roi = perlutil::GetMaskROI();
    BlobAgent::SetSrcBuf($blobagent, $inbuf);
    BlobAgent::SetSrcROI($blobagent, $roi);

    # MAKE SURE MASK INFORMATION IS UP-TO-DATE
    perlutil::UpdateMaskBufs();

    # CLEAR RESULTS FROM PREVIOUS RUNS
    BlobResult::Cleanup($res);

    # RUN BLOB AGENT
    $status = BlobAgent::Go($blobagent);
}

sub PostRun
{
    # DELETE MASKABLE BLOB AGENT
    if ($blobagent) {
        BlobAgent::BlobAgent_Delete($blobagent);
        $blobagent = 0;
    }

    # DELETE THE BLOB RESULT TREE
    if ($res) {
        BlobResult::BlobResult_Delete($res);
        $res = 0;
    }

    # MASK CLEANUP: MANDATORY AND ALWAYS IN PostRun
    perlutil::MaskPostRun();
    perlutil::FreeMaskBufs();
}

sub Draw
{
    # GET BUFFERDM POINTER FOR DRAW ROUTINE TO USE
    $mybufdm = perlutil::get_input_bufferdm();

    # IF BLOB AGENT RAN SUCCESSFULLY, EXTRACT EACH BLOB FROM THE BLOB
    # RESULTS
    if ($status==0 && $mybufdm) {
        # GET POINTER TO META FILE USED FOR DRAWING.
        $metaDC = 0;
        BufferDm::CanDraw($mybufdm);
        $metaDC = BufferDm::GetMetaDC($mybufdm);

        # NEED POINTER TO THIS CUSTOM STEP. USED BY HighlightBlobCustom().
    }
}

```

```

    $pThisStep = perlutil::get_myStepPointer();

    # HIGHLIGHT BLOB TREE IN GREEN|BLUE
    # (BLOBS IN MASK ARE NOT HIGHLIGHTED)
    if ($metaDC) {
        BlobResult::HighlightTreeCustom($res, 0, 255, 255, $pThisStep, $mybufdm);
    }
}

return 1;

```

Masking Usage Example (maskable sobel)

```

use perlutil;
use Buffer;
use Point;
use SobelFact;
use SobelAgent;
use BufferDm;

##### MASKING SOBEL PACKAGE #####
# This script does a masked sobel when used within a Custom Vision Tool. To perform masked
sobel, the Custom Vision
# Tool is inserted into a Dynamic Mask Tool. The Dynamic Mask Tool creates a mask (from other
mask producing
# step(s), ex: OCVTool) and makes it available to the Custom Vision Tool. The mask prohibits
certain areas from being
# processed by the sobel agent created in this step.
#####

package masksobel;

#
# REGISTER this package so that perl steps know this package is available
#
perlutil::register("masksobel");

sub Init
{
    $pSobel=0;
}

sub Cleanup
{
}

sub Apply
{
    # THIS SCRIPT CREATES AN OUTPUT BUFFER HOLDING THE MASKED SOBEL
    # IMAGE.
    perlutil::clear_datum_lists;
    perlutil::add_datum_buffer (1, 1, "SobelImageOutput\nSobel Image Output");
}

sub Update
{
}

sub Start
{

```

```

}

sub Stop
{
}

sub PreRun
{
    # MASK PREPARATIONS: MANDATORY AND ALWAYS IN PreRun
    perlutil::MaskPreRun();
    perlutil::CreateMaskBufs();

    # GET INPUT BUFFER AND ROI
    my $inbuf = perlutil::get_input_buf();
    my $l = perlutil::get_input_left();
    my $t = perlutil::get_input_top();
    my $r = perlutil::get_input_right();
    my $b = perlutil::get_input_bottom();

    # MAKE A DESTINATION BUFFER THE SAME SIZE AS THE INPUT SEARCH AREA
    $outbuf = Buffer::Buffer2($inbuf, $r-$l, $b-$t);

    # GET THE ROI FROM THE EMBEDDED MASKDM
    $roi = perlutil::GetMaskROI();

    # CREATE MASKABLE SOBEL AGENT (2=SOBEL_MAG, 0=shift count, 120=threshold)
    $pSobel = SobelFact::SobelFactMask($inbuf, $outbuf, $l, $t, $roi, 2, 0, 120);
}

sub Run
{
    # MASK PREPARATIONS: MANDATORY AND ALWAYS IN Run
    perlutil::MaskRun();

    # GET A POINTER TO THE OUTPUT BUFFER DATUM
    $outBufferDm = perlutil::get_datum_buffer(1, 0);

    # GET INPUT BUFFER. ROI IS ASSUMED TO BE THE SAME AS IT WAS AT PRERUN
    my $inbuf = perlutil::get_input_buf();

    # GET ROI FROM THE EMBEDDED MASKDM AND MAKE SURE SOBEL IS USING
    # THIS UP-TO-DATE ROI
    $roi = perlutil::GetMaskROI();
    SobelAgent::SetSrcBuf($pSobel, $inbuf);
    SobelAgent::SetSrcROI($pSobel, $roi);

    # MAKE SURE MASK INFORMATION IS UP-TO-DATE
    perlutil::UpdateMaskBufs();

    # RUN MASKABLE SOBEL AGENT
    if ($pSobel) {
        SobelAgent::Go($pSobel);
    }

    # SET OUTPUT BUFFER DATUM TO HOLD THE SOBEL RESULT.
    BufferDm::SetBuffer($outBufferDm, $outbuf);
}

sub PostRun
{
    # DELETE MASKABLE SOBEL AGENT
    if ($pSobel) {
        SobelAgent::SobelAgent_Delete($pSobel);
    }
}

```



```

    }

    # MASK CLEANUP: MANDATORY AND ALWAYS IN PostRun
    perlutil::MaskPostRun();
    perlutil::FreeMaskBufs();
}

sub Draw
{
}

return 1;

```

Masking Usage Example (maskable correlation]

```

use perlutil;
use ROI;
use Buffer;
use Point;
use CorrTempl;
use CorrSrchAgent;
use CorrMatchFact;
#use MaskDm;
use PointDm;

##### MASKED CORRELATION #####

package maskcorr;

#
# REGISTER this package so that perl steps know this package is available
#
perlutil::register("maskcorr");

sub Init{ }
sub Cleanup { }

sub Apply
{
    perlutil::clear_datum_lists;
}

sub Update
{
}

sub Start
{
}

sub Stop
{
}

sub PreRun
{
    # MASK PREPARATIONS: MANDATORY AND ALWAYS IN PreRun
    perlutil::MaskPreRun();
    perlutil::CreateMaskBufs();
}

```

```

sub Run
{
    # MASK PREPARATIONS: MANDATORY AND ALWAYS IN Run
    perlutil::MaskRun();

    # GET INPUT BUFFER AND ROI
    $inbufdm = perlutil::get_input_bufferdm;
    $inbuf = perlutil::get_input_buf();
    my $l = perlutil::get_input_left();
    my $t = perlutil::get_input_top();
    my $r = perlutil::get_input_right();
    my $b = perlutil::get_input_bottom();

    # DELETE PREVIOUS TEMPLATE & AGENT SINCE THEY'RE CREATED AT RUN TIME
    if ($template) { CorrTempl::CorrSearchTempl_Delete ($template); }
    if ($corrAgent) { CorrSrchAgent::CorrSrchAgent_Delete ($corrAgent); }

    # GET THE ROI FROM THE EMBEDDED MASKDM
    $roi = perlutil::GetMaskROI();

    # CREATE TEMPLATE AND AGENT TO DO CORRELATION
    $w = ROI::Width($roi);
    $h = ROI::Height($roi);
    $shotx = $w/2.0;
    $shoty = $h/2.0;
    $template = CorrTempl::CorrSearchTemplMask ($inbuf, $roi, 0, 0, $shotx, $shoty);
    $corrAgent = CorrMatchFact::CorrSrchFact($inbuf, $l, $t, $l+$w, $t+$h, $template);

    # MAKE SURE MASK INFORMATION IS UP-TO-DATE
    perlutil::UpdateMaskBufs();

    # CONFIGURE AND RUN CORRELATION AGENT
    CorrSrchAgent::SetAcceptThresh ($corrAgent, 0.70, 1);
    CorrSrchAgent::SetRobustness ($corrAgent, 1.0);
    CorrSrchAgent::PreGo($corrAgent);
    $status = CorrSrchAgent::Go($corrAgent);

    # GET CORRELATION RESULTS
    $corrmax = CorrSrchAgent::GetResults2($corrAgent);
    $cpix = CorrSrchAgent::GetX($corrmax, 0);
    $cpy = CorrSrchAgent::GetY($corrmax, 0);
    $cps = CorrSrchAgent::GetScore($corrmax, 0);
}

sub PostRun
{
    # MASK CLEANUP: MANDATORY AND ALWAYS IN PostRun
    perlutil::MaskPostRun();
    perlutil::FreeMaskBufs();
}

sub Draw
{
    # DRAW CROSSHAIRS AT EACH SPOT WHERE THE TEMPLATE WAS FOUND
    BufferDm::CrossAt($inbufdm, $cpix, $cpy, 50);
}

return 1;

```

Mask Generation Example [1 input mask only]

```

use perlutil;
use Buffer;
use Point;
use SobelFact;
use SobelAgent;
use ShapeDm;
use MaskDm;

#####      GENERATE_MASK PACKAGE      #####
#
# This script generates an output mask within a Custom Vision Tool. The output mask is the
# result of processing a single
# (1) input. Multiple inputs combined into one output mask is not handled in this script.
#
#####
package generate_mask;

#
# REGISTER this package so that perl steps know this package is available
#
perlutil::register("generate_mask");

# GLOBAL ARRAYS TO HOLD POINTERS TO MASKLIST AND RECTANGULAR EXTENTS OF MASKS WITHIN THE
# INPUT BUFFER. THESE ALLOW FOR UP TO 4 INPUT MASKS.
@masklist = (0,0,0,0);    # expand array if >4 masks will be allowed
@mask_l   = (0,0,0,0);    # expand array if >4 masks will be allowed
@mask_t   = (0,0,0,0);    # expand array if >4 masks will be allowed
@mask_r   = (0,0,0,0);    # expand array if >4 masks will be allowed
@mask_b   = (0,0,0,0);    # expand array if >4 masks will be allowed

# GLOBAL VARIABLES
$LastError;

sub Init
{
    # MAKE 2ND ROI VISIBLE AND ROTATABLE. THIS ROI DETERMINES INPUT AREA
    # FOR MASK.
    $secondRoi = perlutil::get_datum_rectshape(0, 0);
    perlutil::ShowRectShape($secondRoi); # remember to HideRectShape before deleting it!
}

sub Cleanup
{
    # MUST CALL HideRectShape() FOR ALL RECTSHAPE DATUMS THE SCRIPT HAS
    # ADDED AND MADE VISIBLE
    perlutil::HideRectShape($secondRoi);
}

sub Apply
{
    perlutil::clear_datum_lists;

    # ADD ONE RECTSHAPE FOR EACH SUBMASK ... MUST BE 1st IN THE INPUT LIST
    perlutil::add_datum_rectshape(2, 1, "Mask1_Roi\nMask #1 ROI");

    # HOW MANY SUBMASKS. SET TO "1" HERE.
    perlutil::add_datum_int (2, 1, "NumberOfSubMasks\nNumber Of SubMasks", 1);

    # ONE GROUP OF INPUTS FOR EACH SUBMASK
    perlutil::add_datum_int (2, 1,

```

```

        "MaskGeneration(0=fill,1=thresh)\nMaskGeneration(0=fill,1=thresh)", 1);
perlutil::add_datum_int (2, 1, "Polarity(0=LonD, 1=DonL)\nPolarity (0=LonD, 1=DonL)", 0);
perlutil::add_datum_int (2, 1, "LowThreshold\nLow Threshold", 0);
perlutil::add_datum_int (2, 1, "HighThreshold\nHigh Threshold", 128);
perlutil::add_datum_int (2, 1,
        "MaskAdjust(0=erode,1=Dilate)\nMaskAdjust(0=erode,1=Dilate)", 0);
perlutil::add_datum_int (2, 1, "NumberOfAdjustments\nNumber of Adjustments", 0);

# OUTPUT = ONE MASK DATUM
perlutil::add_datum_mask (1, 1, "OutputMask\nOutput Mask", 0);
}

sub Update
{
}

sub Start
{
}

sub Stop
{
}

sub PreRun
{
}

sub Run
{
    $LastError="no error";

    # GET INPUT BUFFER AND ROI FOR THIS TOOL
    my $inbuf = perlutil::get_input_buf();
    my $l = perlutil::get_input_left();
    my $t = perlutil::get_input_top();
    my $r = perlutil::get_input_right();
    my $b = perlutil::get_input_bottom();

    # MUST LET STEP KNOW HOW MANY INDIVIDUAL SUB-MASKS WILL BE USED TO
    # CREATE THE FINAL MASK DATUM LIMIT NUMBER OF MASKS TO 1-4 UNLESS
    # MORE SPACE IS ALLOCATED IN THE GLOBAL ARRAYS AT TOP OF FILE
    $numberOfMasks = perlutil::get_datum_int (0, 1);
    if ($numberOfMasks < 1) { $numberOfMasks=1; }
    if ($numberOfMasks > 4) { $numberOfMasks=4; }

    # SET THE SIZE OF THE MASKDM THIS STEP WILL PRODUCE TO BE THE SIZE OF
    # THE INPUT ROI
    $maskOutDm = perlutil::get_datum (1, 0);
    MaskDm::SetSizeAndAlloc8Bit($maskOutDm, $r-$l, $b-$t);
    MaskDm::Notify8BitChanged($maskOutDm);

    # ALLOCATE A MASK LIST
    $pMaskList = perlutil::CreateMaskList($numberOfMasks);

    $status=1;
    for ($i=0; $i<$numberOfMasks; $i++) {
        # GET POSITION AND EXTENTS FOR EACH MASK ROI. ROI/SHAPE
        # DATUMS ARE 1ST IN INPUT LIST!
        $maskRoi[$i] = perlutil::get_datum_rectshape(0, $i);
        $mask_l[$i] = ShapeDm::GetBoundingRectLeft($maskRoi[$i]);
        $mask_t[$i] = ShapeDm::GetBoundingRectTop($maskRoi[$i]);
        $mask_r[$i] = ShapeDm::GetBoundingRectRight($maskRoi[$i]);
    }
}

```

```

$mask_b[$i] = ShapeDm::GetBoundingRectBottom($maskRoi[$i]);

# CREATE RESOURCES NEEDED FOR THIS MASK
$filltype = perlutil::get_datum_int (0,2);
$polarity = perlutil::get_datum_int (0,3);
$lowthresh = perlutil::get_datum_int (0,4);
$highthresh = perlutil::get_datum_int (0,5);
$adjusttype = perlutil::get_datum_int (0,6);
$numAdjusts = perlutil::get_datum_int (0,7);
$pResources = perlutil::CreateMaskResources($i, $inbuf, $mask_l[$i],
    $mask_t[$i], $mask_r[$i], $mask_b[$i], $filltype, $lowthresh,
    $highthresh, $adjusttype, $numAdjusts);
if ($LastError ne "no error") {
    printf "ERROR ($LastError) DURING MASK RESOURCE ALLOC\n";
}
}

if ($status && ($LastError eq "no error")) {
    for ($i=0; $i<$numberOfMasks; $i++) {
        # CREATE EACH MASK
        $masklist[$i] = perlutil::CreateMask($i, $filltype, $polarity,
            $highthresh, $numAdjusts, $pResources);

        # ADD MASK TO THE LIST FOR COMBINING LATER. PASS IN MASK
        # DATA ADDRESS AND X,Y OFFSETS FOR THIS MASK FROM THE
        # UPPER LEFT CORNER OF THE INPUT ROI.
        perlutil::AddToMaskList($pMaskList, $i, $masklist[$i],
            $mask_l[$i]-$l, $mask_t[$i]-$t);
    }

    # COMBINE ALL MASKS AND PUT INTO THE OUTPUT MASK DATUM
    $status = perlutil::CombineMasks($numberOfMasks, $pMaskList, $maskOutDm);
}

# FREE RESOURCES NOW SINCE THEY ARE ALLOCATED IN SUB RUN.
perlutil::DeleteMaskResources($pResources);
perlutil::DeleteMaskList($pMaskList);
}

sub PostRun
{
}

sub Draw
{
}

return 1;

```

Mask Generation Example [multiple input masks]

```

use perlutil;
use Buffer;
use Point;
use SobelFact;
use SobelAgent;
use ShapeDm;
use MaskDm;

```

```

##### GENERATE_MASK PACKAGE #####

```

```

package generate_mask;

#
# REGISTER this package so that perl steps know this package is available
#
perlutil::register("generate_mask");

# -----
# THERE ARE 5 DATUMS/PARAMETERS FOR EACH SUBMASK IN THE INPUT DATUM LIST
# -----
my $datumsPerMask=5;

# ARRAYS TO HOLD POINTERS TO MASKLIST AND RECTANGULAR EXTENTS OF MASK
# ROIS WITHIN THE INPUT BUFFER. THIS ALLOWS FOR UP TO 4 INPUT MASKS.
@pResources=(0,0,0,0); # expand array if >4 masks will be allowed
@masklist = (0,0,0,0); # expand array if >4 masks will be allowed
@mask_l   = (0,0,0,0); # expand array if >4 masks will be allowed
@mask_t   = (0,0,0,0); # expand array if >4 masks will be allowed
@mask_r   = (0,0,0,0); # expand array if >4 masks will be allowed
@mask_b   = (0,0,0,0); # expand array if >4 masks will be allowed

# -----
# Variables to setup at parse time
# -----
$LastError;
$submasksVisible=0;
$numMasksChanged=0;
$newNumberOfMasks=0;

sub Init
{
    # Make submask ROI(s) visible
    if ($submasksVisible == 0) {
        for ($i=0; $i<$numberOfMasks; $i++) {
            # ASSUMES THAT RECTSHAPES ARE FIRST IN THE INPUT DATUM LIST
            $maskRoi = perlutil::get_datum_rectshape(0, $i+1);
            perlutil::ShowRectShape($maskRoi); # remember to HideRectShape before deleting it!
        }
    }
    $submasksVisible=1;
}

sub Cleanup
{
    if ($submasksVisible) {
        for ($i=0; $i<$numberOfMasks; $i++) {
            # must call HideRectShape for all rectShape datums the script has added AND
            # made visible
            $maskRoi = perlutil::get_datum_rectshape(0, $i+1);
            perlutil::HideRectShape($maskRoi);
        }
    }
    $submasksVisible=0;
}

sub Apply
{
    # HIDE THE OLD SUBMASK ROI(s) IF THEY ARE VISIBLE
    if ($submasksVisible) {
        for ($i=0; $i<$numberOfMasks; $i++) {

```

```

        # must call HideRectShape for all rectShape datums the script has added AND
        # made visible
        $maskRoi = perlutil::get_datum_rectshape(0, $i+1);
        perlutil::HideRectShape($maskRoi);
    }
    $submasksVisible=0;
}

perlutil::clear_datum_lists;

# UPDATE SUB MASK COUNT, $numberOfMasks, IF NUMBER OF MASKS HAS CHANGED
if ($numMasksChanged) {
    $numberOfMasks = $newNumberOfMasks;
}

# HOW MANY SUBMASKS
if ($numberOfMasks == 0) { # IF NOT SETUP YET
    perlutil::add_datum_int(2, 1, "NumberOfSubMasks\nNumber_Of_SubMasks", 1); #
    default to one mask
    $numberOfMasks=1;
} else {
    perlutil::add_datum_int(2, 1, "NumberOfSubMasks\nNumber Of SubMasks", $numberOfMasks);
}

# ONE FOR EACH SUBMASK ... MUST BE FIRST IN THE INPUT LIST
if ($numberOfMasks > 0) {
    # MUST ADD RECTSHAPE FOR EACH SUBMASK
    perlutil::add_datum_rectshape(2, 1, "Mask1_Roi\nMask #1 ROI");
}
if ($numberOfMasks > 1) {
    # MUST ADD RECTSHAPE FOR EACH SUBMASK
    perlutil::add_datum_rectshape(2, 1, "Mask2_Roi\nMask #2 ROI");
}
if ($numberOfMasks > 2) {
    # MUST ADD RECTSHAPE FOR EACH SUBMASK
    perlutil::add_datum_rectshape(2, 1, "Mask3_Roi\nMask #3 ROI");
}
if ($numberOfMasks > 3) {
    # MUST ADD RECTSHAPE FOR EACH SUBMASK
    perlutil::add_datum_rectshape(2, 1, "Mask4_Roi\nMask #4 ROI");
}
if ($numberOfMasks > 0) {
    # ONE GROUP OF INPUTS FOR EACH SUBMASK (MASK #1)
    perlutil::add_datum_int(2, 1, "MSK1:Type (0=fill,1=thresh)\nMSK1:
        Type (0=fill,1=thresh)", 1);
    perlutil::add_datum_int(2, 1, "MSK1:PixToMask(0=LT,1=DK)\nMSK1:PixToMask(0=LT,1=DK)",
        0);
    perlutil::add_datum_int(2, 1, "MSK1:Threshold\nMSK1:Threshold", 128);
    perlutil::add_datum_int(2, 1,
        "MSK1:Adjust(0=erode,1=dilate)\nMSK1:Adjust(0=erode,1=dilate)", 0);
    perlutil::add_datum_int(2, 1, "MSK1:NumberOfAdjusts\nMSK1:Number of Adjusts", 0);
}

if ($numberOfMasks > 1) {
    # ONE GROUP OF INPUTS FOR EACH SUBMASK (MASK #2)
    perlutil::add_datum_int(2, 1, "MSK2:Type (0=fill,1=thresh)\nMSK2:
        Type (0=fill,1=thresh)", 1);
    perlutil::add_datum_int(2, 1, "MSK2:PixToMask(0=LT,1=DK)\nMSK2:PixToMask(0=LT,1=DK)",
        0);
    perlutil::add_datum_int(2, 1, "MSK2:Threshold\nMSK2:Threshold", 128);
    perlutil::add_datum_int(2, 1,
        "MSK2:Adjust(0=erode,1=dilate)\nMSK2:Adjust(0=erode,1=dilate)", 0);
    perlutil::add_datum_int(2, 1, "MSK2:NumberOfAdjusts\nMSK2:Number of Adjusts", 0);
}

```

```

}

if ($numberOfMasks > 2) {
    # ONE GROUP OF INPUTS FOR EACH SUBMASK (MASK #3)
    perlutil::add_datum_int(2, 1, "MSK3:Type (0=fill,1=thresh)\nMSK3:
        Type (0=fill,1=thresh)", 1);
    perlutil::add_datum_int(2, 1, "MSK3:PixToMask(0=LT,1=DK)\nMSK3:PixToMask(0=LT,1=DK)",
        0);
    perlutil::add_datum_int(2, 1, "MSK3:Threshold\nMSK3:Threshold", 128);
    perlutil::add_datum_int(2, 1,
        "MSK3:Adjust(0=erode,1=dilate)\nMSK3:Adjust(0=erode,1=dilate)", 0);
    perlutil::add_datum_int(2, 1, "MSK3:NumberOfAdjusts\nMSK3:Number of Adjusts", 0);
}

if ($numberOfMasks > 3) {
    # ONE GROUP OF INPUTS FOR EACH SUBMASK (MASK #4)
    perlutil::add_datum_int(2, 1, "MSK4:Type (0=fill,1=thresh)\nMSK4:
        Type (0=fill,1=thresh)", 1);
    perlutil::add_datum_int(2, 1, "MSK4:PixToMask(0=LT,1=DK)\nMSK4:PixToMask(0=LT,1=DK)",
        0);
    perlutil::add_datum_int(2, 1, "MSK4:Threshold\nMSK4:Threshold", 128);
    perlutil::add_datum_int(2, 1,
        "MSK4:Adjust(0=erode,1=dilate)\nMSK4:Adjust(0=erode,1=dilate)", 0);
    perlutil::add_datum_int(2, 1, "MSK4:NumberOfAdjusts\nMSK4:Number of Adjusts", 0);
}

# OUTPUT = ONE MASK DATUM
perlutil::add_datum_mask(1, 1, "OutputMask\nOutput Mask", 0);

# MAKE SUBMASK ROI(s) VISIBLE
if ($submasksVisible == 0) {
    for ($i=0; $i<$numberOfMasks; $i++) {
        # ASSUMES THAT RECTSHAPES ARE FIRST IN THE INPUT DATUM LIST
        $maskRoi = perlutil::get_datum_rectshape(0, $i+1);
        perlutil::ShowRectShape($maskRoi); # remember to HideRectShape before deleting it!
        printf("\nSHOW RECTSHAPE %d", $i);
    }
    $submasksVisible=1;
}

}

sub Update
{
    # UPDATE THE NUMBER OF MASKS (EITHER USER HAS CHANGED OR POSTSTREAM OF STEP)
    # printf ("\nUPDATE:: NUMBER OF MASKS WAS %d", $numberOfMasks); # always zero!!!
    $newNumberOfMasks = perlutil::get_datum_int (0, 0);
    # printf ("\nUPDATE:: NEW NUMBER OF MASKS WAS %d", $newNumberOfMasks);
    if ( ($newNumberOfMasks != $numberOfMasks) ) {
        $numMasksChanged = 1;
    }
}

sub Start
{
}

sub Stop
{
}

sub PreRun
{

```



```

}

sub Run
{
    $LastError="no error";

    # UPDATE SUB MASK COUNT IF IT HAS CHANGED
    if ($newNumberOfMasks != $numberOfMasks) {
        $numberOfMasks = $newNumberOfMasks;
    }

    # GET INPUT BUFFER AND ROI FOR THIS TOOL
    my $inbuf = perlutil::get_input_buf();
    my $l = perlutil::get_input_left();
    my $t = perlutil::get_input_top();
    my $r = perlutil::get_input_right();
    my $b = perlutil::get_input_bottom();
##    printf "ROI = $l $t $r $b";

    # MUST LET STEP KNOW HOW MANY INDIVIDUAL SUB-MASKS WILL BE USED TO CREATE FINAL MASK DATUM
    # LIMIT NUMBER OF MASKS TO 1-4 UNLESS MORE SPACE IS ALLOCATED IN THE ARRAYS AT TOP OF FILE
    if ($numberOfMasks < 1) { $numberOfMasks=1; }
    if ($numberOfMasks > 4) { $numberOfMasks=4; }
##    printf "Num Masks = $numberOfMasks";

    # SET THE SIZE OF THE MASKDM THIS STEP WILL PRODUCE TO BE THE SIZE OF THE INPUT ROI
    $maskOutDm = perlutil::get_datum (1, 0);
    MaskDm::SetSizeAndAlloc8Bit($maskOutDm, $r-$l, $b-$t);
    MaskDm::Notify8BitChanged($maskOutDm);

    # ALLOCATE A MASK LIST
    $pMaskList = perlutil::CreateMaskList($numberOfMasks);

    $status=1;
    for ($i=0; $i<$numberOfMasks; $i++) {
        # GET POSITION AND EXTENTS FOR EACH MASK ROI
##        printf "PROCESS MASK $i";
        $maskRoi[$i] = perlutil::get_datum_rectshape(0, $i+1); # ROI/SHAPE DATUMS MUST BE
                        1st IN INPUT LIST!
        $mask_l[$i] = ShapeDm::GetBoundingRectLeft($maskRoi[$i]);
        $mask_t[$i] = ShapeDm::GetBoundingRectTop($maskRoi[$i]);
        $mask_r[$i] = ShapeDm::GetBoundingRectRight($maskRoi[$i]);
        $mask_b[$i] = ShapeDm::GetBoundingRectBottom($maskRoi[$i]);
##        printf "Mask[$i] ROI = $mask_l[$i] $mask_t[$i] $mask_r[$i] $mask_b[$i]";

        # CREATE RESOURCES NEEDED FOR THIS MASK
        $baseDatum = $datumsPerMask * $i + $numberOfMasks + 1; # ZERO BASED INDEX OF PARAMETER
        # DATUMS FOR THIS SUBMASK (i.e., location in input datum list)
##        printf "baseDatum = $baseDatum";
        $filltype = perlutil::get_datum_int (0, $baseDatum);
        $polarity = perlutil::get_datum_int (0, $baseDatum + 1);
        $thresh = perlutil::get_datum_int (0, $baseDatum + 2);
        $adjusttype = perlutil::get_datum_int (0, $baseDatum + 3);
        $numAdjusts = perlutil::get_datum_int (0, $baseDatum + 4);
        $lowthresh = $thresh; # not used anymore by CreateMaskResources.
        $highthresh = $thresh; # not used anymore by CreateMaskResources.
        $pResources[$i] = perlutil::CreateMaskResources($i, $inbuf, $mask_l[$i],
            $mask_t[$i], $mask_r[$i], $mask_b[$i], $filltype, $lowthresh,
            $highthresh, $adjusttype, $numAdjusts);

        # IF RESOURCES ARE OK, THEN CREATE THE MASK
        if ($LastError ne "no error") {

```

```

        printf "ERROR ($LastError) DURING MASK RESOURCE ALLOCATION\n";
    } else {
##      printf "CREATE MASK $i";
##      printf "polarity = $polarity, thresh=$thresh";
        $masklist[$i] = perlutil::CreateMask($i, $filltype, $polarity,
            $thresh, $numAdjusts, $pResources[$i]);
        # ADD THIS MASK TO THE LIST FOR COMBINING LATER. PASS IN MASK DATA ADDRESS
        # (BUFFER*) AND X,Y OFFSETS FOR THIS.
        # MASK FROM THE UPPER LEFT CORNER OF THE INPUT ROI.
        perlutil::AddToMaskList($pMaskList, $i, $masklist[$i],
            $mask_l[$i]-$l, $mask_t[$i]-$t);
    }
}

if ($LastError eq "no error") {
    # COMBINE ALL MASKS AND PUT INTO THE OUTPUT MASK DATUM
    $status = perlutil::CombineMasks($numberOfMasks, $pMaskList, $maskOutDm);
}

for ($i=0; $i<$numberOfMasks; $i++) {
    perlutil::DeleteMaskResources($pResources[$i]);
}
perlutil::DeleteMaskList($pMaskList);
}

sub PostRun
{
}

sub Draw
{
}

return 1;

```

Read Digital IO Example

```

use perlutil;
use IOListDm;

##### read digital I/O #####
package read_Digital_IO;

#
# REGISTER this package so that perl steps know this package is available
#
perlutil::register("read_Digital_IO");

sub Init{ }
sub Cleanup { }

sub Apply
{
    # USE AN IOLIST DATUM TO ACCESS ALL TYPES OF IO.
    perlutil::clear_datum_lists;
    perlutil::add_datum_iolistdm(2, 1, "IO List\nIO List");
    perlutil::add_datum_int(1, 1, "DIORedValue\nDigital Input Value", 0);
}

sub Update

```

```

{
}

sub PreRun
{
    # PREPARE THE IO OBJECT.
    $pIOListDm = perlutil::get_datum_iolistdm(0,0);
    if ($pIOListDm != 0)
    {
        IOListDm::PrepareIO($pIOListDm);
    }
}

sub Run
{
    $pIOListDm = perlutil::get_datum_iolistdm(0,0);
    if ($pIOListDm != 0)
    {
        # READ IO AS SPECIFIED IN THE IOLIST DATUM.
        $status = IOListDm::ReadMyIOPoint($pIOListDm);
        perlutil::set_datum_int(1,0, $status);

        #printf "ReadMyIOPoint -- %08X\n", $status;    # useful for debugging

        perlutil::SetPassed(1);
    }
    else
    {
        perlutil::SetPassed(0);
    }
}

sub PostRun
{
}

sub Draw
{
}

sub Start
{
}

sub Stop
{
}

return 1;

```

Write Digital IO Example

```

use perlutil;
use IOListDm;

##### write digital I/O #####
package write_Digital_IO;

#
# REGISTER this package so that perl steps know this package is available
#
perlutil::register("write_Digital_IO");

```

```
sub Init{ }
sub Cleanup { }

sub Apply
{
    # USE AN IOLIST DATUM TO ACCESS ALL TYPES OF IO.
    perlutil::clear_datum_lists;
    perlutil::add_datum_iolistdm(2, 1, "IO List\nIO List");
    perlutil::add_datum_int(2, 1, "DIOWriteValue\nDigital Output Value", 0);
}

sub Update
{
}

sub PreRun
{
    # PREPARE THE IO OBJECT
    $pIOListDm = perlutil::get_datum_iolistdm(0,0);
    if ($pIOListDm != 0)
    {
        IOListDm::PrepareIO($pIOListDm);
    }
}

sub Run
{
    $pIOListDm = perlutil::get_datum_iolistdm(0,0);
    if ($pIOListDm != 0)
    {
        # WRITE IO AS SPECIFIED IN THE IOLIST DATUM.
        $value = perlutil::get_datum_int(0,1);
        IOListDm::WriteMyIOPoint($pIOListDm, $value);

        perlutil::SetPassed(1);
    }
    else
    {
        perlutil::SetPassed(0);
    }
}

sub PostRun
{
}

sub Draw
{
}

sub Start
{
}

sub Stop
{
}

return 1;
```

EnumDm Example

```

use perlutil;
use EnumDm;

#-----
#   Basic use of EnumDm functions. This script does not perform any particular function.
#-----

package test_enumdm;
#
# REGISTER this package so that perl steps know this package is available
#
perlutil::register("test_enumdm");

sub Init    { }
sub Cleanup { }
sub PreRun  { }
sub Update  { }
sub PostRun { }
sub Draw    { }
sub Start   { }
sub Stop    { }

sub Apply
{
    # SETUP OPTIONS ARRAY WITH CHOICES FOR ENUM DATUM.
    my @options = ("AND", "OR", "XOR");

    # CREATE ENUMDM WITH "OR" OPTION SELECTED
    perlutil::add_datum_enum (2, 1, "CombineMode\nCombineMode:", \@options, 1);
}

sub Run
{
    $string = "          "; # long enough to hold any strings it needs to

    # GET INDEX OF CURRENT ENUMDM SELECTION.
    $func = perlutil::get_datum_enum (0, 0); # returns 1, which is the index currently selected

    # SET INDEX OF ENUMDM TO SELECT  "XOR"
    perlutil::set_datum_enum (0, 0, 2);

    # SETS $string TO THE OPTION CURRENTLY SELECTED FOR THIS ENUMDM
    # bOK is set to TRUE and $string is set to "XOR"
    $EnumDm = perlutil::get_datum(0, 0);
    $bOK = EnumDm::GetValueByString ($EnumDm, $string);

    # SETS THE CURRENT SELECTION FOR THIS ENUMDM TO "AND"
    $string = "AND";
    $EnumDm = perlutil::get_datum(0, 0);
    EnumDm::SetValueByString ($EnumDm, $string);

    # THE CURRENT SELECTION FOR THIS ENUMDM IS UNCHANGED SINCE "NOT" is not a valid selection
    $string = "NOT";
    $EnumDm = perlutil::get_datum(0, 0);
    EnumDm::SetValueByString ($EnumDm, $string);
}

return 1;

```

References

Read Digital I/O

The read Digital I/O reads a specified digital input, physical or virtual, and outputs its value.

Other Steps Used

Custom Step — A Custom Step must be inserted into a Job before a read Digital I/O can be selected.

Custom Vision Tool — A Custom Vision Tool must be inserted into a Job before a read Digital I/O can be selected.

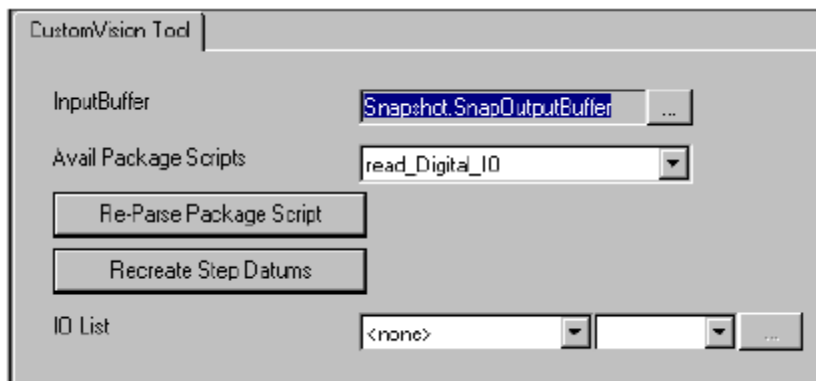
Theory of Operation

An input datum specifies the digital input to be read. The output from this step is the value of the specified digital input.

Description

The CustomVision Tool properties page, pertaining to read Digital I/O, is shown in Figure 6–1.

FIGURE 6–1. CustomVision Tool Properties Page — Read Digital I/O



Settings

- **Avail Package Scripts** — Allows you to select the appropriate Perl script package. The script package used to implement the functionality of this step is `read_Digital_IO`.
- **InputBuffer** — Allows selection of the source of the input buffer datum.
- **IO List** — This datum contains two drop-down lists for configuring I/O. The left list selects the type of I/O (physical, virtual, analog, etc.). The right list selects the I/O point.
- **Recreate Step Datums** — Causes the input and output datum lists in the step to be recreated. Click this button only when a datum is added, removed or changed in the script. If the script is changed, but no input or output datums are changed, this button does not need to be clicked.
- **Re-Parse Package Script** — Causes the script to be parsed when clicked. Whenever any changes to the script are made, click this button to ensure these changes take effect.

Training

None.

Results

Results are determined by the package script selected. The datums in the step can be set through function calls in the script (`perlutil::set_datum_*`) to expose these results outside of this step.

Write Digital I/O

The write Digital I/O writes a specified digital output (physical or virtual) with a specified value.

Other Steps Used

Custom Step — A Custom Step must be inserted into a Job before a write Digital I/O can be selected.

Custom Vision Tool — A Custom Vision Tool must be inserted into a Job before a write Digital I/O can be selected.

Theory of Operation

The selected digital output bit is set with the specified value. An IO List datum is supplied in order to specify:

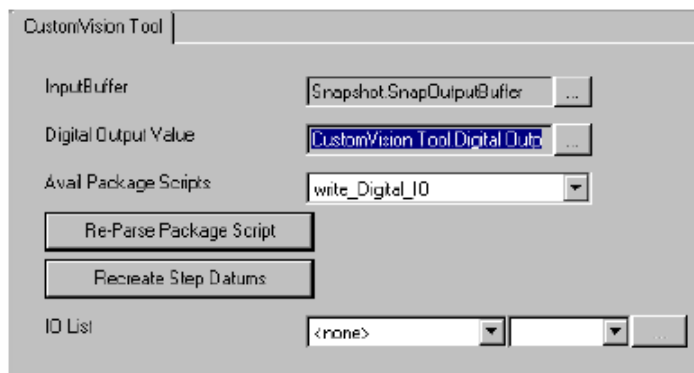
- Type of I/O (physical, virtual, analog, etc.).
- Digital I/O point to write to.

At runtime, write Digital I/O writes the value of the datum referenced by Digital Output Value to the specified I/O point.

Description

The CustomVision Tool properties page, pertaining to write Digital I/O, is shown in Figure 6–2.

FIGURE 6–2. CustomVision Tool Properties Page — Write Digital I/O



Settings

- Avail Package Scripts — Allows you to select the appropriate Perl script package. The script package used to implement the functionality of this step is write_Digital_IO.
- Digital Output Value — Specifies the value to write to the digital output point. This is an input datum that must be connected to another datum in the Job.
- InputBuffer — Allows selection of the source of the input buffer datum.

- **IO List** — This datum contains two drop-down lists for configuring I/O. The left list selects the type of I/O (physical, virtual, analog, etc.). The right list selects the I/O point.
- **Recreate Step Datums** — Causes the input and output datum lists in the step to be recreated. Click this button only when a datum is added, removed or changed in the script. If the script is changed, but no input or output datums are changed, this button does not need to be clicked.
- **Re-Parse Package Script** — Causes the script to be parsed when clicked. Whenever any changes to the script are made, click this button to ensure these changes take effect.

Training

None.

Results

Not Applicable.

Helpful Programming Information

Multiple Instances of Scripts and Local Variables

The perl scripts that control the processing in Custom Steps/Tools are compiled into packages. The package provides a namespace for the variables within the script, and all variables are global within the package. This means that multiple instances of the same Custom Step/Tool (for example, CircleFind) will have the same variables used in multiple steps. This may result in unpredictable behavior, especially for variables that are set up in one subroutine of a package and then used in another subroutine. If a variable is changed in one Custom Step/Tool, then the variable will be changed for all Custom Step/Tools that use this script/package.

Microscan has provided functions to make variables local to a package, and therefore unique, even among multiple instances of the same package. **Localize::makelocal(\$variable)** creates a name for **\$variable** that is unique to a Custom Step/Tool even when the variable is used in a common package.

Anytime multiple instances of a script/package are used, the **makelocal(...)** function must be used on variables that need to be unique for each instance of the script. In the following sample script, a vision agent is created in the PreRun of a script and used later in the Run. Without the makelocal() function, if multiple Custom Steps/Tools using this script were present in an inspection, then the same \$blobagent variable would be used by all of them. This would result in a blob agent that is configured for the last Custom Step/Tool that did a PreRun. All steps, however, would use this blob agent. The same is true for the \$blobresult variable.

Localize::makelocal() ensures that these variables are unique for each Custom Step/Tool.

Note: An alternative method that is used by some scripts provided by Microscan uses an array to hold variables that need to be unique for each Custom Step/Tool. The array is indexed by an instance variable that is provided as an integer datum in the UI of the step. The makelocal() function is more flexible; however, either will work.

```
:
:
use Localize;
:
:

package blobstuff;

perlutil::register("blobstuff");

# THESE VARIABLES MUST BE UNIQUE FOR EACH INSTANCE OF THIS SCRIPT/PACKAGE
Localize::makelocal(\$blobresult);
Localize::makelocal(\$blobagent);

sub Init      { }
sub Cleanup   { }
sub Start     { }
sub Stop      { }
sub Update    { }

sub Apply
{
    perlutil::clear_datum_lists;
    perlutil::add_datum_int (2, 1, "LowThreshold", 0);
    # input 0: low threshold for blob
    perlutil::add_datum_int (2, 1, "HighThreshold", 128);
    # input 1: high threshold for blob
}

sub PreRun
{
    # PARAMETERS FOR BLOB
```

```

my $lowThresh = perlutil::get_datum_int (0, 0);
my $highThresh = perlutil::get_datum_int (0, 1);

# GET INPUT BUFFER AND ROI
my $mybuf = perlutil::get_input_buf();
my $l = perlutil::get_input_left();
my $t = perlutil::get_input_top();
my $r = perlutil::get_input_right();
my $b = perlutil::get_input_bottom();

# CREATE A BLOB RESULT.
$blobresult = BlobResult::BlobResult(0.25, 0.25);

# CREATE AND CONFIGURE A BLOB AGENT.
$blobagent = BlobFact::BlobFact($mybuf, $l, $t, $r, $b, $blobresult, $lowThresh,
$highThresh);
BlobAgent::SetMinBlob($blobagent,10);
BlobAgent::SetMaxBlob($blobagent,1000000);
BlobAgent::SetProcessSwitches($blobagent,7);
}

sub Run
{
    # UPDATE INPUT BUFFER AND INPUT POINT
    $inbuf = perlutil::get_input_buf();
    $l = perlutil::get_input_left();
    $t = perlutil::get_input_top();
    $inpoint = Point::PointInt($l, $t);

    # CLEANUP PREVIOUS RESULTS
    BlobResult::CleanUp($blobresult);

    # RUN BLOB AGENT
    BlobAgent::SetSrcBuf($blobagent, $inbuf);
    BlobAgent::SetSrcPt($blobagent, $inpoint);
    BlobAgent::SetResult($blobagent, $blobresult);
    $status = BlobAgent::Go($blobagent);

    # IF BLOB AGENT RUN FAILED FOR SOME REASON, CLEANUP BLOB RESULTS GENERATED
    if ($status) { BlobResult::CleanUp($blobresult); }

    Point::Point_Delete($inpoint);
}

sub PostRun
{
    # DELETE RESOURCES CREATED IN PRE-RUN
    if ($blobresult) {
        BlobResult::BlobResult_Delete($blobresult);
        $blobresult = 0;
    }
    if ($blobagent) {
        BlobAgent::BlobAgent_Delete($blobagent);
        $blobagent = 0;
    }
}

sub Draw { }

```

Memory Leaks — Creating and Deleting Objects

To prevent memory leaks, any objects created in a script must also be deleted. Any call of the form `package::package` will allocate memory for the specified object. A matching call must be made to `package::package_Delete` to deallocate the appropriate memory and avoid memory leaks. For example, creating and deleting a rectangle object would be:

```
$theRect = Rect::Rect($l, $t, $r, $b);
:      :      :
Rect::Rect_Delete ($theRect);
```

Most agents are created through calls to a factory package. For example, creating and then destroying a sobel agent would be as follows.

```
$pSobelAgent = SobelFact::SobelFact($inbuf, $outbuf,
$l, $t, $r, $b, $sobelFunc, $sobelDiv, $thresh);
:      :      :

SobelAgent::SobelAgent_Delete($pSobelAgent);
```

Other functions may allocate memory. For example:

```
$pArray = PtListDm::GetPointListByArray ($pPtListDm);#
allocates memory
:      :      :

PtListDm::FreePointListByArray($pArray);# deallocates
memory
```

Consult the detailed function descriptions for more information about a particular function, including the appropriate deallocation function.

Glossary of Terms

- **Factory** — A class of objects used to create vision agent objects.
- **Input Search Area** — The area contained within the input ROI of a Custom Vision Tool.
- **Master Script File** — This file is `perlscr` and is located in the `perlmod` directory of the current Visionscape installation. This file determines

which script files are parsed by the perl interpreter. Each of the script files contains one or more functional packages that will then be selectable from a Custom Step/Tool.

- **Monster** — The ASIC available on Visionscape GigE Cameras is referred to as the vision monster.
- **Package** — Data and functions bundled together form a package. For clarity consider two types of packages functional and support. A functional package is basically the processing associated with a Custom Step/Tool. Support packages provide access to Microscan functions. CircleFind is an example of a functional package. Many calls to support packages are made within the CircleFind package to implement the functional package. A functional package can be thought of as a whole puzzle, and the support packages are the pieces that make up the puzzle.

A call to a support package function is in the form:

```
packageName::functionName()
```

For example, perlutil::get_input_buf().

- **Part** — A coordinate system.
- **Script** — A file, written in perl, which determines the functionality of a Custom Step/Tool. The subroutines Init, Cleanup, Apply, Update, PreRun, Run, PostRun, Draw, Start, Stop, and optionally Train are supported in each script. The scripts consist of calls to native perl code and calls to Microscan supplied perl code which allows the user access to Microscan's vision library. Script files have a .pm extension and are located in the perlmod directory of the current Visionscape installation. The script file must be included in the master script file in order for the functional package in that script to be available to a Custom Step/Tool.

Open Source Software Used in Visionscape

The product contains, among other things, Perl 5.6.2 Open Source Software, licensed under an Open Source Software License and developed by third parties. This Perl 5.6.2 Open Source Software files are protected by copyright. Your rights to use the Open Source Software beyond the mere execution of Microscan's program, is governed by the relevant Open Source Software license conditions.

Your compliance with those license conditions will entitle you to use the Open Source Software as foreseen in the relevant license. In the event of conflicts between Microscan license conditions and the Open Source Software license conditions, the Open Source Software conditions shall prevail with respect to the Open Source Software portions of the software. The Open Source Software is licensed royalty-free (i.e., no fees are charged for exercising the licensed rights, whereas fees may be charged for reimbursement of costs incurred by Microscan).

.A list of the Open Source Software programs contained in this Product and the Open Source Software licenses are available in this Manual. Furthermore the license conditions can be found at the following internet websites:

<http://www.perl.com/>

Warranty Regarding Further Use of the Open Source Software

Microscan provides no warranty for the Open Source Software programs contained in this device, if such programs are used in any manner other than the program execution intended by Microscan. The licenses listed below define the warranty, if any, from the authors or licensors of the Open Source Software. Microscan specifically disclaims any warranties for defects caused by altering any Open Source Software program or the product's configuration. You have no warranty claims against Microscan in the event that the Open Source Software infringes the intellectual property rights of a third party.

Technical support, if any, will only be provided for unmodified software.

Open Source Software Used

TABLE A-1.

Open Source Software Component	License
Perl5.6.2	Artistic License

Open Source Software Licenses

Preamble

The intent of this document is to state the conditions under which a Package may be copied, such that the Copyright Holder maintains some semblance of artistic control over the development of the package, while giving the users of the package the right to use and distribute the Package in a more-or-less customary fashion, plus the right to make reasonable modifications.

Definitions

"Package" refers to the collection of files distributed by the Copyright Holder, and derivatives of that collection of files created through textual modification.

“Standard Version” refers to such a Package if it has not been modified, or has been modified in accordance with the wishes of the Copyright Holder as specified below.

“Copyright Holder” is whoever is named in the copyright or copyrights for the package.

“You” is you, if you're thinking about copying or distributing this Package.

“Reasonable copying fee” is whatever you can justify on the basis of media cost, duplication charges, time of people involved, and so on. (You will not be required to justify it to the Copyright Holder, but only to the computing community at large as a market that must bear the fee.)

“Freely Available” means that no fee is charged for the item itself, though there may be fees involved in handling the item. It also means that recipients of the item may redistribute it under the same conditions they received it.

1. You may make and give away verbatim copies of the source form of the Standard Version of this Package without restriction, provided that you duplicate all of the original copyright notices and associated disclaimers.
2. You may apply bug fixes, portability fixes and other modifications derived from the Public Domain or from the Copyright Holder. A Package modified in such a way shall still be considered the Standard Version.
3. You may otherwise modify your copy of this Package in any way, provided that you insert a prominent notice in each changed file stating how and when you changed that file, and provided that you do at least ONE of the following:
 - a. place your modifications in the Public Domain or otherwise make them Freely Available, such as by posting said modifications to Assent or an equivalent medium, or placing the modifications on a major archive site such as uunet.uu.net, or by allowing the Copyright Holder to include your modifications in the Standard Version of the Package.
 - b. use the modified Package only within your corporation or organization.

- c. rename any non-standard executables so the names do not conflict with standard executables, which must also be provided, and provide a separate manual page for each non-standard executable that clearly documents how it differs from the Standard Version.
 - d. make other distribution arrangements with the Copyright Holder.
- 4. You may distribute the programs of this Package in object code or executable form, provided that you do at least ONE of the following:
 - a. distribute a Standard Version of the executables and library files, together with instructions (in the manual page or equivalent) on where to get the Standard Version.
 - b. accompany the distribution with the machine-readable source of the Package with your modifications.
 - c. give non-standard executables non-standard names, and clearly document the differences in manual pages (or equivalent), together with instructions on where to get the Standard Version.
 - d. make other distribution arrangements with the Copyright Holder.
- 5. You may charge a reasonable copying fee for any distribution of this Package. You may charge any fee you choose for support of this Package. You may not charge a fee for this Package itself. However, you may distribute this Package in aggregate with other (possibly commercial) programs as part of a larger (possibly commercial) software distribution provided that you do not advertise this Package as a product of your own. You may embed this Package's interpreter within an executable of yours (by linking); this shall be construed as a mere form of aggregation, provided that the complete Standard Version of the interpreter is so embedded.
- 6. The scripts and library files supplied as input to or produced as output from the programs of this Package do not automatically fall under the copyright of this Package, but belong to whomever generated them, and may be sold commercially, and may be aggregated with this Package. If such scripts or library files are aggregated with this Package via the so-called "undump" or "unexec" methods of producing a binary executable image, then distribution of such an image shall neither be construed as a distribution of this Package nor shall it fall under the restrictions of Paragraphs 3 and 4, provided that

you do not represent such an executable image as a Standard Version of this Package.

7. C subroutines (or comparably compiled subroutines in other languages) supplied by you and linked into this Package in order to emulate subroutines and variables of the language defined by this Package shall not be considered part of this Package, but are the equivalent of input as in Paragraph 6, provided these subroutines do not change the language in any way that would cause it to fail the regression tests for the language.
8. Aggregation of this Package with a commercial distribution is always permitted provided that the use of this Package is embedded; that is, when no overt attempt is made to make this Package's interfaces visible to the end user of the commercial distribution. Such use shall not be construed as a distribution of this Package.
9. The name of the Copyright Holder may not be used to endorse or promote products derived from this software without specific prior written permission.
10. THIS PACKAGE IS PROVIDED "AS IS" AND WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

