

Introduction to

Sherlock 7



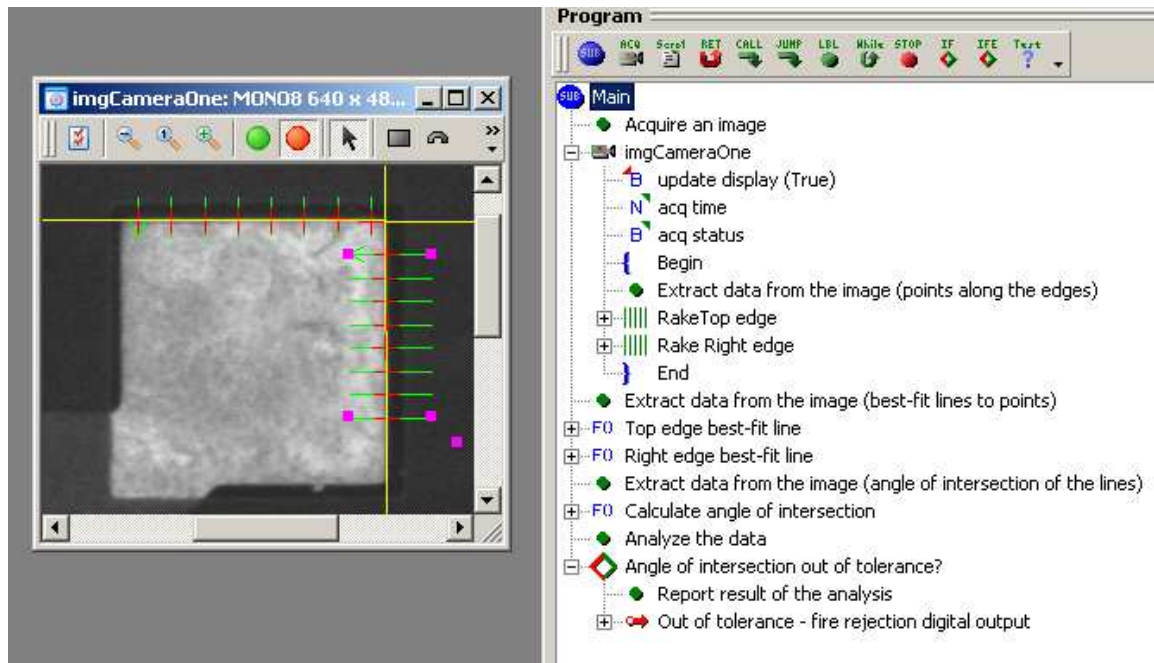
Introduction

Welcome to Sherlock 7. Sherlock is a graphical IDE (Integrated Development Environment) in which you design, test, debug and deploy machine vision applications.

A typical machine vision application is composed of four steps:

1. Acquire an image
2. Extract data from the image
3. Analyze the data
4. Report on or make decisions based on the analysis (write results to a file, trigger a reject mechanism, set a PLC register value...)

Each of the four steps of an application is achieved by selecting program elements from within Sherlock's graphical interface, adding them to the program, and setting their execution parameters. **You do not write code to create a machine vision application with Sherlock.**



A simple Sherlock program

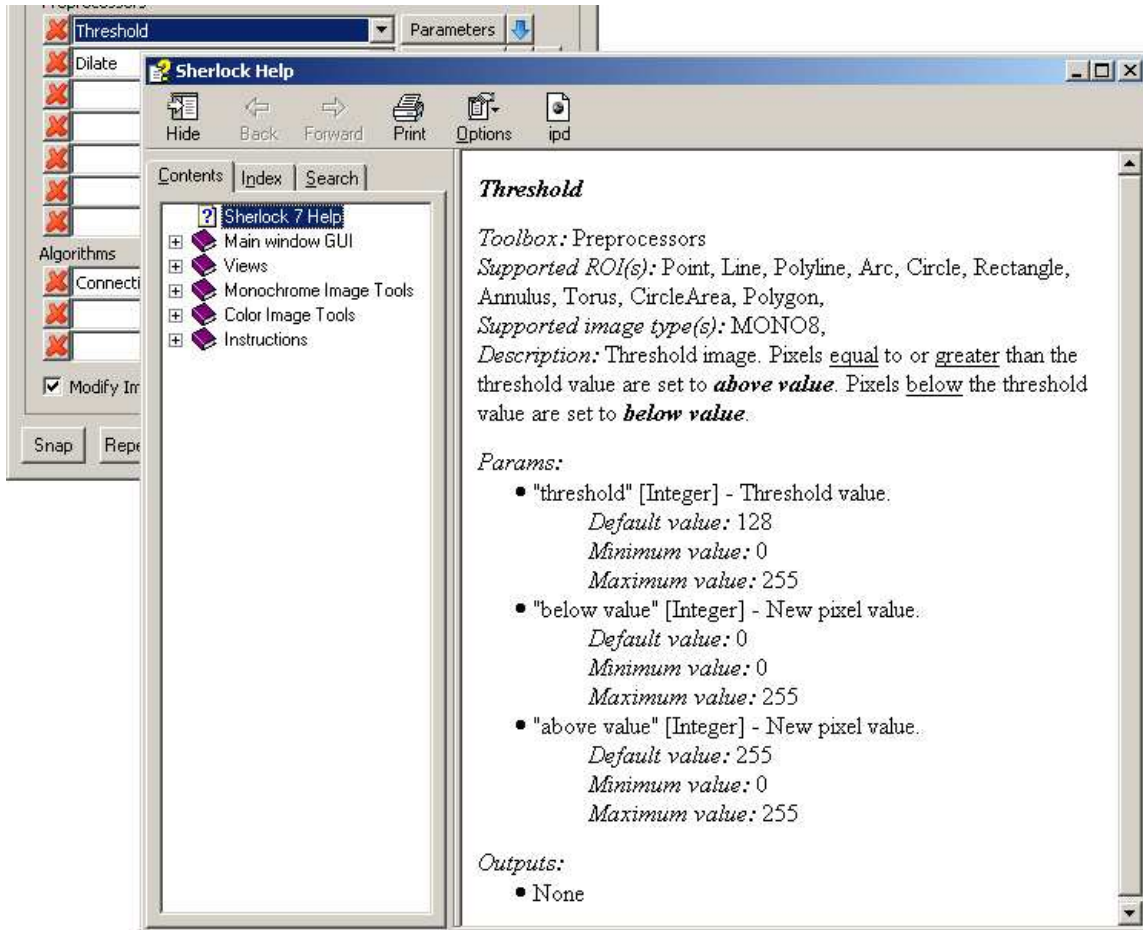
A complete Sherlock application is called an **investigation**. An investigation is saved as a file with an ".ivs" extension.

This manual explains how to use Sherlock to create an investigation, from acquiring an image, through communicating the results of the analysis. Sherlock is a very rich environment with hundreds of programming elements to choose from, so not every aspect of every element can be explained; but by the end of the manual you will know where to find the elements you need, how to add them to the investigation, and how to customize the elements to fit your needs.

Online help

For information on programming elements not explicitly covered in this manual, or for more detailed information on the elements that are covered, use online help. Help is available through

Sherlock's main menu (**Help→Help Topics**), and by hitting the **F1** key when a programming element is selected.

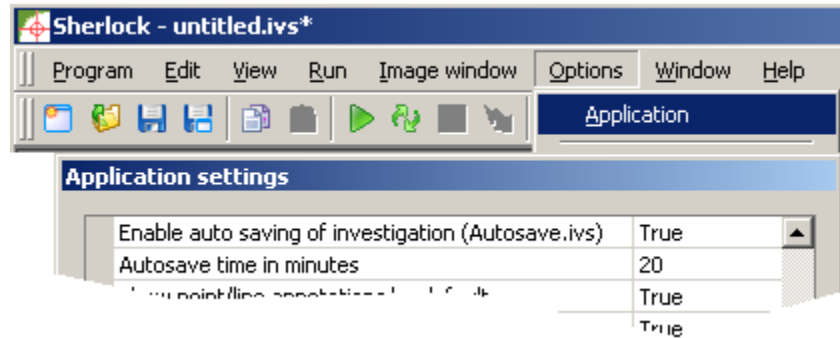


When bad things happen to good developers

As with any computer application, there is always the possibility that an operating system malfunction, a power outage, or some other event beyond your control will cause your computer to hang, reboot, or otherwise cause you to lose your work. Words to the wise:

Save early, save often

Sherlock has an autosave feature that by default saves the open investigation to the file <Sherlock>\Programs\Autosave.ivs every 20 minutes. To change the time between autosaves, open **Options→Application** from Sherlock's main menu. You can also disable the autosave feature, but why in your right mind would you?



The autosave feature is not active while the investigation is executing.

Some image basics

Here are a few things you'll need to know about how images are stored and referenced in Sherlock.

Monochrome pixel values

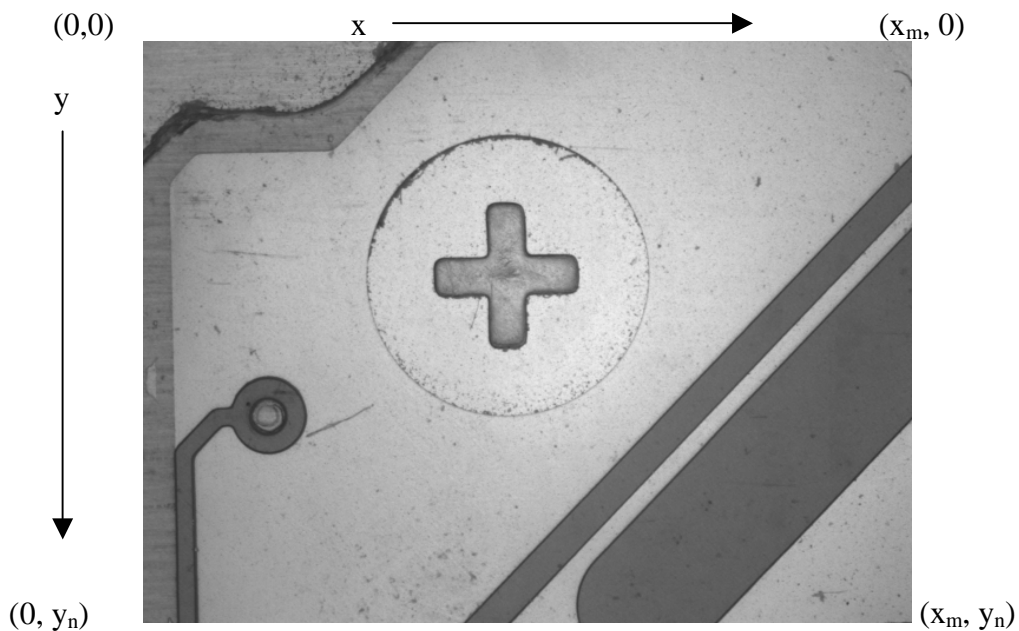
Most machine vision applications work with 8-bit monochrome images. In an 8-bit monochrome image, 0 = black and 255 = white.



(The representation of color images is discussed elsewhere.)

Image origin

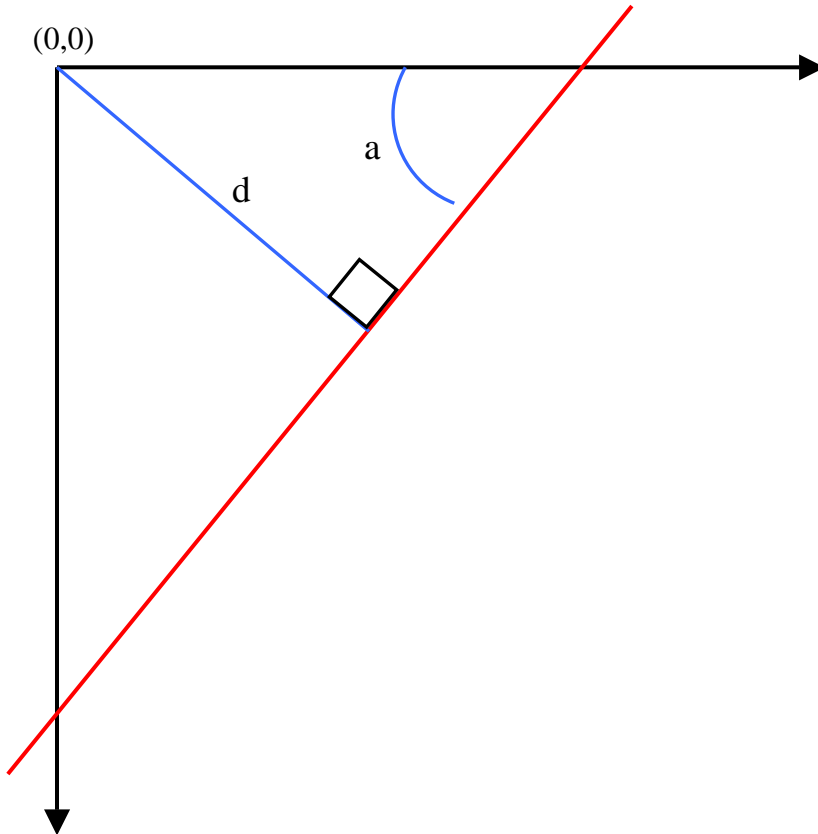
The origin (0,0) is at the upper-left corner of the image.



Definition of a line

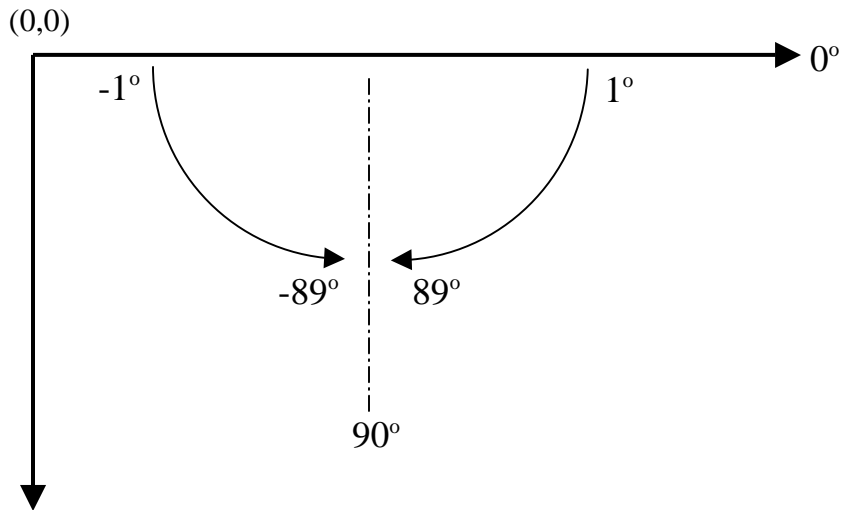
A **line** is one of the five data types in Sherlock; the other types are the **number**, **point** (x,y coordinate), **Boolean** (True/False), and character **string**. All of these data types are discussed later in this manual. Because the definition of a line is non-intuitive, it is presented here for your enlightenment.

In Sherlock, a line is defined by **a**, the angle of the line relative to the x-axis, and **d**, the shortest distance from the image origin to the line (the perpendicular). A line has no start or end point; it extends infinitely in both directions.

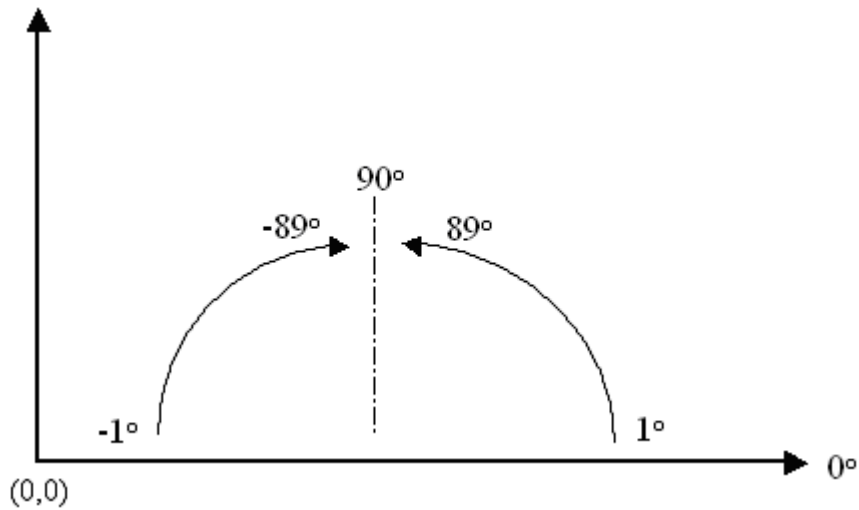


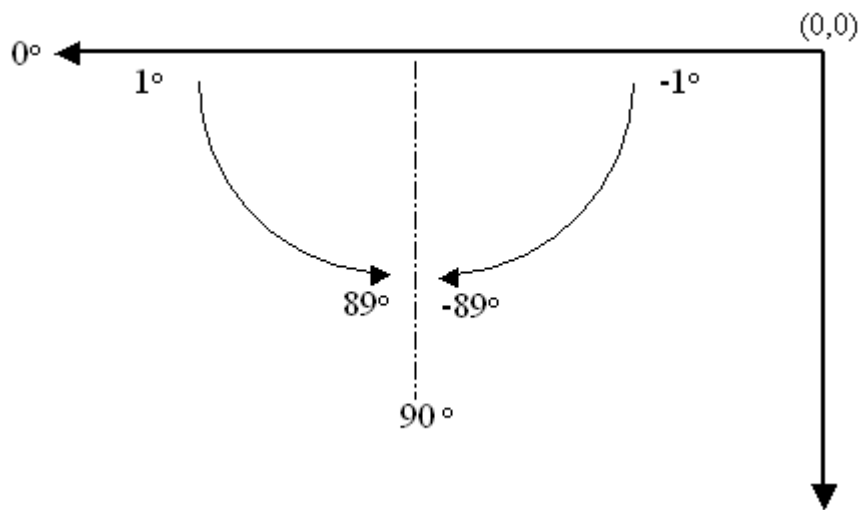
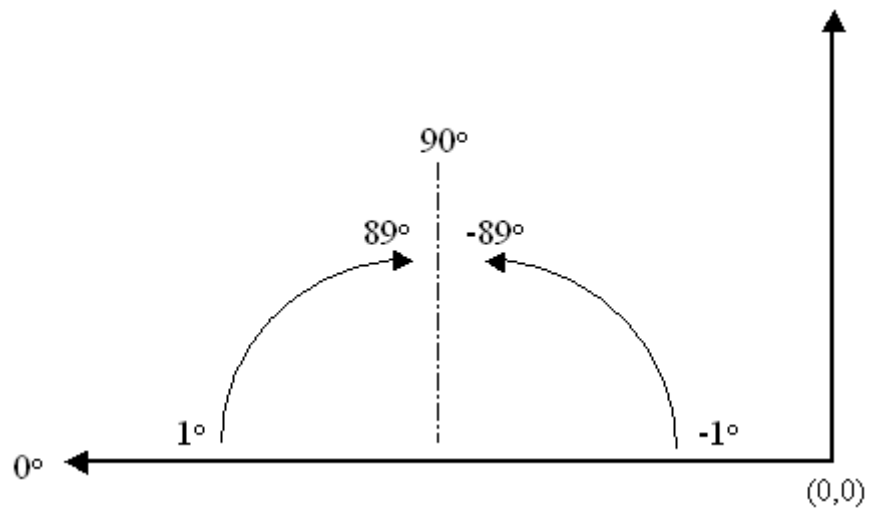
The location of the origin and the direction of the axes affect how angles are measured.

The default origin is at the upper-left corner of the image.



Changing the location of the origin and the direction of the axes (see the chapter **Calibration**) affects how angles are measured.





Interface

Investigation windows

A Sherlock investigation consists of several windows, the contents of most of which you control. Some of the windows are present in every investigation; some are present by your choice.

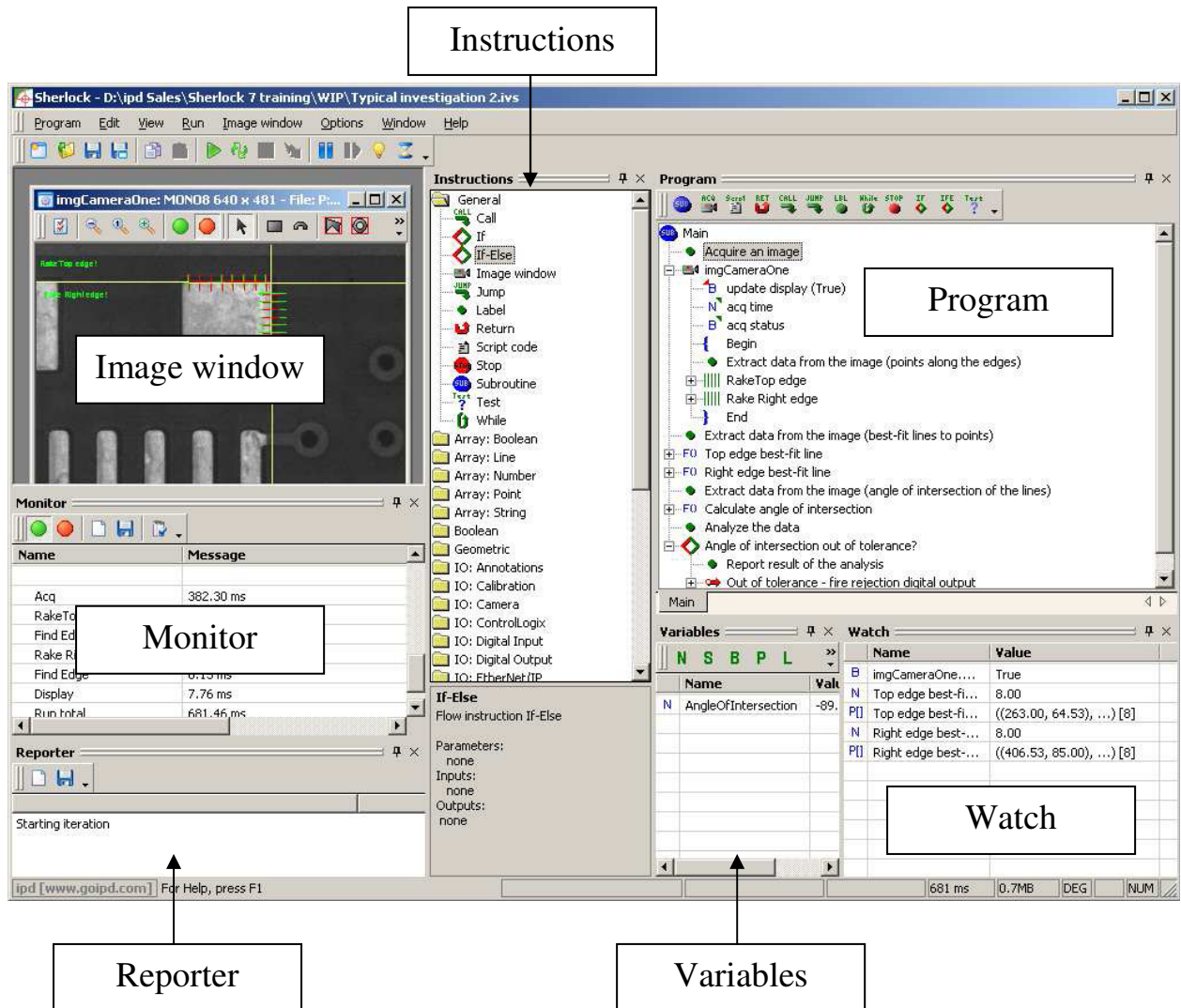


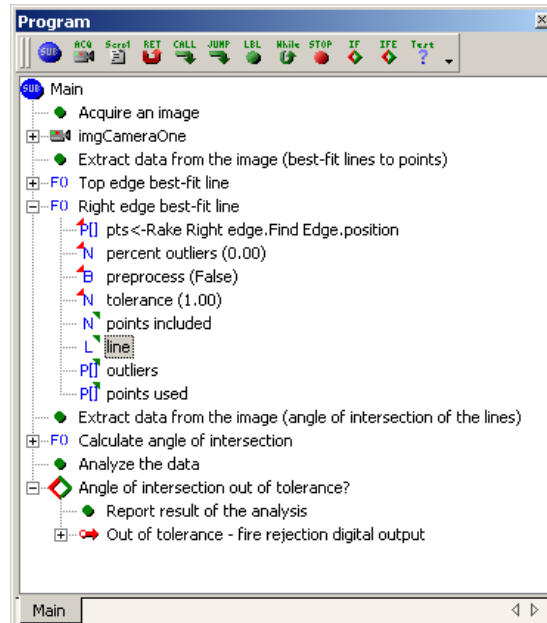
Image Window

An image window contains an image and the tools that extract data from it.



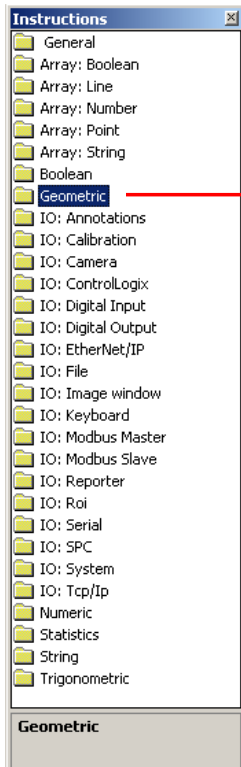
Program

The program is the sequence of steps you define to perform the four steps of your machine vision application – acquire an image; extract data from the image; analyze the data; and report on or make decisions based on the results of the analysis.

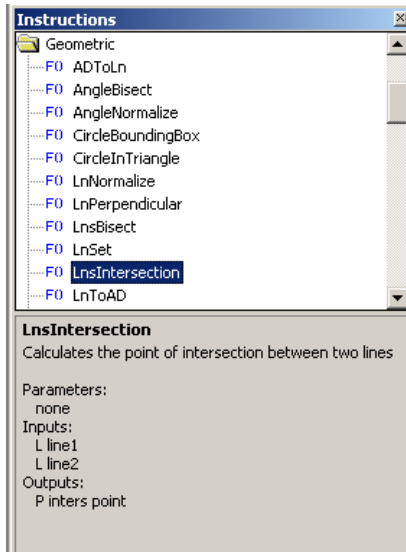


Instructions

Many of the programming elements you add to the program are available from the Instructions window.

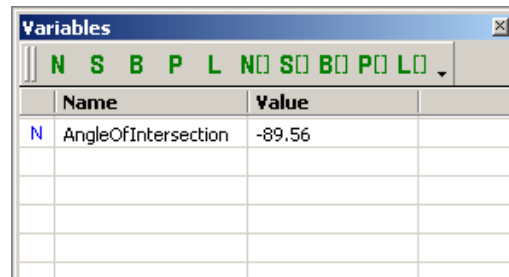


The instructions are grouped into functional folders, each of which contains many instructions.



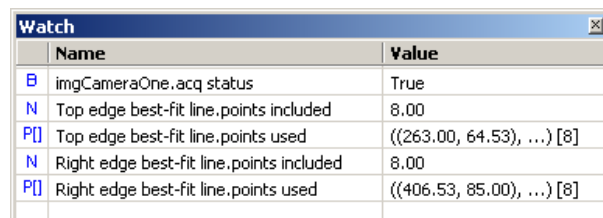
Variables

Variables are placeholders for data. You create and display variables in the variables window.



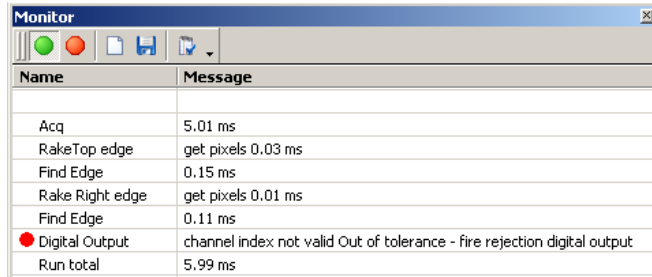
Watch

The watch window can display, at your discretion, the output of various programming elements.



Monitor

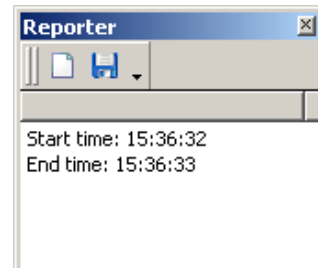
The monitor displays runtime information, in particular how long the various elements of the program took to execute. It also displays errors and warnings.

The Monitor window has a title bar with a close button. Below the title bar is a toolbar with icons for a list, a green circle, a red circle, a document, a save icon, and a dropdown arrow. The main area is a table with two columns: 'Name' and 'Message'.

Name	Message
Acq	5.01 ms
RakeTop edge	get pixels 0.03 ms
Find Edge	0.15 ms
Rake Right edge	get pixels 0.01 ms
Find Edge	0.11 ms
Digital Output	channel index not valid Out of tolerance - fire rejection digital output
Run total	5.99 ms

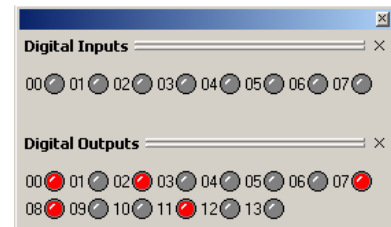
Reporter

The reporter is a window to which you can write text during program execution.



Digital IO

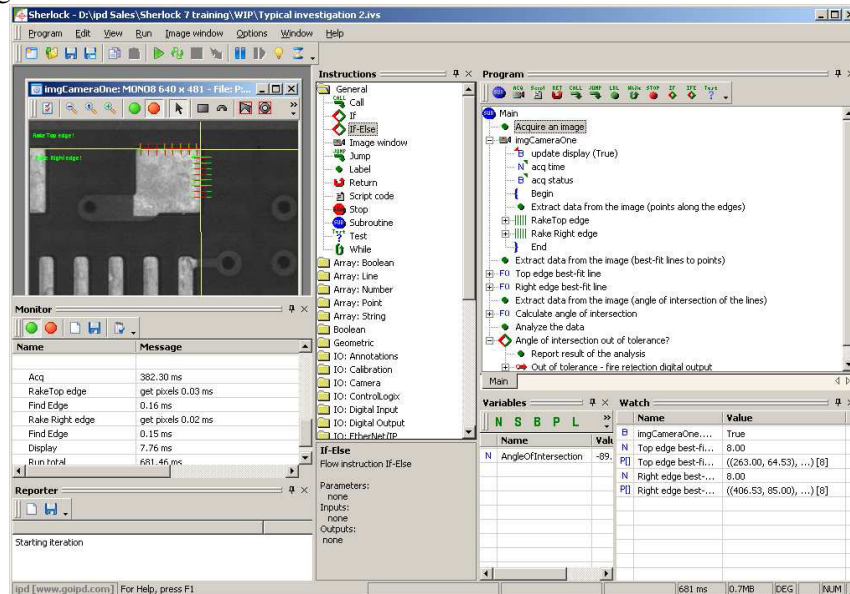
If your system has digital inputs and outputs that Sherlock has access to, the Digital Inputs and Digital Outputs windows shows their states.



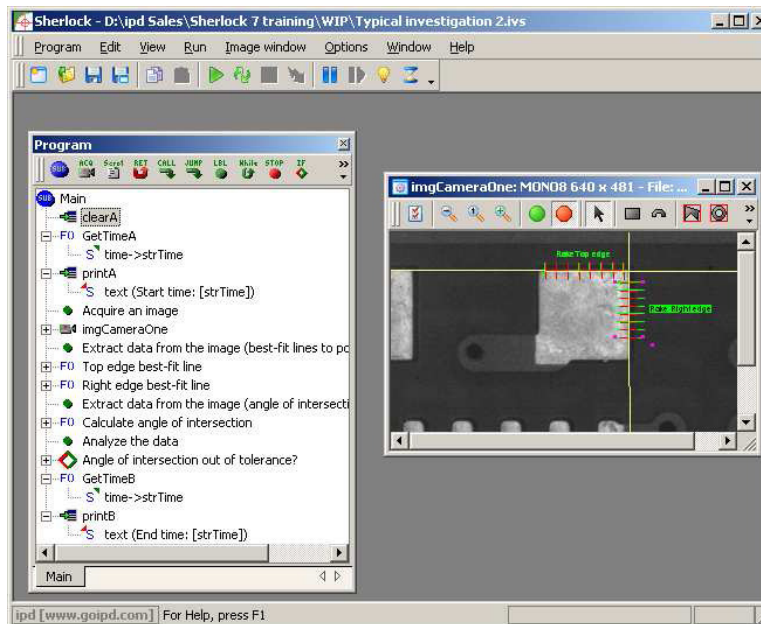
Windows layout

You can arrange the windows in an investigation any way you want to. They can be docked, free-floating, tabbed, and “pinned.” You can even choose not to display some or any of the windows at all.

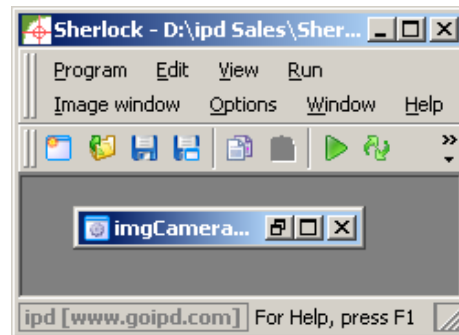
This investigation...



...is the same as this one...



...and this one!



The status bar

The status bar at the bottom of the main Sherlock window displays information about various aspects of the investigation.



Left to right:

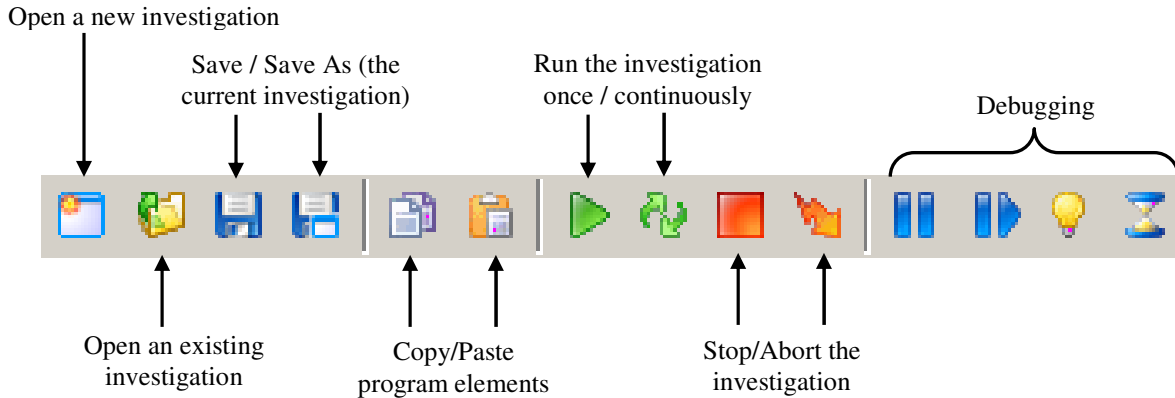
- If the image window has been calibrated, the position of the cursor in calibrated units.
- The position of the cursor in pixels
- The value of the pixel at the cursor. This will be a single value for a grayscale image (for example, **pix:254**), or the red, green and blue components for a color image (for example, **pix:132,045,025** for a dark maroon pixel.)



- Execution time of the last iteration of the investigation
- Memory usage
- Angle measurement units (**DEG**rees or **RAD**ians)
- Cap lock on (**CAP**) or off (blank)
- Num lock on (**NUM**) or off (blank)
- Scroll lock on (**SCRL**) or off (blank)

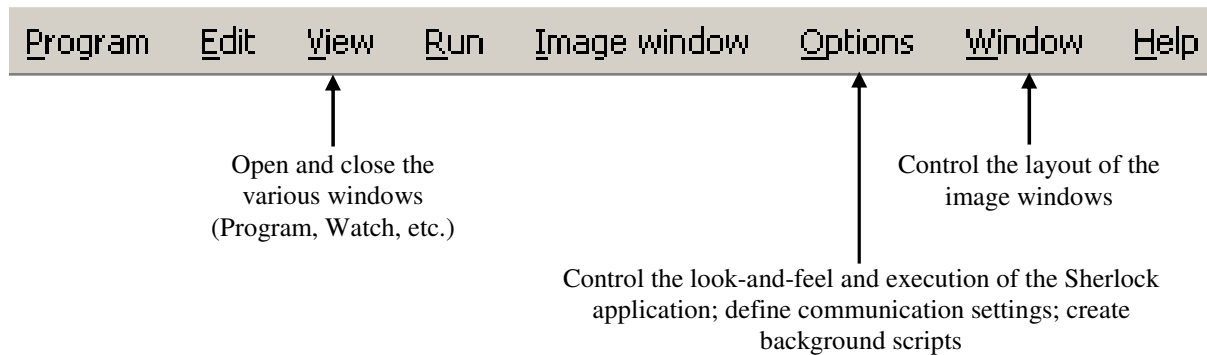
The main toolbar

The buttons on Sherlock's main toolbar are used to open, save, edit, execute, and debug an investigation.



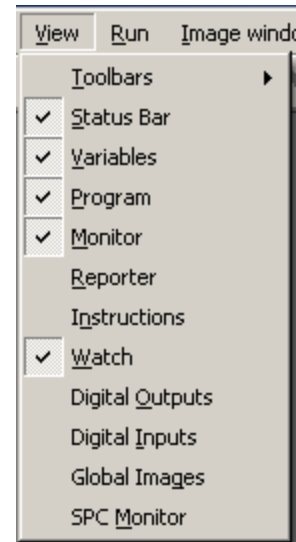
The main menu

The **File**, **Edit** and **Run** items echo the main toolbar buttons that open and save investigations, copy program elements, and run the investigation. The **Image window** items echo image window toolbar buttons (zoom in/out, live image, insert ROI, load/save image).



Check or uncheck a window on the **View** menu to display or close it. Closing a window does not destroy its contents.

In this example, the status bar, variables, program, monitor, and watch windows are displayed. All other windows are closed.



The various items in the **Options** menu are covered elsewhere.



You can run only one instance of Sherlock at a time, and Sherlock can open only one investigation at a time.

Image Window

An image window contains an image to be processed, and the tools to process it. An investigation has at least one image window, and it may have many; there is no limit to the number of image windows. An image window displays a single image at a time.

To add an image window to the investigation, click the **Create image window instruction** button on the program window's toolbar.

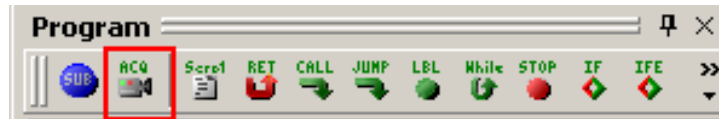


Image window options

Double left click on an image window to display its **Options** dialog; or click the Options button



on the image window's toolbar.

Image window name

Every image window is assigned a name when it is created: **imgA**, **imgB**, **imgC**, etc.

You can rename an image window (**Top View**, **Camera 0**) in its **Options** dialog, or in the program window (left-click once on the name, wait briefly, then left-click again). All names must be unique within an investigation.

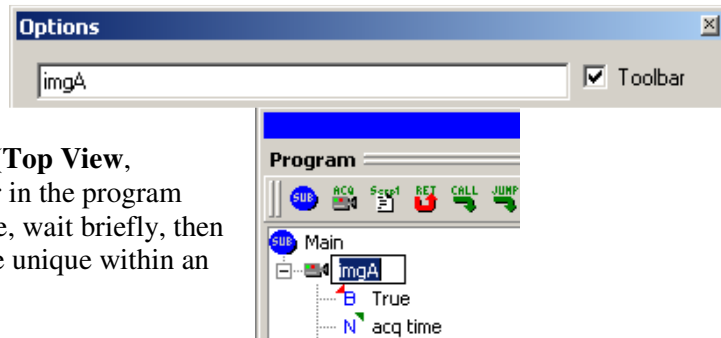


Image Source

An image window can get its image from one of several sources: a camera, another image window, a combination of image windows, a single image file, a sequence of image files, or a reading from an algorithm. To set an image window's source, click the appropriate radio button.

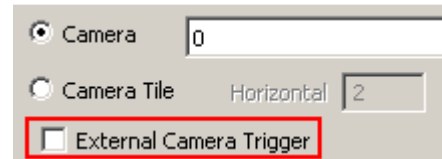
● Camera

In most investigations, at least one image window acquires its



image from a camera. The drop-down list next to **Camera** shows the indices of the available cameras; you select the camera from which you want to acquire images for this image window. If your system has more than one camera, the investigation will probably have an image window for each camera.

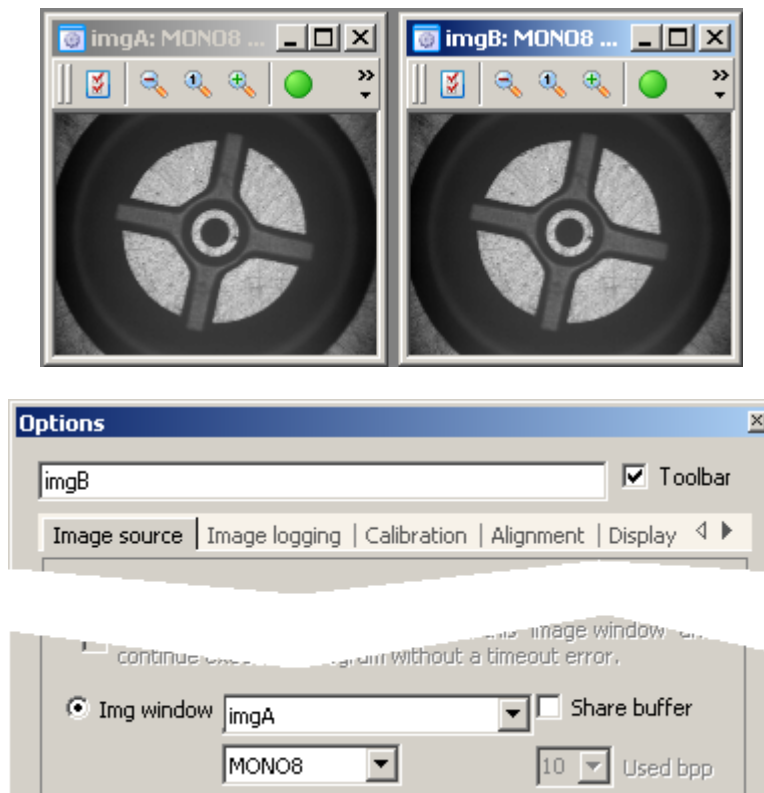
Acquisition is software-triggered (free-running) unless **External Trigger** is checked. Software triggering acquires the next full frame from the camera as soon as the image window is reached in the program. External triggering waits for an input signal to initiate acquisition. Most machine vision applications use a part-in-place sensor or other hardware to determine when the object to be inspected is in front of the camera; when the object is in place, a signal is sent to the hardware to initiate acquisition.



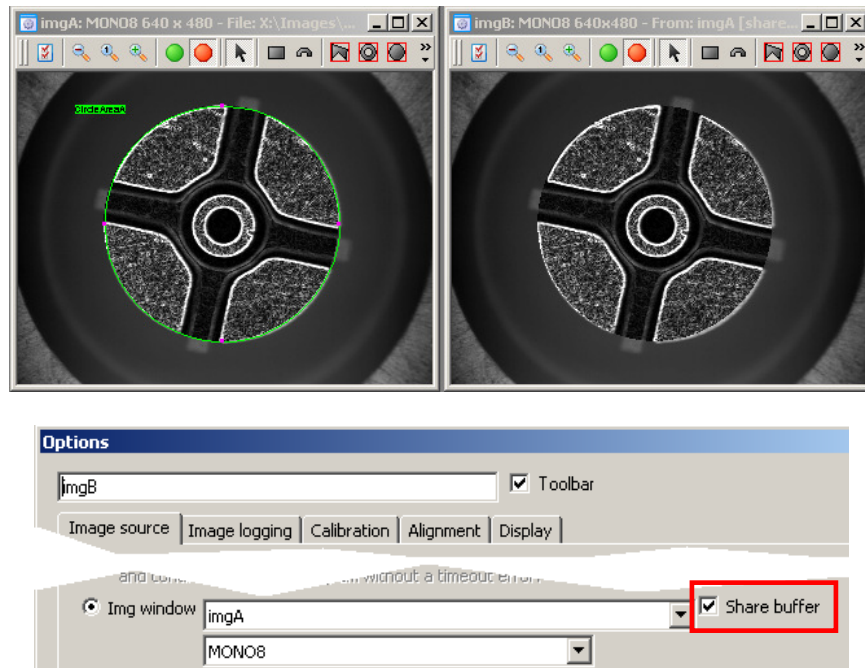
- **Image window**

Sometimes you want to use the same image in more than one image window, for example to apply different processing to the same image acquired from a camera.

In this example, image window **imgB** gets its image from image window **imgA**.

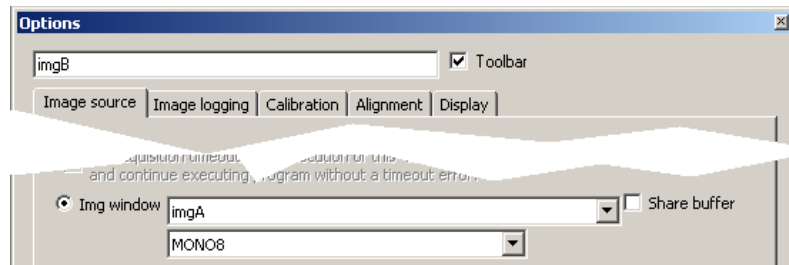
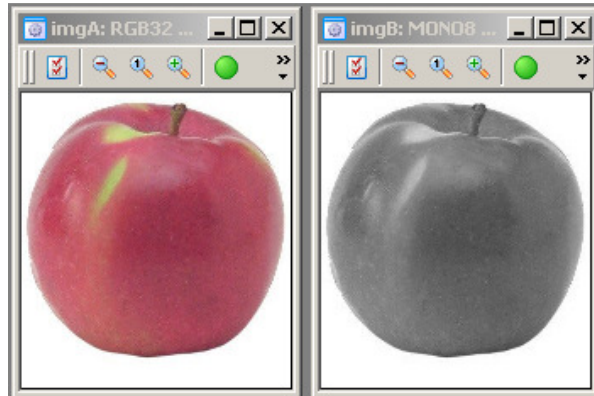


If the destination image window's **Share buffer** box is checked, any modifications made to the source image are reflected in the destination image; if it is not checked, the “raw” image is used as the source. (See the chapter **ROIs, Preprocessors and Algorithms**)

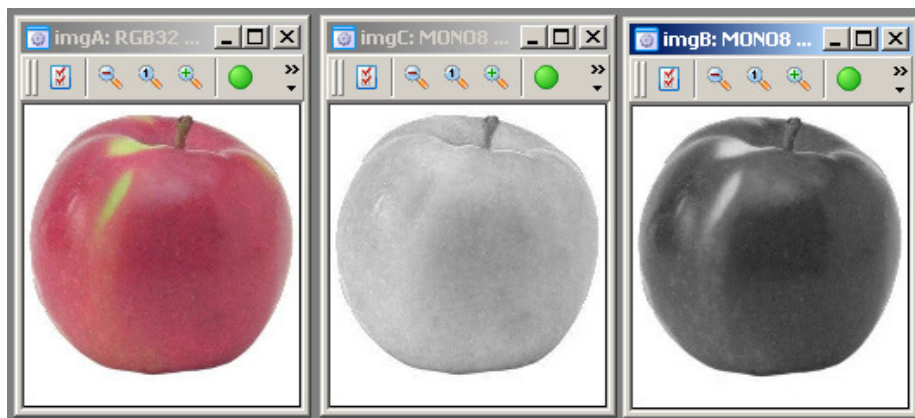


You can set the destination image window's type (monochrome, RGB color, YUV color, etc.) to be the same as the source image window's type, or different.

ImgA contains an RGB color image. **ImgB**'s source is set to **imgA**, and its type is set to MONO8.



If the source image window contains a color image, you can select one of the color planes (R, G, B or Y, U, V) as the source for a monochrome image window. In the monochrome image window, pixels with high values of the color are shown as high grayscale values, and pixels with low values of the color are shown as low grayscale values.



RGB color image

Red plane as monochrome

Green plane as monochrome

• Multiple image windows

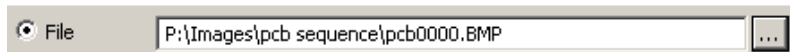
Three monochrome image windows can be combined to make a single color image.



When composing an image from three monochrome image windows, the only choices for the image type are RGB32 , YUV32 and YCBCR32.

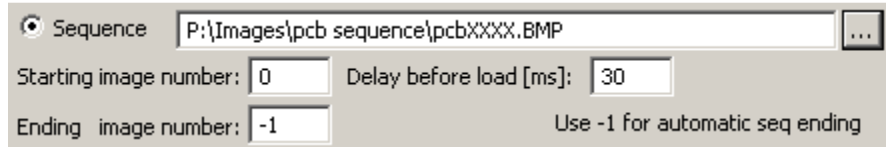
• Image file

It is common to use a static image file as the image source when you are developing an investigation. Sherlock supports image types bmp, tif, jpg, and png.



• Image sequence

A series of image files can be used to simulate camera input.



An image sequence simulates camera acquisition by loading a series of image files into the image window, one at a time. The image files in an image sequence must have names of the format

anythingNNNN.ext

where

anything is a valid root file name – for example, **image**, **LeftView** or **phone_keypad**.

NNNN is a four digit sequence number, 0000, 0001, 0002, 0003, etc. Numbers in the sequence must be contiguous.

ext is a valid image file extension (bmp, tif, jpg, png).

When the investigation runs, the first time the image window executes it loads the first image file in the sequence, according to the number in **Starting image number**. (By convention sequence numbers start at 0000, but they do not have to. To start at a sequence number other than 0, change **Starting image number**.) On each subsequent execution of the image window, the next image file in the sequence is loaded. If **Ending image number** is -1, all the image files in the sequence are loaded, then the first image is loaded again, and so on. If **Ending image number** is set to a positive number, the image file with that sequence number will be the last image loaded from the sequence, even if there are more files in the sequence.

Example

There are image files with the names widget0000.bmp, widget0001.bmp, widget0002.bmp, ..., widget0099.bmp, and widget7777.bmp.

If the image sequence settings are left at their defaults (**Starting image number** = 0, **Ending image number** = -1), the image window will load, in succession, images widget0000.bmp through widget0099.bmp, then start at widget0000.bmp again. Widget7777.bmp will not be loaded, because its sequence number is not contiguous with widget0099.bmp.

If the image sequence settings are changed to **Starting image number** = 23, **Ending image number** = 87, the image window will load, in succession, images widget0023.bmp through widget0087.bmp, then start at widget0023.bmp again.

Delay before load sets the minimum time that will occur between loading images in an image sequence. You can set this to a higher value to get a better look at what's going on in the investigation when you run it.

• Reading

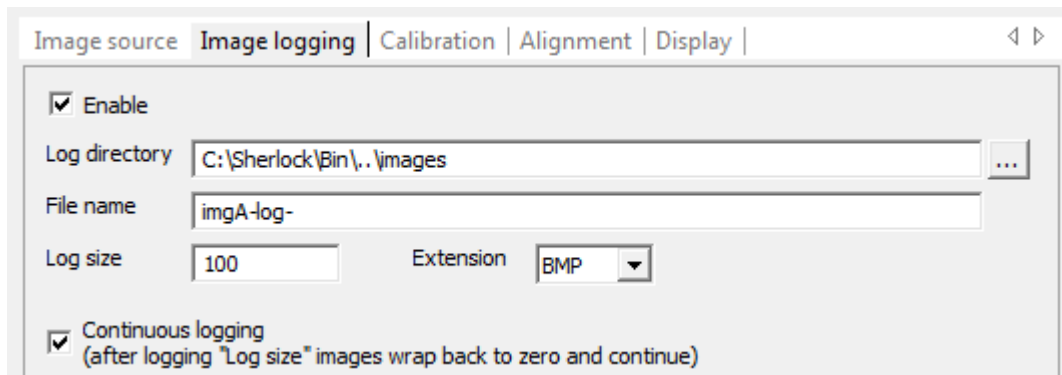
An image can be created from the output of an algorithm.

For example, the **Color Map** algorithm outputs a gray-scale image that maps the colors in a color image to grayscale values. Behavior of the Reading option is algorithm-dependant.



Image logging

When enabled, image logging saves every acquired image to an image file. Logging images is a good way to gather images that you can use later as an image sequence for off-line application development and testing.



The default root file name is **<image window name>-log-**. You can change it to any valid root file name. You can also change the log directory.

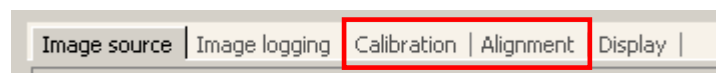
Log size determines how many images will be logged. Each image file is appended with an index number, starting at 0000. With the settings shown above, the images would be named imgA-log-0000.bmp, imgA-log-0001.bmp, ..., imgA-log-0099.bmp.



Saving an image to disk is a relatively slow process. You should not leave image logging enabled when you deploy an investigation, unless you want or need to save every acquired image and are willing to live with the increase in overall processing time.

Calibration and Alignment

Calibration and **Alignment** are explained in separate chapters.



The image window toolbar

Zoom out, reset to normal, zoom in



Display the
Options
dialog

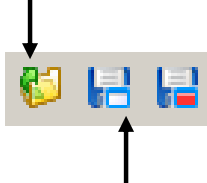
Start/stop camera acquisition for this
image window without processing

Add ROIs to the image window



ROIs define the pixels in an image window that you want to analyze. They are explained in the chapter **ROIs, Algorithms and Preprocessors**.

Load an image from a file into the image window



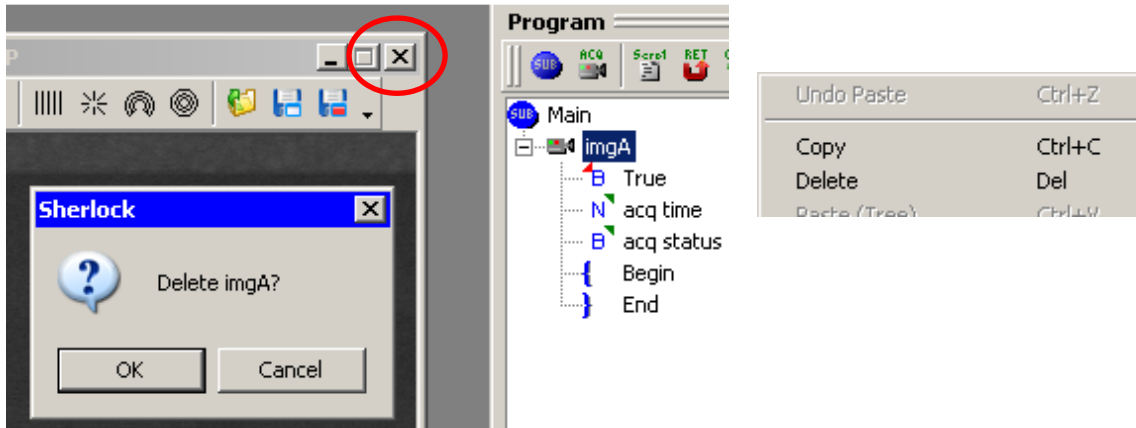
Save the image in the image window with overlay graphics to a file

Save the image in the image window without overlay graphics to a file

Deleting image windows

To delete an image window

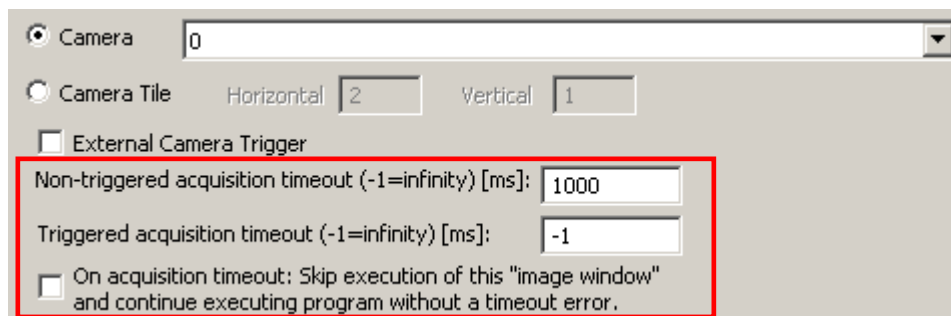
- Click its “close” control
 - Select it in the program window and hit the keyboard **Delete** key
- or
- Right-click it in the program window and select **Delete** from the pop-up menu



Deleting an image window destroys it and its contents completely – any work you have done in the image window is lost. When you delete an image window, a message box gives you the opportunity to cancel the deletion. You cannot restore a deleted image window – no “undo.” (But remember Sherlock\Programs\Autosave.ivs!)

Acquisition timeouts

If a camera is selected for the image window source, you can specify what the investigation should do if the camera does not acquire within a defined time. For information on acquisition timeouts, see the chapter **Acquisition timeouts**.



ROIs, Preprocessors and Algorithms

The second step in a machine vision application is extracting information from an image. This could mean measuring an object (width, height, area, circularity, distance between features, etc.), checking for part presence (for example, making sure a chip has been soldered onto a printed circuit board), reading a barcode, reading a character string, checking for surface defects, verifying colors, and so on.

ROIs

To extract information from an image, you select a set of pixels within the image to process, and select an algorithm to apply to those pixels.

For example, if the purpose of an investigation is to read the character strings on the label of this part, only the pixels within the red box – the region of interest, or **ROI** – need to be processed.



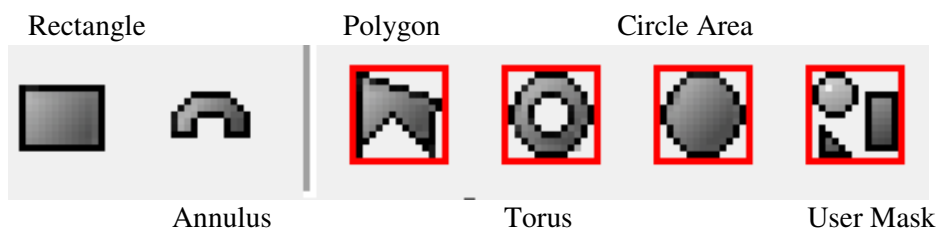
Types of ROIs

There are two main types of ROIs: area and line.

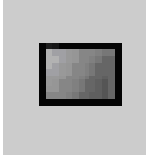
Area ROIs

Area ROIs process the pixels they enclose, including the pixels on their borders.

There are six area ROI shapes.

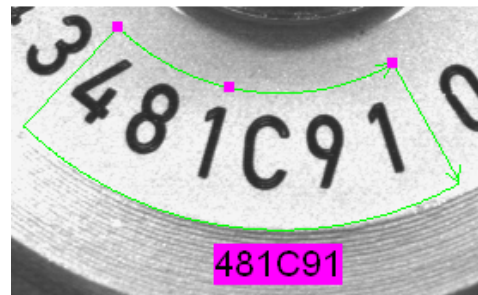
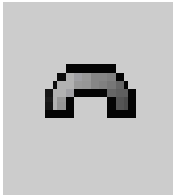


Rectangle ROI



Rectangle ROI reading the barcode

Annulus ROI

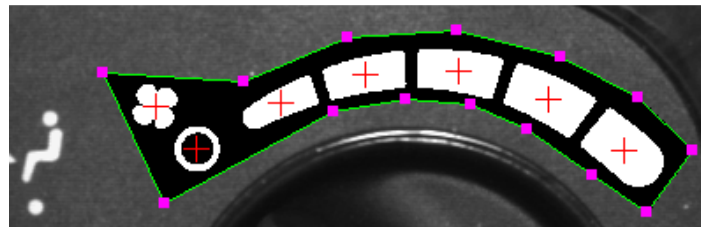


Annulus ROI reading a character string

Polygon ROI

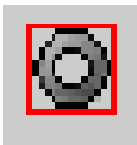


A Polygon ROI can be any shape and can have any number of vertices.

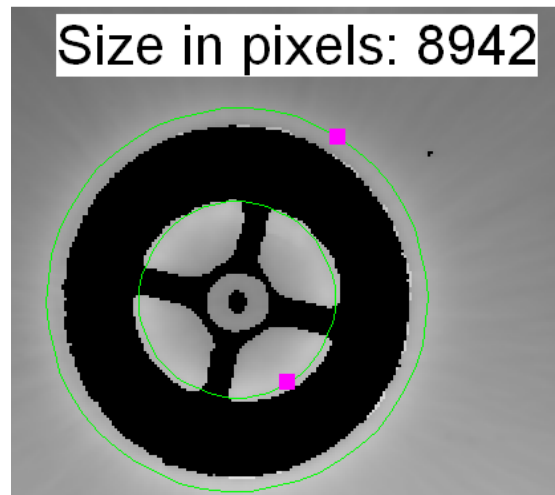


Polygon ROI locating and measuring bright objects

Torus ROI



The Torus ROI processes the pixels between its two defining circles, but not the pixels inside the inner circle.

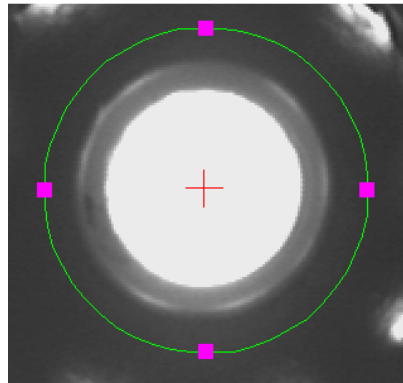


Torus ROI counting the dark pixels in the outer band only (not the spokes or “axle”)

CircleArea ROI



The CircleArea ROI processes all the pixels on its circumference and inside the circle. (Compare to the Circle line ROI.)

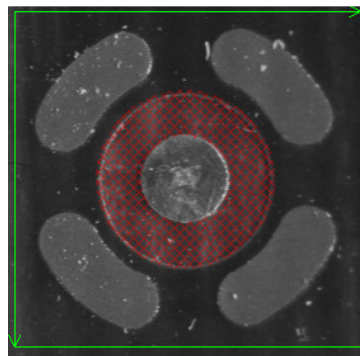


CircleArea ROI finding the center of the bright area

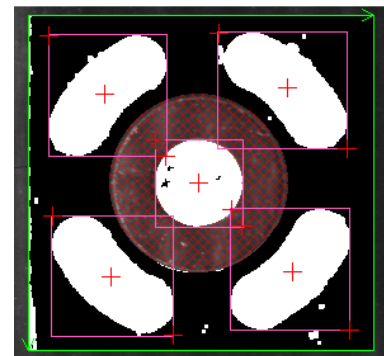
User Mask ROI



With the User Mask ROI, you can define areas within a rectangular area that are excluded from processing.



User Mask ROI with torus-shaped "don't process" area (red hashmarks) before processing



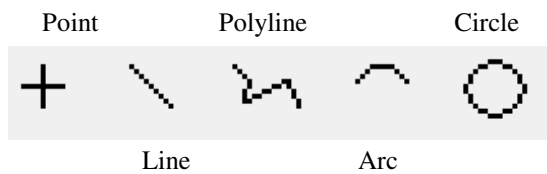
After processing

Line ROIs

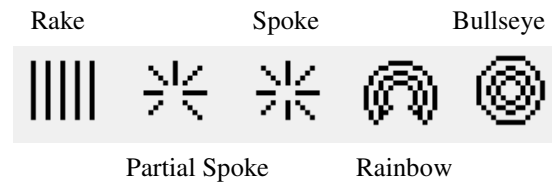
Line ROIs process pixels along a line or lines. They are most often used to find edges.

There are two types of line ROIs, simple and compound.

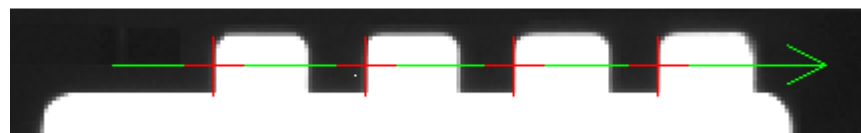
Simple



Compound



Line ROI

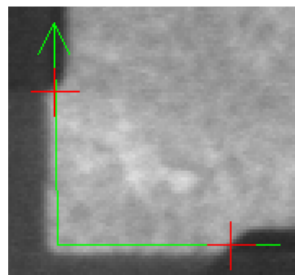


Line ROI finding dark-to-light edges

Polyline ROI



A Polyline ROI can include any number of vertices.

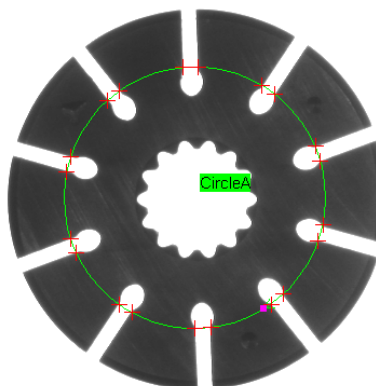


A Polyline ROI finding dark-to light and light-to-dark edges on an irregularly-shaped object

Circle ROI



The Circle ROI processes only the pixels on its circumference, not the pixels inside the circle. (Compare to the CircleArea ROI.)

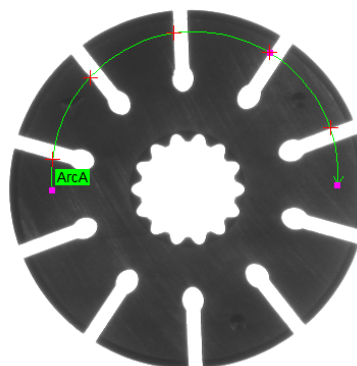


Circle ROI finding dark-to light and light-to-dark edges

Arc ROI



The Arc ROI processes the pixels along an arc.



Arc ROI finding dark-to light edges.

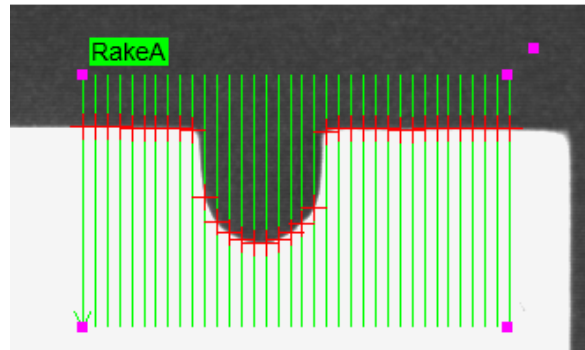


The Point ROI processes a single pixel. There isn't much to show.

Rake ROI



A Rake ROI consists of a series of parallel lines.

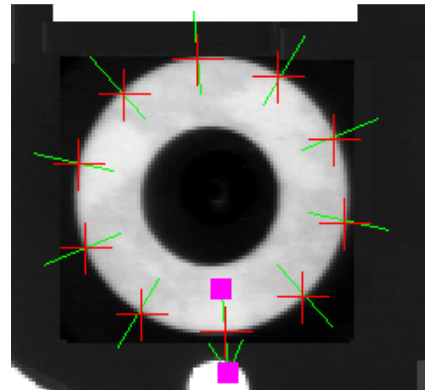


A Rake ROI finding points along the edge of a stamped-metal part.

Spoke ROI



A Spoke ROI consists of a series of lines radiating out from a center point.

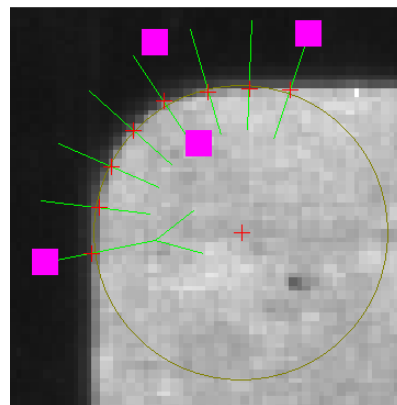


Spoke ROI finding the edge points on a circle

Partial Spoke ROI



A Parital Spoke ROI consists of a series of lines radiating out from a center point, along an arc.

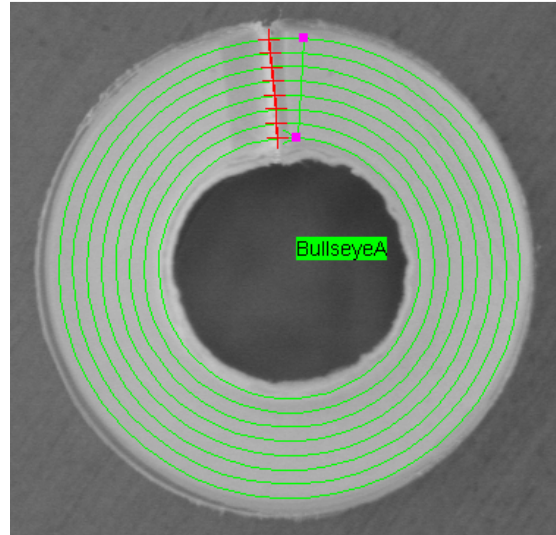


PartialSpoke ROI finding edge points along a curved corner. (The points are then used to find the best-fit circle.)

Bullseye ROI



The Bullseye ROI consists of a series of concentric circles.
The Bullseye ROI processes only the pixels on its component circles, not the pixels between them.

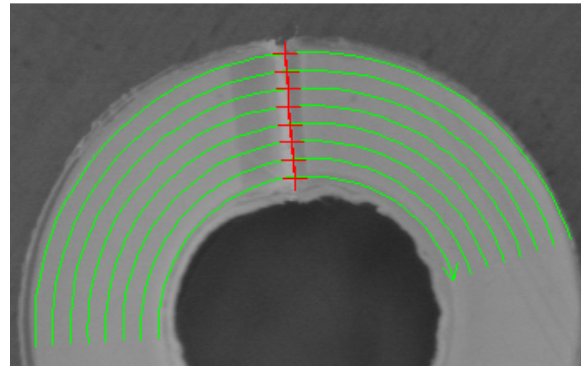


Bullseye ROI finding the defect (dent) in an object by finding light-to-dark edges

Rainbow ROI

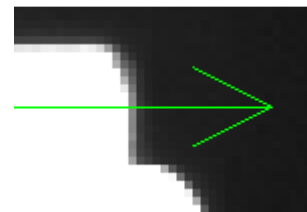


The Rainbow ROI consists of a series of concentric arcs.
The Rainbow ROI processes only the pixels on its component arcs, not the pixels between them.



Rainbow ROI finding the same defect

Every line ROI (except the Point ROI), and every individual line within a compound line ROI, has direction, as shown by the arrowhead. The pixels in a line ROI are processed from the beginning of the line (no arrowhead) to the end of the line (arrowhead). This is important when setting parameters for edge-finding algorithms.



Adding ROIs

To add an ROI to an image window, click its button in the image window's toolbar, then click the number of points necessary to define the ROI. (See the table of **Coordinate indices** later in this chapter.)

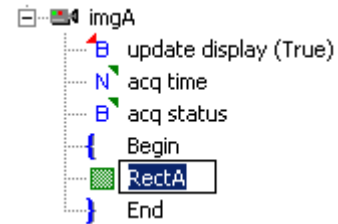
Naming ROIs

ROIs are assigned default names that indicate their type and order of creation: **RectA**, **TorusA**, **RectB**, **RectC**, **SpokeA**, etc.

To rename an ROI, double-left-click on it in the image window or on its name in the program window to display its **Edit** dialog. Enter the new name in the text box at the top.



You can also rename the ROI directly in the program window. Left-click once on the name, wait briefly, then left-click again. Enter the new name.



Deleting ROIs

To delete an ROI

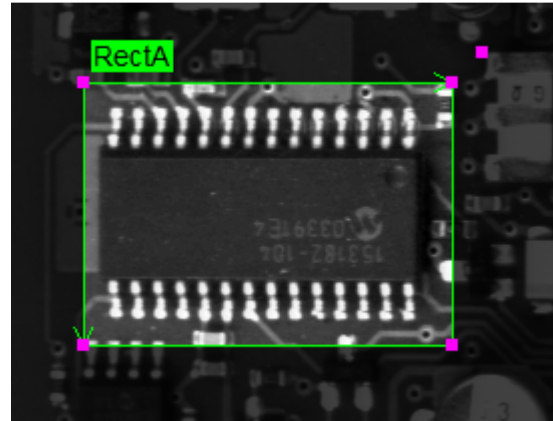
- Select it in the image window or the program window and hit the keyboard **Delete** key or
- Right-click it in the program window and select **Delete** from the pop-up menu

Deleting an ROI is an immediate and irreversible action. You are not given an opportunity to cancel a deletion, nor can you “undo” a deletion.

Preprocessors

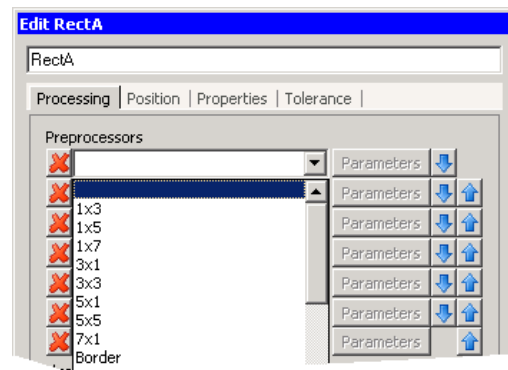
Sometimes it is desirable or necessary to modify the pixels in an ROI before information is extracted from them. A preprocessor modifies the pixels in an ROI, but does not extract information.

For example, here a rectangle ROI calls the **Normalize** preprocessor to expand the gray scale range of a dark image. Such preprocessing might be necessary to make further analysis by Sherlock more robust, or merely to make the image more easily viewed by the user.



Assigning preprocessors

To view the list of preprocessors available for an ROI, double-click on the ROI to display its **Edit** dialog, then click on the drop-down button next to any of the **Preprocessor** entries. (The list is the same for all **Preprocessor** entries within an ROI.)

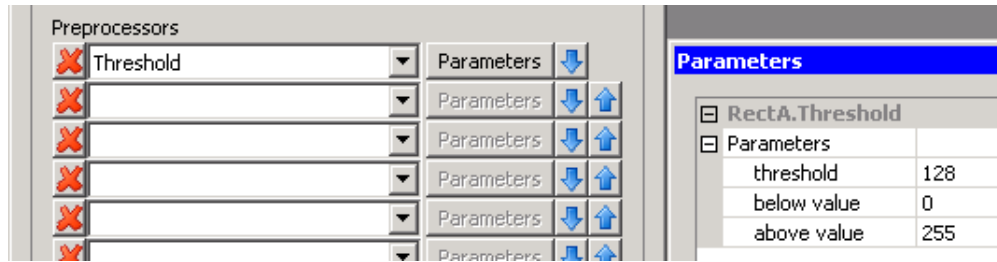


Select a preprocessor from the list assign it to the preprocessor entry.

Preprocessor parameters

If a preprocessor has parameters you can set to control its operation – and many do – the **Parameters** button to the right of the selected processor will be enabled. For example, the **Threshold** preprocessor sets each pixel in an ROI to one of two values, depending on whether the pixel is above or below a threshold. To set the **threshold**, **below value** and **above value** parameters, click on the **Parameters** button to display the preprocessor's **Parameters** dialog.

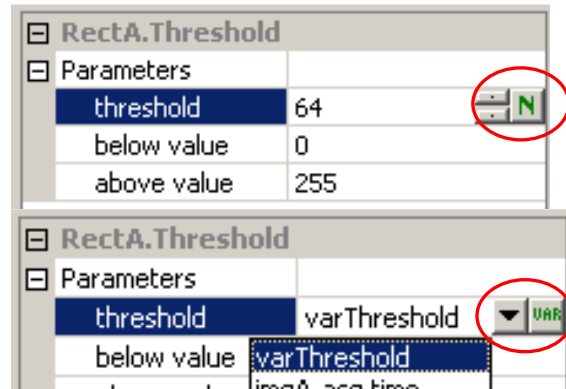
(For online help on a preprocessor's parameters and functionality, highlight the entry and hit the **F1** key, or select **Help→Help topics** from Sherlock's main menu.)



To change a parameter value, you can enter the value directly (**N**)

or select a variable that will contain the value when you run the investigation (**var**).

Click on **N** and **var** to toggle between the two modes.



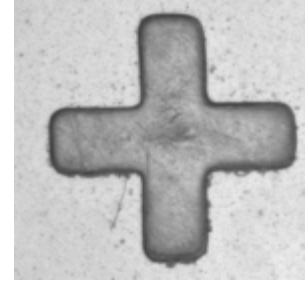
Whether you hard-code a parameter value (**N**) or select a variable that will contain the value at runtime (**var**) depends on how “flexible” the preprocessor has to be. For the **Threshold** preprocessor, if all of the acquired images to be analyzed will exhibit the same grayscale characteristics, you can probably hard-code the **threshold** value. But if the lighting can vary over time or the material you are analyzing can become darker or lighter, you may have to calculate the threshold within the investigation at runtime and assign it to a variable.

(For more information on variables, see the chapter **Program, Readings and Variables**.)

Preprocessor order

Each ROI can execute up to seven preprocessors. Preprocessors execute sequentially. The modifications made by one preprocessor are inherited by the next, so their order is sometimes important.

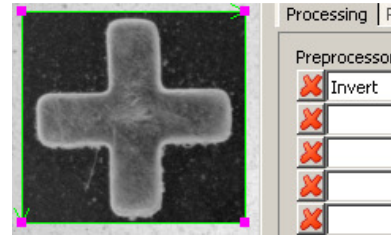
This is the source image for the following examples.



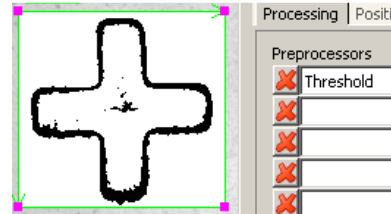
Example 1

Here only an **Invert** processor has been applied.

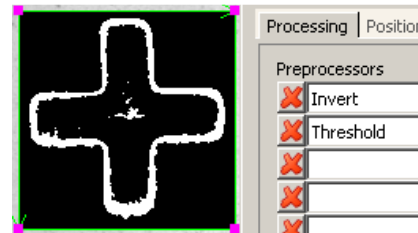
(The **Invert** preprocessor inverts pixel values – 0 becomes 255, 1 becomes 254, 2 becomes 253 ... 255 becomes 0 – making a “negative image”).



Here only a **Threshold** preprocessor has been applied.
(**Threshold** = 128, **below value** = 0, **above value** = 255)

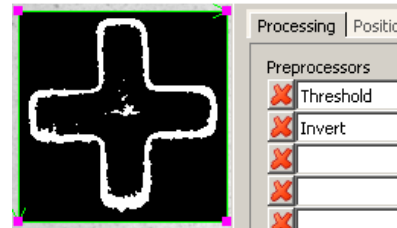


Here the **Threshold** preprocessor follows the **Invert** preprocessor.



Here the **Invert** preprocessor follows the **Threshold** preprocessor.

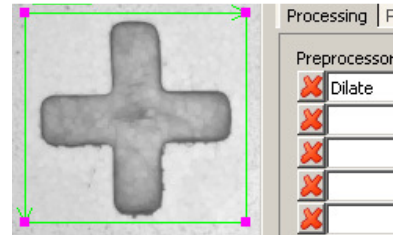
There is no difference in the result.



Example 2

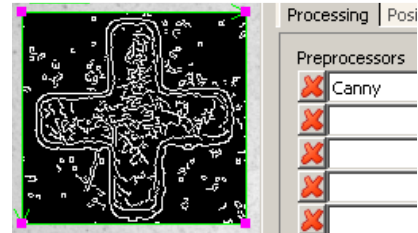
Here only a **Dilate** preprocessor has been applied.

(The **Dilate** preprocessor “thickens” a bright line. Dilating a dark line thins it. The effect is subtle on a full grayscale image.)

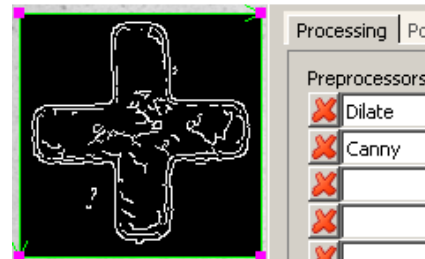


Here only a **Canny** preprocessor has been applied.

(The **Canny** preprocessor is an edge enhancer.)




Here the **Canny** preprocessor follows the **Dilate** preprocessor.

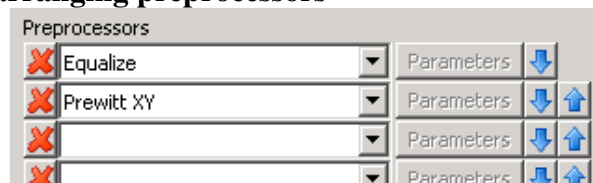




Here the **Dilate** preprocessor follows the **Canny** preprocessor. Big difference!



Deleting and rearranging preprocessors

Use the  button to delete a preprocessor.



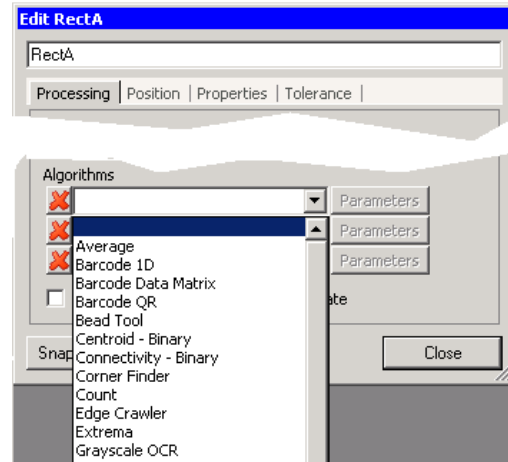
Use the   buttons to change the order of preprocessor execution

Algorithms

An algorithm extracts information from the pixels in an ROI, but does not modify them.

Assigning algorithms

To view the list of algorithms available for an ROI, double-click on the ROI to display its **Edit** dialog, then click on the drop-down button next to any of the **Algorithm** entries. (The list is the same for all **Algorithm** entries within an ROI.)



As with preprocessors, many algorithms have parameters you can set to control their operation. For example, the **Count** algorithm counts the number of pixels of a specific value. To set the value of the pixels to be counted, click on the **Parameters** button to display the algorithm's **Parameters** dialog.

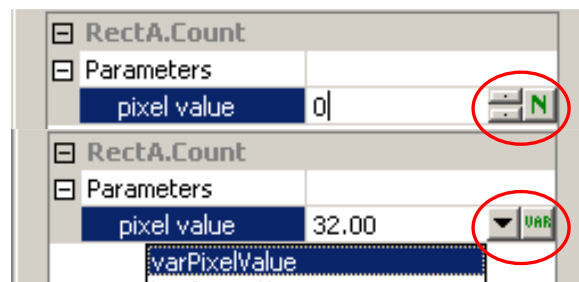
(For online help on an algorithm's inputs, outputs and functionality, highlight the entry and hit the **F1** key, or select **Help→Help topics** from Sherlock's main menu.)




To change a parameter value, you can enter the value directly (**N**)

or select a variable that will contain the value when you run the investigation (**var**).

Click on **N** and **var** to toggle between the two modes.




Each ROI can execute up to three algorithms. Algorithms execute sequentially and independently; one algorithm's execution has no effect on another's.

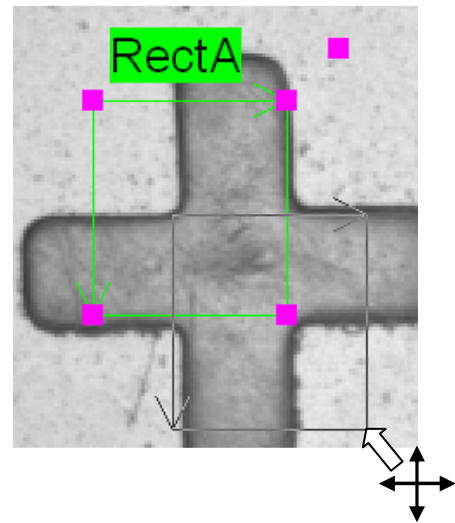
Use the  button to delete an algorithm.




ROI position and properties

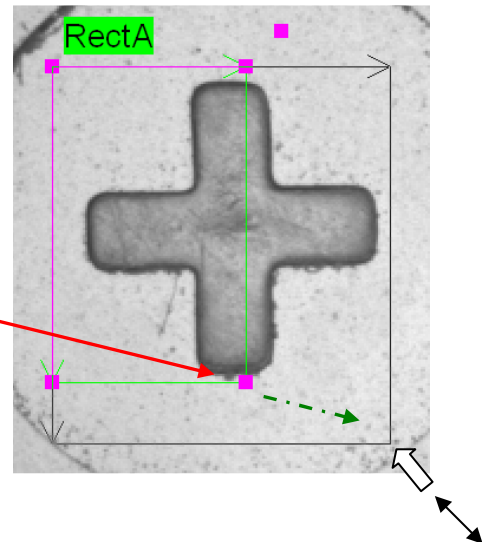
Position and size


To reposition an ROI, move the mouse pointer over the ROI until it turns into the move icon , click-and-hold the left mouse button, and drag.

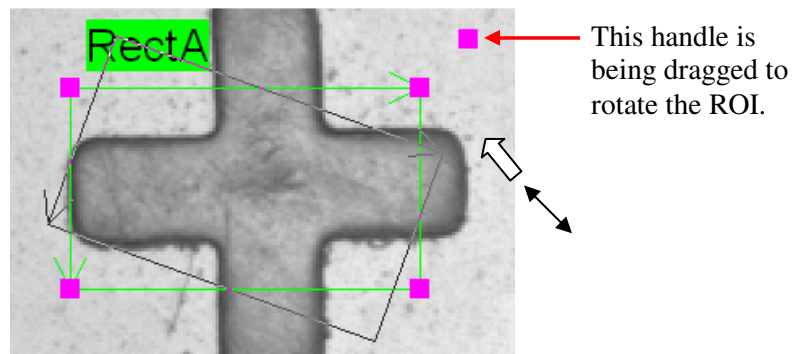


To resize an ROI, move the mouse pointer over one of the ROI's shape handles until it turns into the resize icon , click-and-hold the left mouse button, and drag.

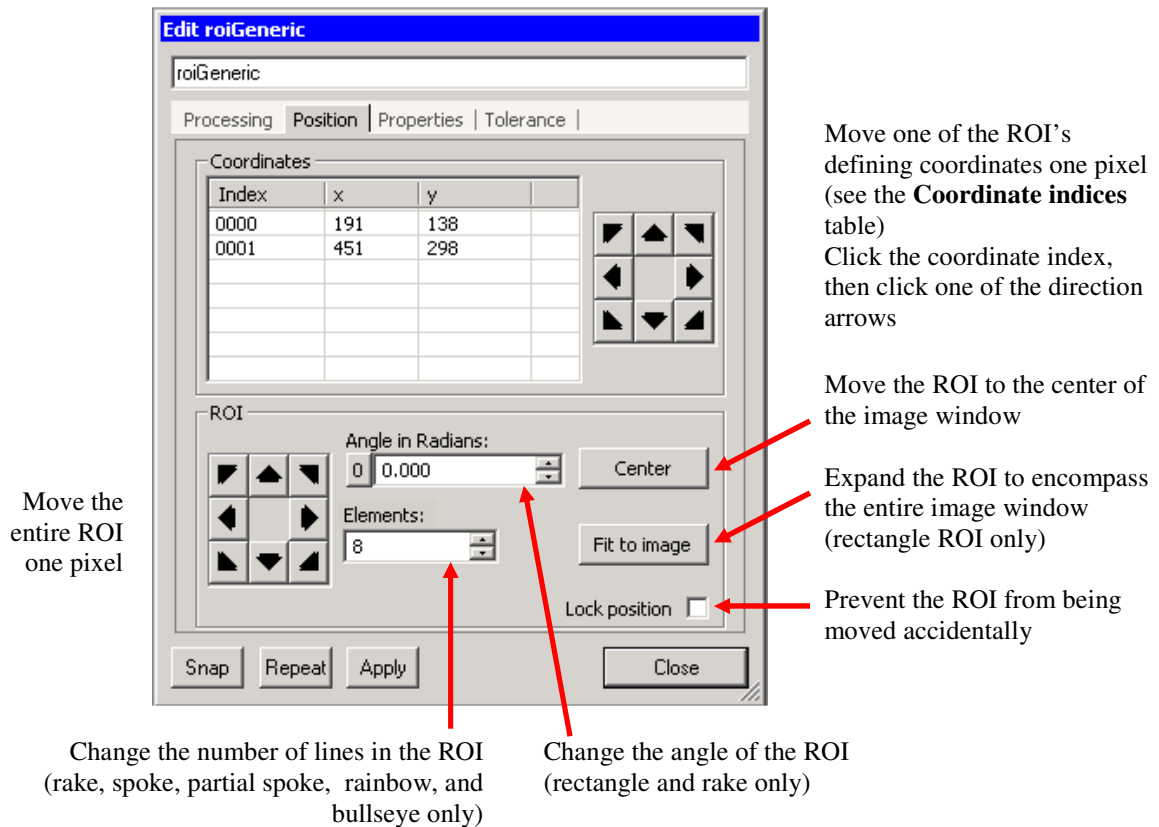
This corner handle is being dragged to resize the ROI.



To rotate a rectangle or rake ROI only, move the mouse pointer over the ROI's rotate handle until it turns into the rotate icon , click-and-hold the left mouse button, and drag.



You can fine-tune the position of an ROI on its **Edit** dialog's **Position** tab.



The **Snap** button acquires a new image in the image window, but does not apply the ROI's preprocessors or algorithms.

The **Apply** button executes the ROI's preprocessors and algorithms. This is a one-time-per-image action; you must acquire a new image before you can click the **Apply** button again.

If you click the **Repeat** button, the **Snap** button repeatedly acquires a new image in the image window and executes the ROI's preprocessors and algorithms. The **Repeat** button is changed to a **Stop** button; click the **Stop** button to exit this mode.

ROI coordinate indices

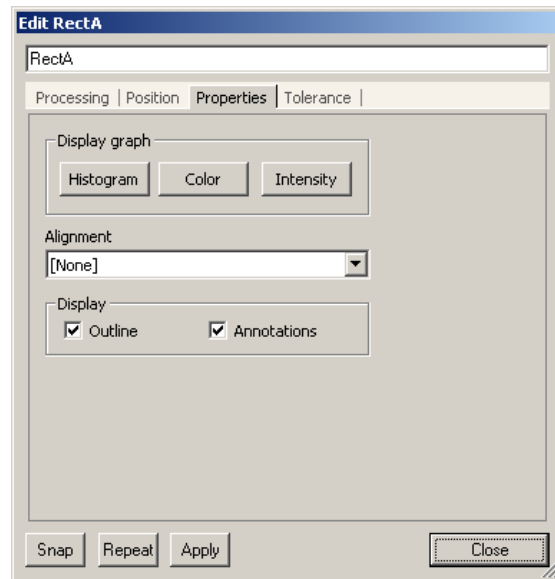
An ROI's position and size are defined by a set of coordinates. The coordinates can be accessed programmatically by indices; each index refers to a single (x, y) coordinate.

ROI type	Coordinate index			
	0000	0001	0002	0003
Point	Point			
Line	Start	End		
Circle	Center	Point on circle		
Arc	Start point of arc	End point of arc	Point on arc	
Rake	Upper left	Lower right		
Spoke	Center	Start point of one line	End point of one line	
Partial Spoke	Start point of arc	End point of arc	Point on inner or outer circle	Point on outer or inner circle
Rainbow	Start point of arc	End point of arc	Point on inner or outer arc	Point on outer or inner arc
Bullseye	Center	Point on inner or outer circle	Point on outer or inner circle	
Rectangle and User Mask ¹	Upper left	Lower right		
Annulus	Start of arc	End of arc	Point on inner or outer arc	Point on outer or inner arc
Torus	Center	Point on inner or outer circle	Point on outer or outer circle	
Circle Area	Center	Point on circle		
Polygon and Polyline	One for each vertex, including the start and end points; double-click the end point			

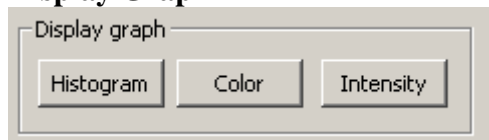
¹ Resizing a User Mask ROI is not recommended, as it can lead to unexpected movement of the “don’t processes” areas within the rectangle. For this reason, it is drawn without handles in its corners, thus preventing interactive resizing. You can, with care, resize it by manipulating its coordinates on the **Position** tab.

Properties

You can extract information about the pixels in the ROI and turn some of its graphics off and on from the **Edit** dialog's **Properties** tab.

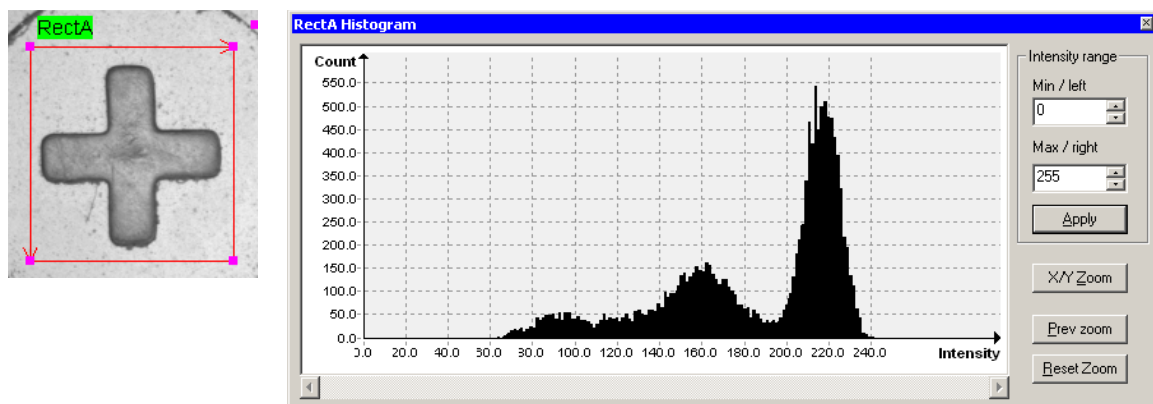


Display Graph



Histogram

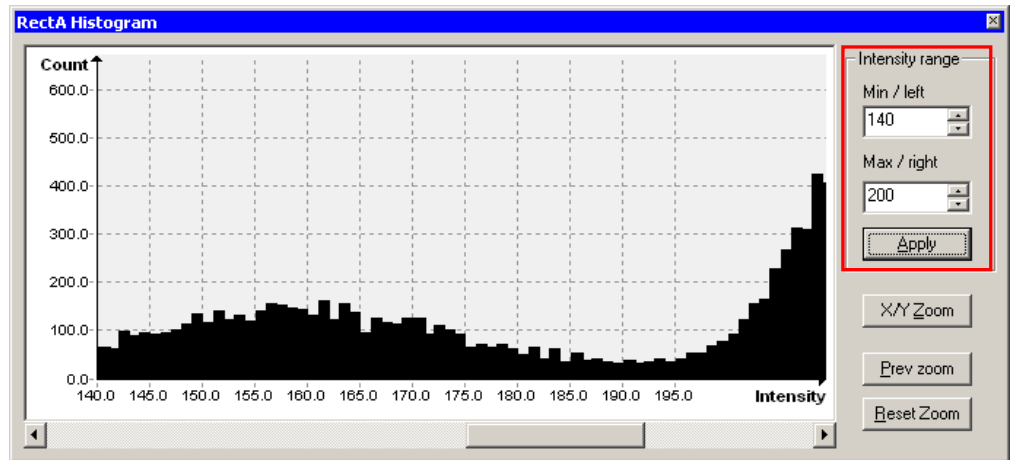
For monochrome area ROIs, clicking the **Histogram** button displays a graph of the count of each pixel value (intensity) in the ROI. The x-axis shows the range of pixel values (0 to 255), and the y-axis shows the count of pixels at each value. **The histogram does not show pixel positions.**



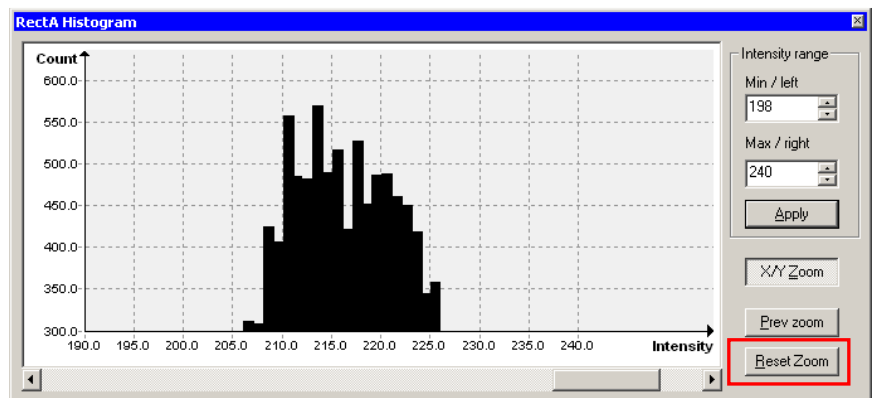
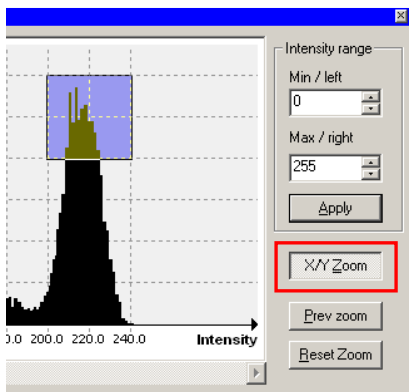
In this ROI, most of the pixels in **RectA** have values between about 60 and 240. The peak to the right is composed mostly of the light-gray pixels outside the cross (values from about 200 to 240), and probably a few inside.

The histogram can be useful for determining what value the **threshold** parameter of a thresholding preprocessor should be. For this ROI, a threshold of about 190 would separate the darker-gray cross from the lighter-gray background.

You can reduce the range of the displayed intensities by changing the values in the **Min / left** and **Max / right** boxes and clicking the **Apply** button.



To “zoom in” on any area of the histogram, click the **X/Y Zoom** button; then click, drag, and release the mouse pointer over the area of interest. Click the **X/Y Zoom** button again to exit zoom mode.



Click the **Reset Zoom** button to return the histogram to its full view.

To determine the exact pixel count at a particular intensity

- move the mouse pointer over the y-axis until it turns into the select icon
- click the left mouse button once
- move the mouse pointer to the left or right until it turns into the arrow icon
- click the left mouse button once

A horizontal detail line is drawn that you can drag up and down.

To determine the exact intensity point on the x-axis

- move the mouse pointer over the x-axis until it turns into the select icon
- click the left mouse button once

- move the mouse pointer up or down until it turns into the arrow icon
- click the left mouse button once

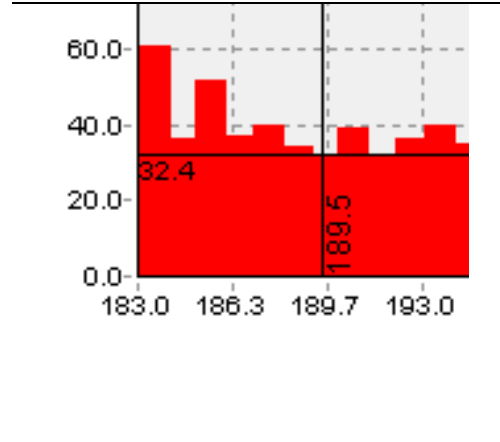
A vertical detail line is drawn that you can drag left and right.

Example

The **X/Y Zoom** button was clicked to zoom in on an area of the histogram.

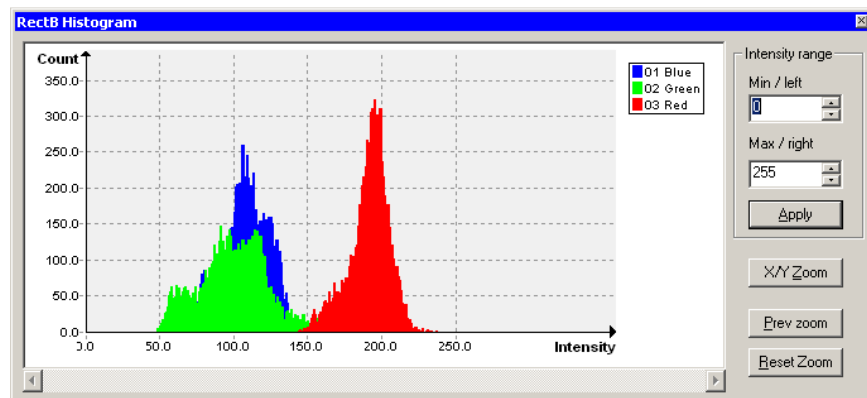
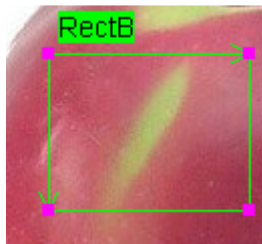
Horizontal and vertical lines were added and dragged to determine that the pixel count at intensity 189 is 32. (The detail lines display floating-point numbers, but there is no such thing as a noninteger intensity or pixel count; round down to the nearest integer.)

(The color of the histogram curve was changed to red by double-clicking the left mouse button inside the histogram to display its **Properties** dialog, clicking the **Curve** tab, and changing the **Linecolor**.)



To delete a detail line, move the mouse pointer over it until it turns into the select icon, then click the right mouse button.

For color area ROIs, clicking the **Histogram** button displays a graph of the counts of the component red, green and blue values in the ROI. **The histogram does not show how these values are combined to make any particular (red, green, blue) pixel.**

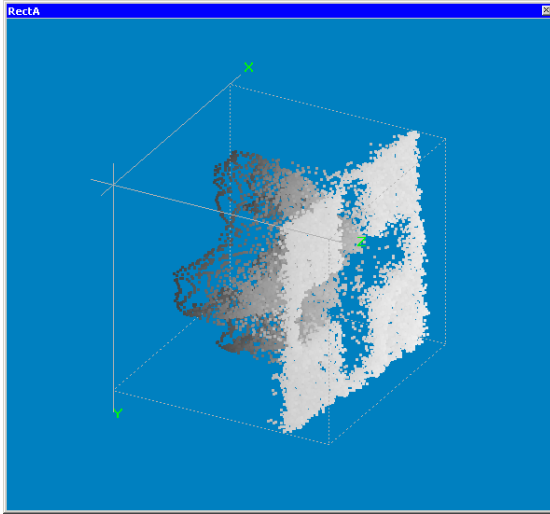


Intensity

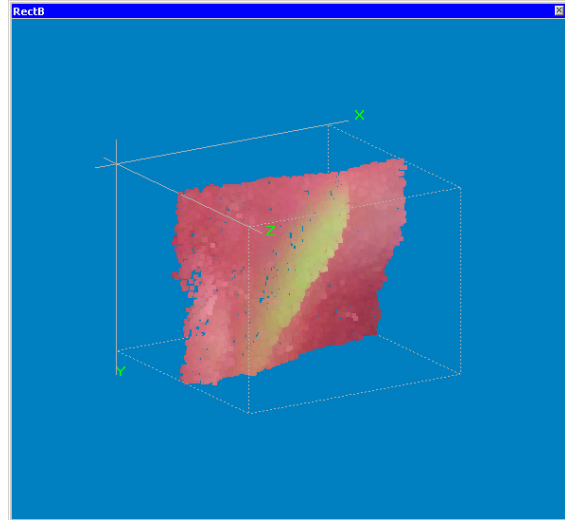
For monochrome area ROIs, clicking the **Intensity** button displays a three-dimensional graph of the position and intensity of each pixel.

For color area ROIs, clicking the **Intensity** button displays a three-dimensional graph of the position and intensity of each pixel, with its corresponding color mapped onto its intensity.

You can reorient the display by holding down the left mouse button and dragging the mouse.



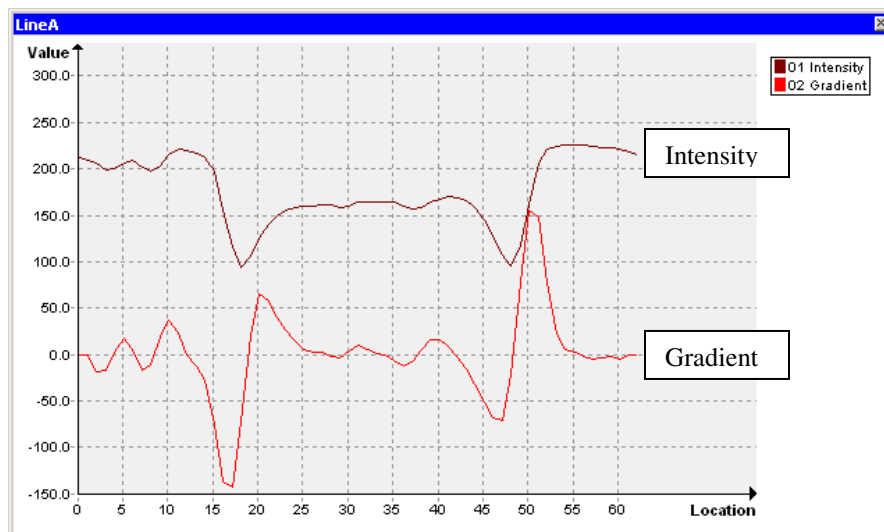
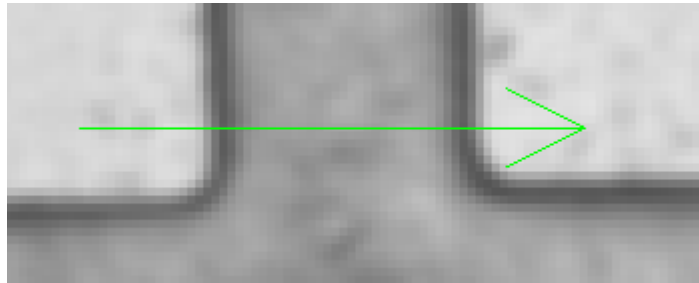
Monochrome intensity display



Color intensity display

Profile

For simple monochrome line ROIs (line, arc, circle, polyline), a **Profile** button replaces the **Histogram** button. Click this button to display graphs of the pixel intensities and gradient (rate of change) along the line.



Intensity

From pixel 0 to approximately pixel 15, the pixel intensity of **LineA** is just above 200. At about pixel 15 (the left-side dark border), the intensity drops down to about 100, then up to about 160

(the gray “inside” of the cross), then down to about 100 again at about pixel 47 (the right-side dark border), then back up to 200-plus from about pixels 48 to 62.

Gradient

The gradient graph shows not absolute pixel intensity, but rate and direction of change. The first significant change in **LineA** occurs at the transition from the light gray background to the left-side dark border, a negative gradient (light to dark). The next significant change occurs at the transition from the dark border to the gray “inside” of the cross, a positive gradient (dark to light). And so on.



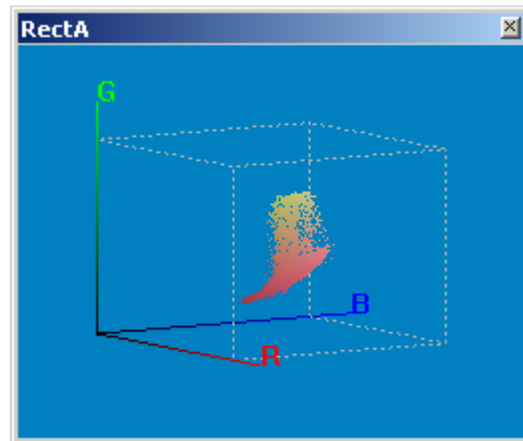
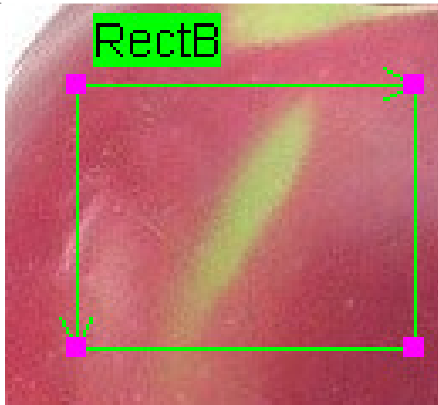
The **Histogram**, **Intensity** and **Profile** displays are not available for compound line ROIs.



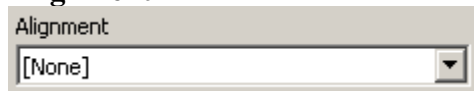
The **Histogram**, **Intensity** and **Profile** displays are available only while you are developing the investigation; they cannot be displayed while the investigation is running.

Color

The **Color** button is enabled for color images only. Clicking this button displays a three-dimensional color cube with the ROI’s pixels arranged according to their (red, green, blue) components.

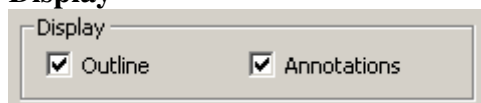


Alignment



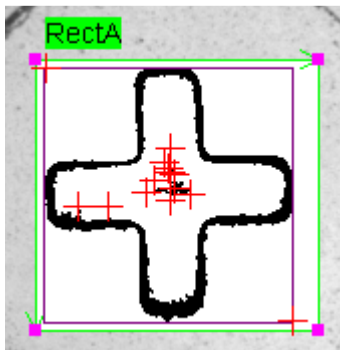
For information on alignment, see the chapter **Landmarks and Alignment**.

Display

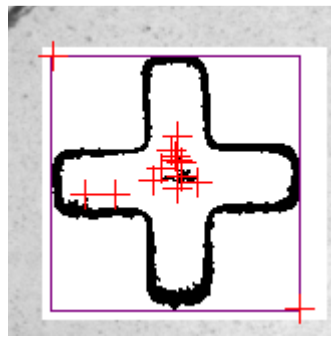


Unchecking the **Outline** box disables display of the ROI’s outline and label.

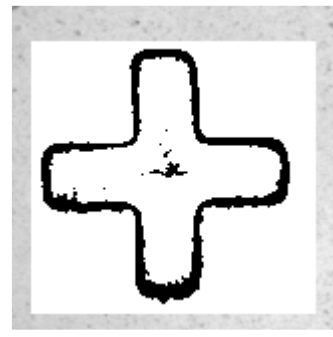
Unchecking the **Annotations** box disables display of the annotations made by certain algorithms.



ROI with outline (green rectangle) and annotations (red crosses and purple rectangle) enabled. The red crosses are the centers of dark objects found by the **Connectivity-Binary** algorithm; the purple rectangle is the bounding box of the largest dark object.

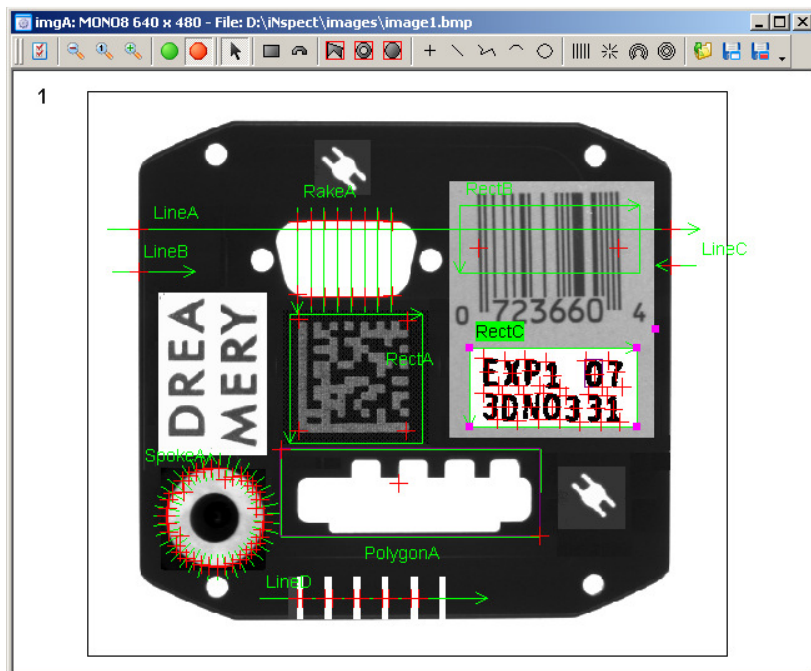


ROI with outline disabled, annotations enabled.



ROI with outline and annotations disabled.

The most common reason for turning off outlines and annotations is to present a less-cluttered image to the user. An image window with more than a couple of ROIs displaying their outlines and annotations quickly becomes quite “busy”.



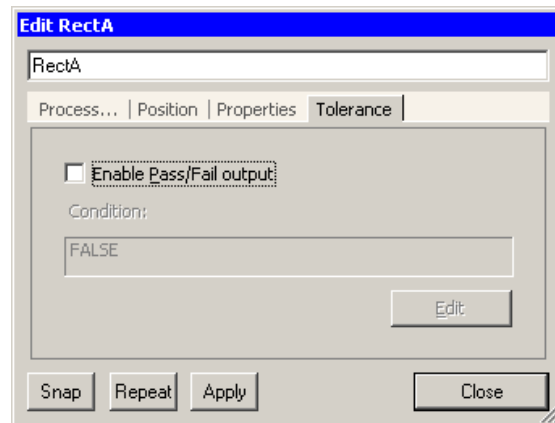
Disabling an ROI’s outlines and annotations has no effect on its execution – all preprocessors and algorithms are still executed.

Disabling outlines and annotations may save some very small amount of processing time, since they do not have to be redrawn on every image update.

ROI Tolerance

You can define conditions that determine whether an ROI passes or fails on the ROI's **Tolerance** tab.

For information on ROI tolerance, see the chapter **Instructions**.



An image window can contain as many ROIs of as many types as you need.

To preprocess or not to preprocess?

Remember that an ROI's preprocessors are executed before its algorithms, and that preprocessors modify the pixels in the ROI. You must be careful that you do not apply preprocessors that modify the pixels so much or in such a way that the algorithms return invalid information.

On the other hand, it is sometimes necessary or beneficial to apply preprocessors before the algorithms, to “clean up” the image or enhance features.

Algorithms named **Something-Binary** (for example, **Connectivity-Binary**) require binarized input, in which the pixels in the ROI are either 0 (black) or 255 (white). This is usually achieved by applying one of the threshold preprocessors.

Program, Readings and Variables

The Program

As you add ROIs to an image window and set their preprocessors and algorithms, they appear in the **Program** pane, within the scope ({ **Begin** / } **End**) of the image window. When you run the investigation, the ROIs' preprocessor and algorithms are executed in order from top to bottom.

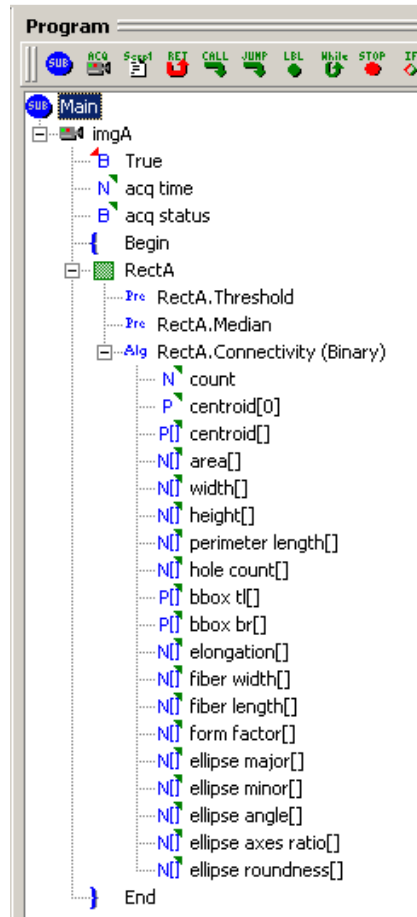
When you run this investigation in Continuous mode

Image window **imgA** acquires an image

ROI **RectA** executes the **Threshold** preprocessor, then the **Median** preprocessor, then the **Connectivity** algorithm

The values **count** through **ellipse roundness[]** are returned by the **Connectivity** algorithm

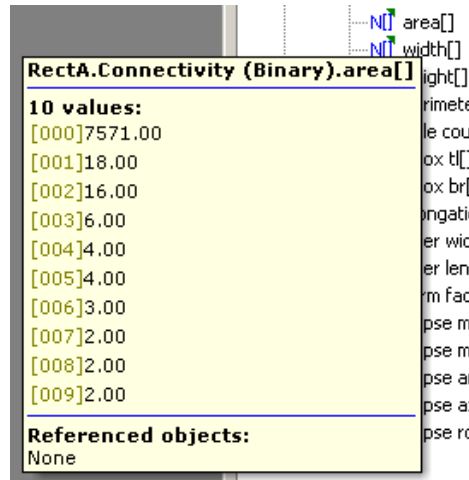
The investigation returns to the beginning and repeats the process



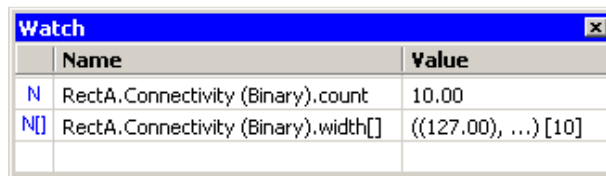
Readings

Every algorithm returns at least one value or “reading”; some algorithms return many readings. There are five types of readings: Number (**N**), Boolean (**B**), String (**S**), Point (**P**), and Line (**L**). Readings can be singular (**N**) or arrays (**N[]**). In the preceding program, the **Connectivity** algorithm returns a single number **count** (the number of objects the algorithm found); a single point **centroid[0]** (the coordinate of the centroid of the first object); an array of points **centroid[]** (the coordinates of the centroids of all of the objects); an array of numbers **area[]** (the areas of all of the objects); etc.

To view the contents of a reading (after running the investigation at least once), hover the mouse pointer over the reading.



You can display readings in the **Watch** pane. To display a reading, drag-and-drop it from the **Program** pane. (If the **Watch** pane is not visible, select **View→Watch** from the main menu .) The reading's value is updated every time the reading changes.

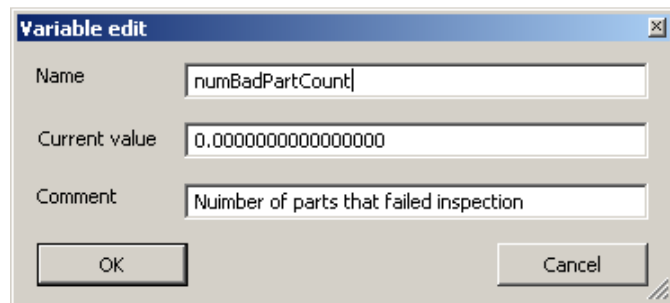


Variables

As in any programming environment, variables in Sherlock are used as temporary placeholders for data. The types of variables you can create are the same as the types of readings algorithms can generate: Number, Boolean, String, Point and Line (singular and arrays for all types).



To create a variable, select the desired type from the **Variables** pane's toolbar. (If the **Variables** pane is not visible, select **View→Variables** from the main menu.) Variables are assigned default names and initial values.



Double-left-click on a variable to display its edit dialog. You can rename the variable, set its initial value and add a comment. It is a good idea to give variables meaningful names – for example, **numBadPartCount** instead of **varA**.

You cannot directly modify algorithm readings – for example, you cannot delete elements from an array of numbers returned as a reading from an algorithm – but you can modify data in variables, so one of the most common uses of variables is as holding places for readings.

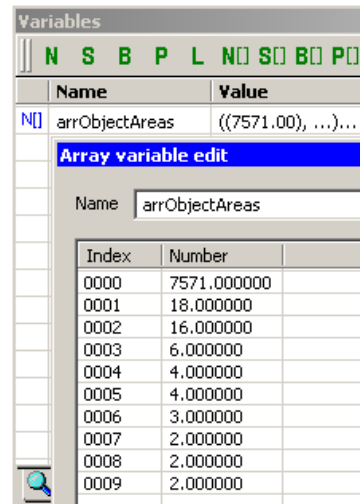
To save a reading to a variable, double-left-click on the reading to display its edit dialog. The **Store in variable** drop down list shows only the variables of the same type as the reading. (**arrObjectAreas** and **varD** are both type **N[]**)



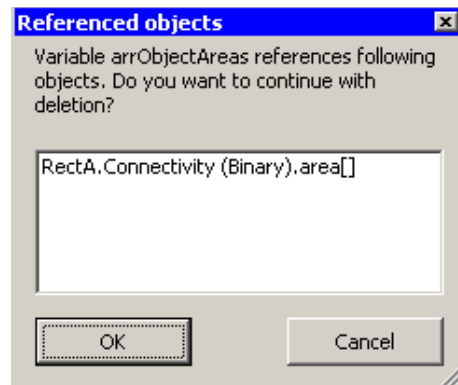
Or right-click on the reading to display its **Connect value** menu.



When you run the investigation, variables connected to readings are filled with the readings' data.
Double-left-click on a variable to view its contents.



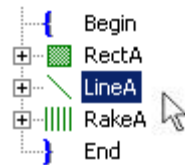
To delete a variable, select it in the Variables pane and hit the keyboard's delete key. If you select a variable that is used anywhere in the program, you will be warned!



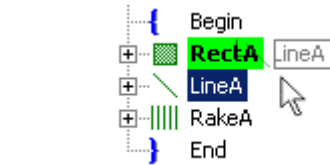
You can drag-and-drop variables within the **Variables** window to put them in any order you want. You can double-click on the **Variable** window's **Name**, **Value** and **Comment** column headings to automatically order the variables.

Rearranging program elements

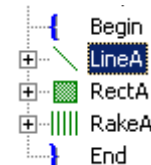
As you add elements to the program, you may find that you need to rearrange them for reasons of program logic. You can move elements with the drag-and-drop method.



To move ROI **LineA** before **RectA**, select it, hold down the left mouse button...



... drag it on top of **RectA**...



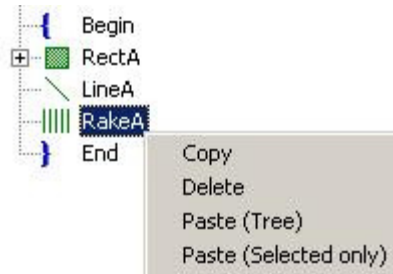
... and release the mouse button.

Copying program elements

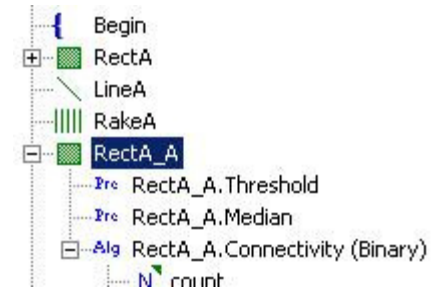
You can copy individual program elements. When you create a copy of an element, the new element inherits the properties of the source element. It is assigned a new name, since two elements cannot have the same name.



To make a copy of **RectA**, select it with the mouse, click the right mouse button, and select **Copy** from the pop-up menu.



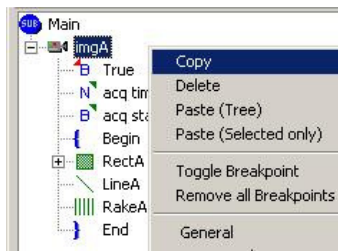
Select the element after which you want to create the copy, click the right mouse button, and select **Paste (Selected only)** from the pop-up menu.



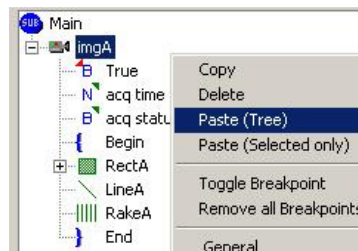
The new element is named **RectA_A**.

After you make a copy of an element, you can edit it to change its name and functionality. Changes made to a copy of an element do not affect the source element, and vice-versa.

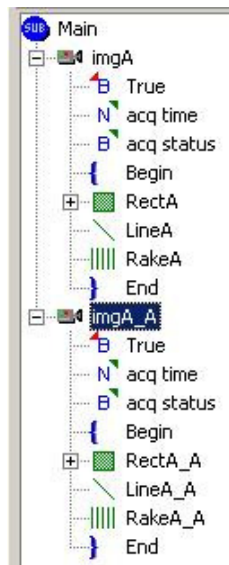
You can copy entire sections of the program tree.



To make a copy of **imgA** and all the elements it contains, select it with the mouse, click the right mouse button, and select **Copy** from the pop-up menu.



Select the element after which you want to create the copy, click the right mouse button, and select **Paste (Tree)** from the pop-up menu.

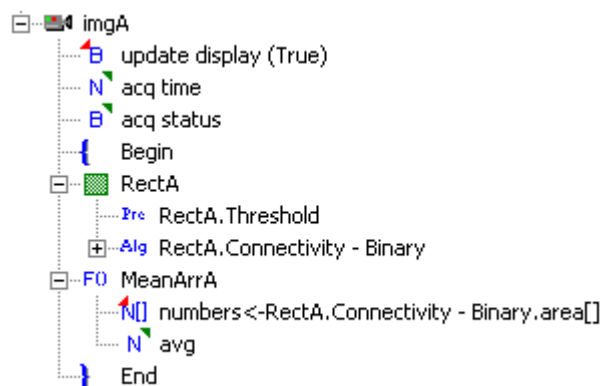


A new image window is created; it contains copies of all of the elements from the source image window.

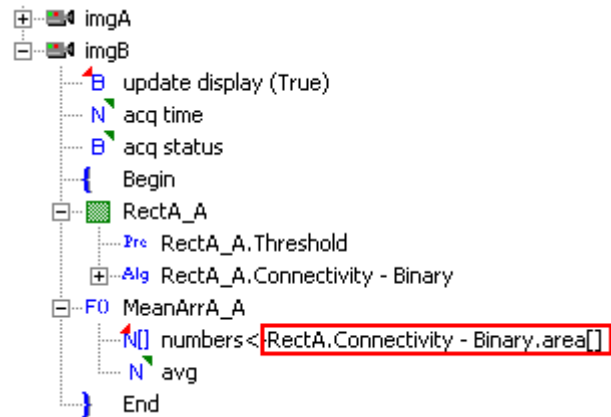


When you copy and paste program elements, you also copy and paste their settings. It is up to you to change the settings of any preprocessors, algorithms and instructions when you copy them so that they perform as you want them to. This is particularly true for instructions' inputs. (For information on instructions, see the chapter **Instructions**.)

This code snippet shows one image window with one rectangle ROI executing the **Threshold** preprocessor and the **Connectivity – Binary** algorithm. The array of areas from **Connectivity- Binary** is passed as input to the **Statistics : Mean Array** instruction to calculate the average object area.



The same functionality was needed in a second image window, so **RectA** and **MeanArrA** were copied from **imgA** and pasted into **imgB**. But note that the input to the new instance of **Statistics : Mean Array** is still the array of areas from **Connectivity – Binary** in **RectA** in **imgA**. The investigation will execute, and **MeanArrA_A** will return an average value, but not the correct one! You have to change the **numbers** input to **RectA_A.Connectivity – Binary.area[]**.



You can copy-and-paste program elements, but you cannot cut-and-paste them. To move program elements, copy-and-paste them, then delete the originals.

Determining variable use

In the process of creating and debugging an investigation, you may create variables that you never use, or you may create variables and forget where they are used. To determine whether and where a variable is used, move the mouse pointer over it in the **Variables** window.

The variable **numGoodPartCount** is referenced by three instructions, **SetNumberA**, **AddA** and **SetNumberB**. It is referenced twice by **AddA** (as the input **number** and as the output **sum**).

Variables	
	Name
S	strTodaysDate
N	numBadPartCount
N	numGoodPartCount
	boolPartFailed

numGoodPartCount
Referenced objects:
 SetNumberA.output
 AddA.number
 AddA.sum
 SetNumberB.output

The variable **strTodaysDate** is not used in the investigation.

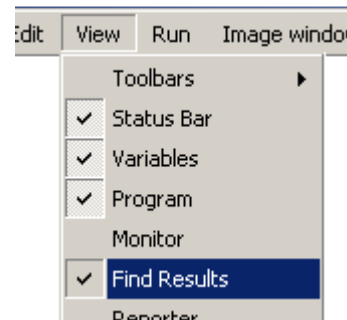
Variables	
	Name
S	strTodaysDate
	numBadPartCount
	numGoodPartCount
	boolPartFailed

strTodaysDate
Referenced objects:
 None

Locating program elements

A typical Sherlock investigation contains hundreds of lines of code, spread across several subroutines. (For information on subroutines, see the chapter **Instructions**.) You may want or need to know where a particular element is located. Continuing the previous example, you may need to locate the instruction **SetNumberA** to see how it uses the variable **numGoodParts**.

First, from Sherlock's main menu, display the **Find Results** window.

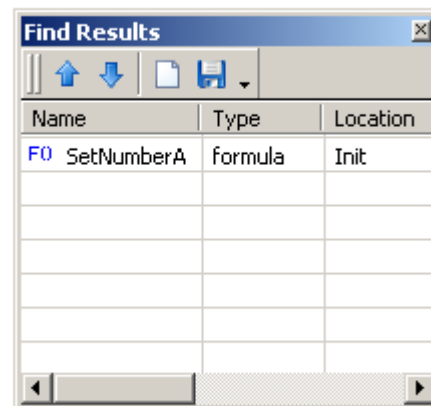
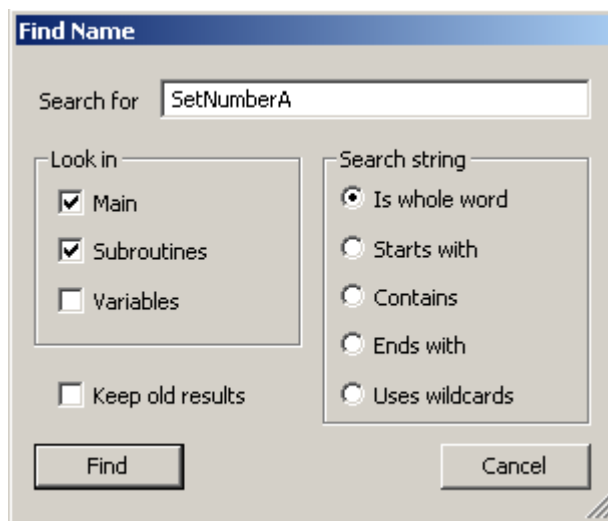


Next, from Sherlock's main menu, open the **Find Name** dialog.

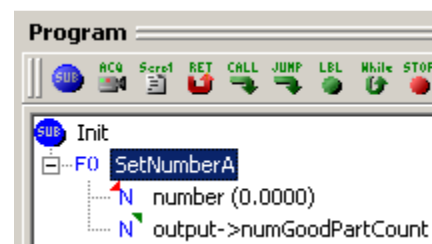


Because **SetNumberA** is an instruction (formula), it will be in either the Main routine or a subroutine.

It is in the subroutine **Init**



Double-click on **SetNumberA** in the **Find Results** window to show its location in the **Program**.



The **Find Name** utility does not show where variables are used in the investigation. The only place it finds variables is in the **Variables** window.

Instructions

Instructions manipulate data, perform calculations, control program flow, communicate with hardware devices, write to and read from text files, and perform many other useful tasks.

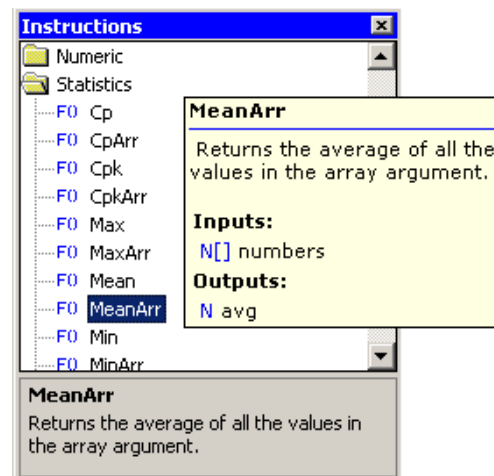
Instructions are divided into functional groups, displayed as folders. Because there are many, many instructions, and because the collection of instructions is constantly expanding, neither all of the groups nor all of the instructions can be discussed here. Here is a brief overview of the functional groups as of Sherlock 7.1.4.x.

Functional group	Contains instructions to
Array : <i>Data type</i>	Manipulate arrays of <i>Data type</i> (Boolean, Line, Number, Point, String). These instructions manipulate the structures of arrays; they do not process their contents.
Boolean	Perform Boolean operations, such as AND, OR, and XOR.
Geometric	Perform geometric calculations, such as fitting points to a line or circle, calculating point-to-point distance, and finding the intersection point of two lines.
IO : <i>Target</i>	Communicate with <i>Target</i> devices or objects, including cameras, ROIs, PLCs and text files.
Numeric	Perform mathematical operations, such add, multiply, divide and compare numbers; and initialize variables
Statistics	Perform statistical operations on sets of numbers.
String	Manipulate strings, such as concatenate, search for substrings, and trim.
Trigonometric	Perform trigonometric operations, such as cosine, arccosine, convert radians to degrees, and convert degrees to radians.

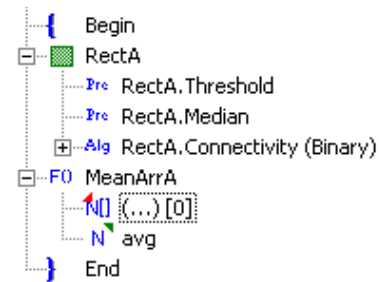
Adding instructions

To display the list of instructions, select **View → Instructions** from the main menu. Open a folder to see the instructions in the group. An instruction's name gives you some idea what the instruction does, although for the sake of brevity, some of the instruction names are a bit cryptic.

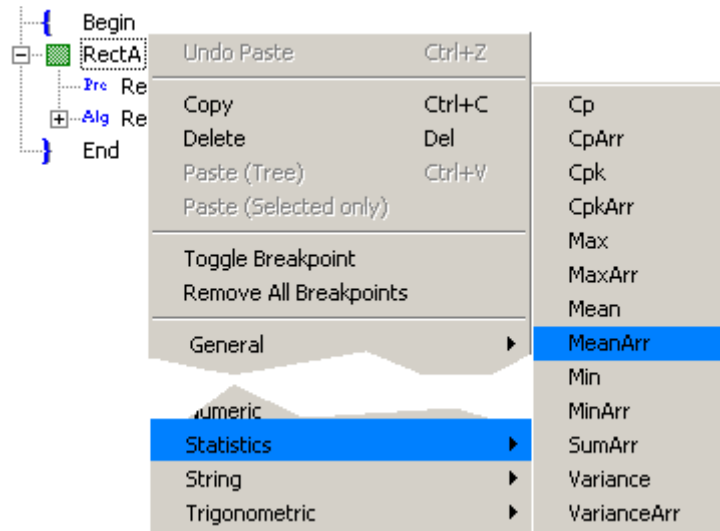
If you hover the mouse pointer over an instruction, a pop-up window describes the instruction's functionality, inputs, and outputs.



Drag-and-drop an instruction on top of the program element after which you want the instruction to appear. (In this example, the **Statistics : MeanArr** instruction was dropped on top of **RectA**. The preprocessors and algorithm are members of **RectA**; they are not program elements themselves.)



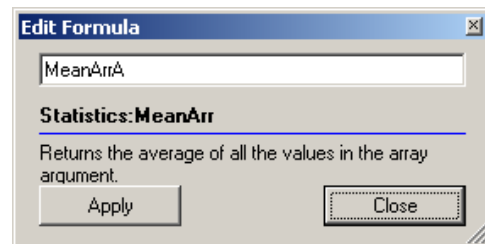
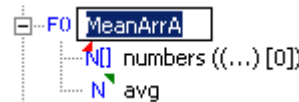
You can also add an instruction by right-clicking an existing program element to display the instruction pop-up menu. Navigate through the menu to the instruction you want, then left-click it. The instruction is added after the program element you right-clicked to display the menu.




Renaming instructions


To rename an instruction, left-click it once, wait briefly, then left-click it again. Enter the new name.

Or double left-click the instruction to display its edit dialog, and enter the new name in the text box.



Instruction inputs and outputs

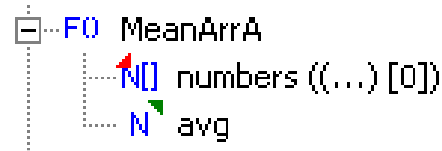
An instruction's inputs are denoted by red inward-pointing triangles. 

Its outputs are denoted by green outward-pointing triangles. 

Not every instruction requires inputs, and not every instruction generates outputs.

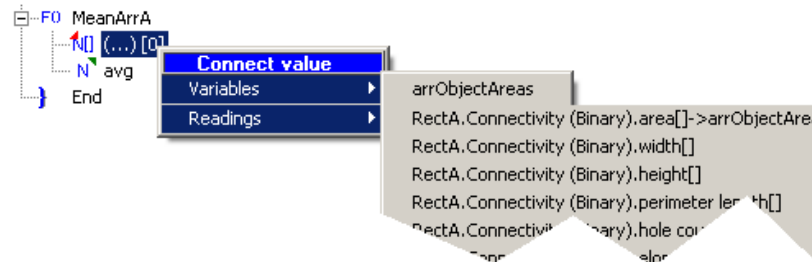
The characters before an input or output name tell you what data type the input needs or the output returns: Number (**N**), Boolean (**B**), String (**S**), Point (**P**), and Line (**L**). Inputs and outputs can be singular (**N**) or arrays (**N[]**).

For example, the **MeanArr** (Mean Array) instruction takes an array of numbers as input (**N[] numbers**), and returns a single number (**N avg**), the average of the numbers in the array.

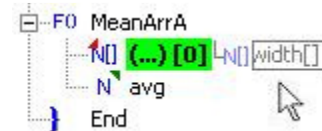


Defining input values

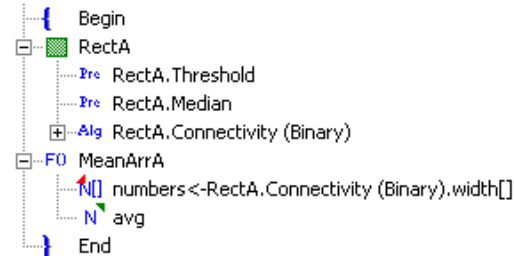
To assign a source to an input, right-click the input to display the **Connect value** menu. Only variables and readings of the correct type are displayed. Select the input from the list.



You can also assign a source by dragging-and-dropping a valid candidate from elsewhere in the program. Here, **width[]** was dragged from **RectA.Connectivity - Binary** and is being dropped on top of the input.



This instruction calculates the average of the widths of the objects found by **RectA's Connectivity - Binary** algorithm. The result is available from the output **avg**.



The Watch window

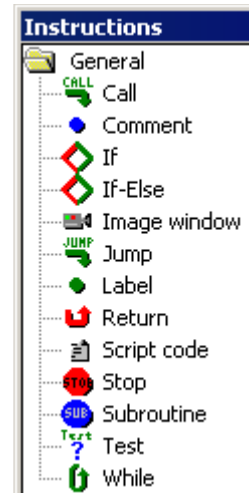
An easy way to view the value of an instruction's output is to drag-and-drop it from the **Program** to the **Watch** window. To display the **Watch** window, select **View → Watch** from the main menu

Watch		
	Name	Value
N	RectA.Connectivity (Binary).count	10.00
N[]	RectA.Connectivity (Binary).width[]	((127.00), ...) [10]
N	MeanArrA.avg	14.90

The **Watch** window is updated every time a value changes.

Program flow instructions

The **General** folder contains instructions that, for the most part, control program flow. These instructions are also available from the **Program** window's toolbar.



To add a program flow instruction to the program:

Drag-and-drop the instruction from the **General** folder to the **Program** window onto the program element after which you want to add the instruction.

or

Right click an existing program element after which you want to add the instruction to display the **Instruction** pop-up menu, and left click the instruction.

or

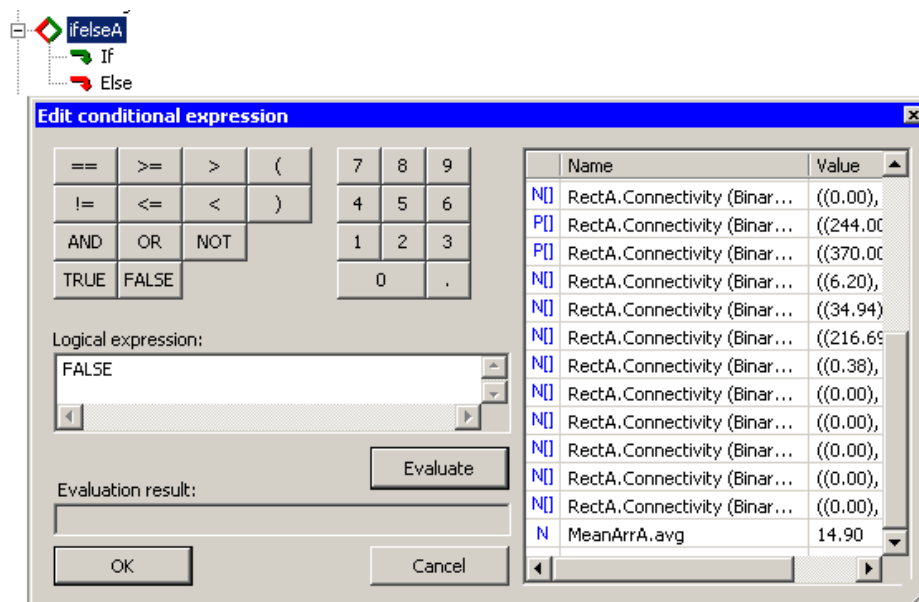
In the **Program** window, left-click to select the program element after which you want to add the instruction, then left click the instruction's button in the toolbar.



If and **If-Else**

The **If** and **If-Else** instructions conditionally execute instructions based on the evaluation of a conditional expression.

When you add an **If** or **If-Else** instruction and double-left-click it, the **Edit conditional expression** dialog is displayed.



The default conditional expression is **FALSE**. You create a real conditional expression from single-value algorithm readings, instruction outputs and variables; the logical operators (**==**, **!=**, **AND**, **OR**, etc.); numbers; and character strings, that evaluates to **TRUE** or **FALSE** at runtime.



In the conditional expression, readings, outputs and variables must be surrounded by square brackets ([]). If you drag-and-drop a reading or variable from the **Name** list on the right of the dialog into the conditional expression box, the brackets are added automatically.

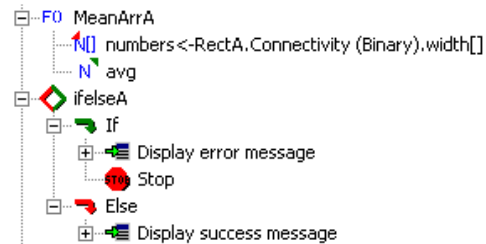
Do not precede the conditional expression with the word “If” — for example, **If [MeanArrA.avg] < 10**. The “If” is implied.



The **Name** list on the right of the dialog contains every single-value reading and variable that exists in the investigation, not just those that will contain “fresh” data when the **If** or **If-Else** instruction is executed. You are responsible for creating a conditional expression that makes sense at the time it is evaluated.

You can add any program elements you want to the **If** and **Else** branch of an **If-Else** instruction.

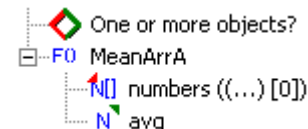
If the logical expression evaluates to **TRUE** when the instruction is executed, the **If** branch of the **If-Else** is followed. If the expression evaluates to **FALSE**, the **Else** branch is followed. (In this example, **Display error message** and **Display success message** are instances of the **IO:Reporter:Print** instruction.)



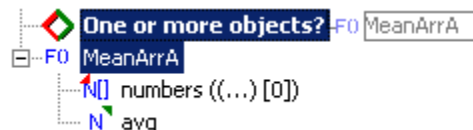
A simple **If** instruction has no **Else** branch.

When you add the first program element to an **If** instruction, it is placed at the same level as the **If** instruction, not within its scope.

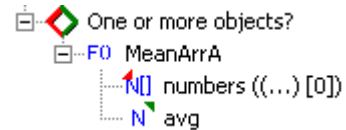
This code snippet is meant to calculate the average value of an array of numbers only if there are one or more objects in the array, as determined by the **If** instruction **One or more objects?** But when the **MeanArr** instruction is added, it is placed at the same level as the **If** instruction. Its execution is not conditional on the result of the **If** instruction – it is always executed.



Here the **MeanArr** instruction is drag-and-dropped on top of the **If** instruction.



Now the **MeanArr** instruction executes only if the **If** instruction evaluates to **TRUE**.



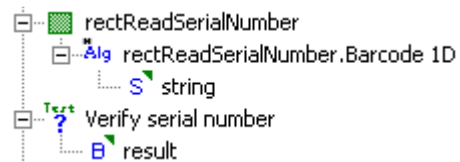
Hint: You can leave either branch of an **If-Else** instruction empty, so you may as well always use **If-Else** instead of **If** instructions.



Test

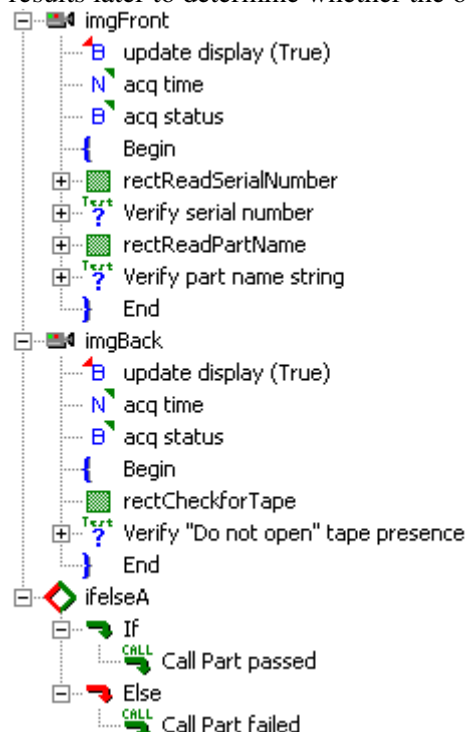
The **Test** instruction uses the same conditional expression dialog as the **If** and **If-Else** statements. It has one Boolean output, which will be either **True** or **False**.

Verify serial number compares the string returned by the Barcode 1D algorithm to the expected barcode.



Conditional expression:
[rectReadSerialNumber.Barcode 1D.string] == "5N2205225"

One use of the **Test** instruction is to generate pass/fail results for several conditions, and use the results later to determine whether the overall inspection succeeded or failed.



Each instance of **Test** compares the result from an algorithm to an expected value. This is the conditional expression for **ifelseA**:

Conditional expression:
([Verify serial number.result] == TRUE) AND
([Verify part name string.result] == TRUE) AND
([Verify "Do not open" tape presence.result] == TRUE)

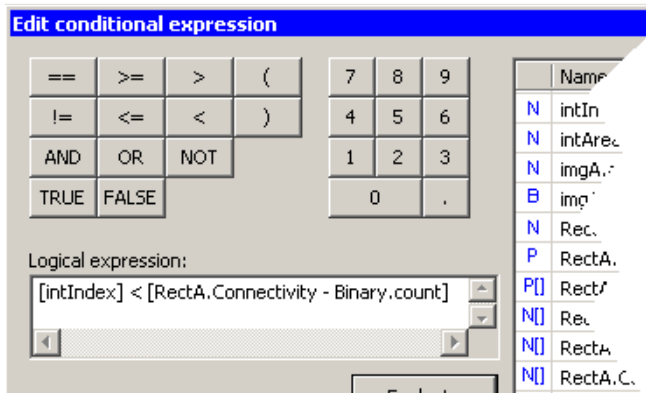
The **Test** instructions could be removed, and the conditional expression for **ifelseA** changed to this:

Conditional expression:
([rectReadSerialNumber.Barcode 1D.count] == "5N2205225") AND
([rectReadSerialNumber.Barcode 1D.string] == "VA40E") AND
([rectCheckforTape.Average.average] > 100)

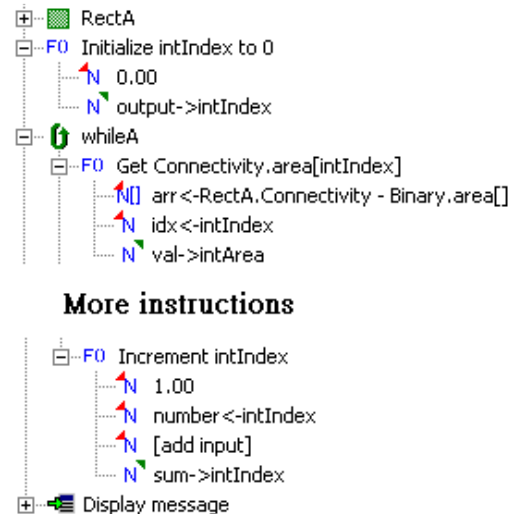
Most people feel that the first is easier to read and manage, but the choice is yours.

While

The **While** instruction repeatedly executes the instructions within its scope as long its conditional expression evaluates to **TRUE**. It is useful for, among other things, extracting data from arrays.



The **While** instruction uses the same conditional expression dialog as the **If** and **If-Else** instructions. In this example, a number variable **intIndex** was created to access individual elements of **RectA.Connectivity.area[]**. As long as **intIndex** is less than the number of elements in the array (the count of objects found), the **While** loop executes the instructions within its scope. (Array indices start at zero).



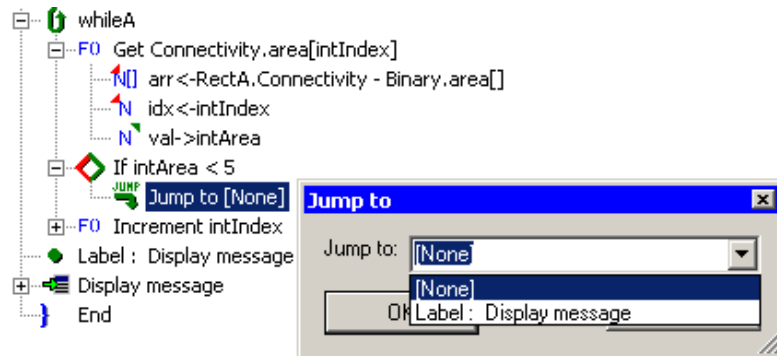
Initialize intIndex to 0 is a renamed instance of the **Numeric : SetNumber** instruction.

Get Connectivity.area[intIndex] is a renamed instance of the **Array:Number GetAt** instruction. The element **Connectivity.area[intIndex]** is extracted and saved to the variable **intArea**.

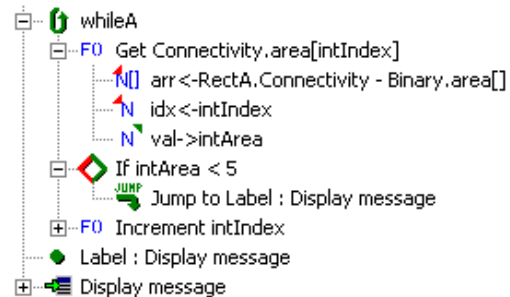
After some analysis is performed (**More instructions**), **intIndex** is incremented with an instance of the **Numeric Add** instruction. The end of the loop is reached, so control returns to the top of the loop. When **intIndex** is equal to **count**, the loop is exited, and **Display message** is executed.

You can use the **Jump** and **Label** instructions to interrupt normal program flow.

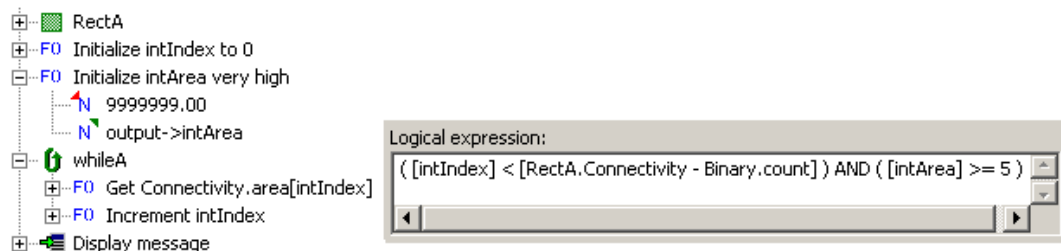
A **Label** is a “do nothing” instruction that can be inserted anywhere and renamed to anything. The **Jump** instruction’s dialog shows all labels.



Within the **While** loop, if an element of the **Connectivity.area[]** array is less than 5, the loop is exited immediately.



Careless use of the **Jump/Label** combination can lead to unexpected and hard-to-diagnose logic problems (“spaghetti code”). It is always possible to write a program such that the **Jump/Label** combination is unnecessary. The preceding example could be rewritten to avoid the use of the **Jump/Label** combination:

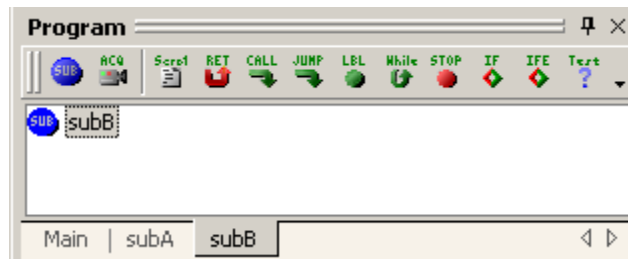


The **While** instruction’s logical expression was rewritten to check for both **intIndex** being within the range of the array, and the last-read area being greater than or equal to 5. If either clause evaluates to **FALSE** the loop is exited. (Note that **intArea** is initialized to a high value [99999999], to ensure that the first time the logical expression is evaluated, the **[intArea]>=5** clause evaluates to **TRUE**.)

Subroutine Call and Return

For the sake of program readability and logic, it is a good idea to divide the program into subroutines.

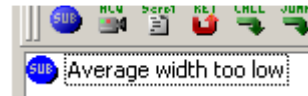
To add a subroutine to the program, click the **Sub** button on the **Program** window's toolbar or drag-and-drop the **Subroutine** instruction from the **Instruction** window's **General** folder. An empty subroutine with a default name (**subA**, **subB**, etc.) is created.



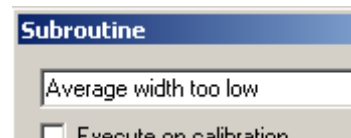
Renaming subroutines

You can and should rename subroutines. It is not uncommon for an investigation to contain several subroutines, and the names **subA**, **subB**, **subC**, etc., give no indication what the subroutines do.

To rename a subroutine, click it once, wait briefly, click it again, and enter the new name



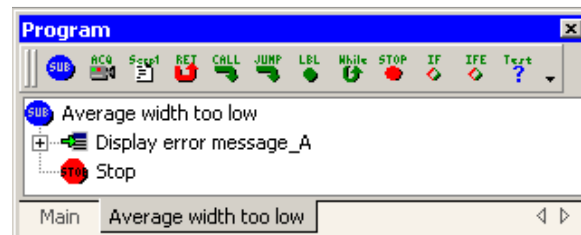
Or double left-click the instruction to display its edit dialog, and enter the new name in the text box.



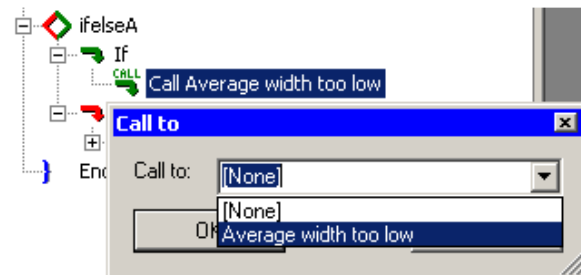
Adding code to subroutines

A subroutine can contain any program elements.

Here the code from the **If** branch of the **If-Else** instruction was copy-and-pasted from the **Main** routine to a subroutine named **Average width too low**.



Back in the **Main** routine, the instructions **Display error message** and **Stop** were replaced with a **Call** instruction. When you double-left-click on a **Call** instruction, its dialog shows all the subroutines in the program. Select the subroutine to which you want to pass control.



Now when you run this investigation in continuous mode

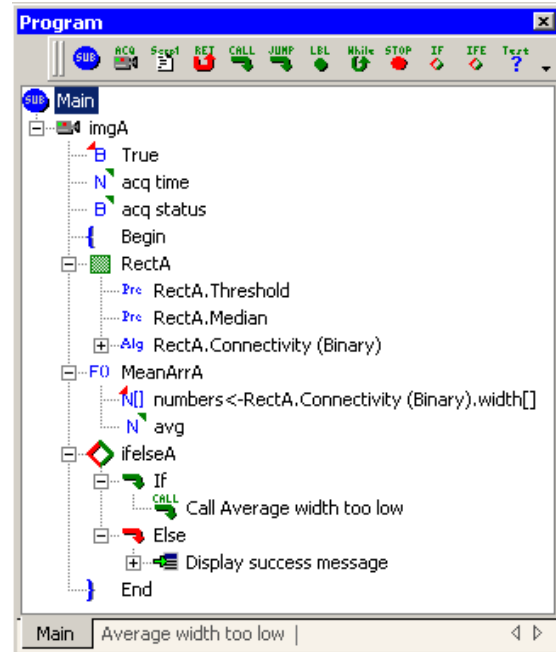
ImgA acquires an image

RectA executes its preprocessors and algorithm

MeanArrA calculates the average of the values in **width[]**

If the average width is less than 10, the subroutine **Average width too low** is called; otherwise **Display success message** is executed

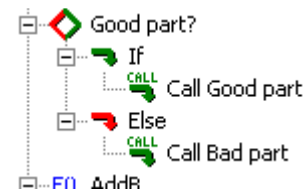
The investigation returns to the beginning and repeats the process



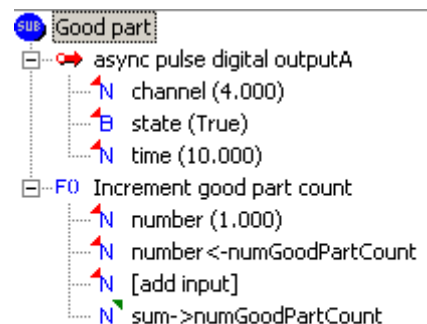
Returning from a subroutine

At the end of a subroutine, control automatically returns to the calling routine, where program execution continues.

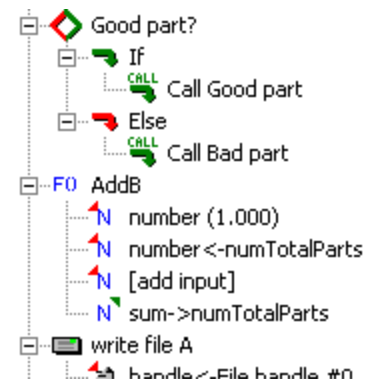
If a part passes inspection, the subroutine **Good part** is called; if it fails, the subroutine **Bad part** is called.



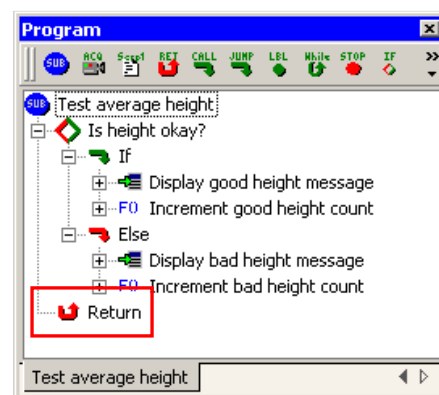
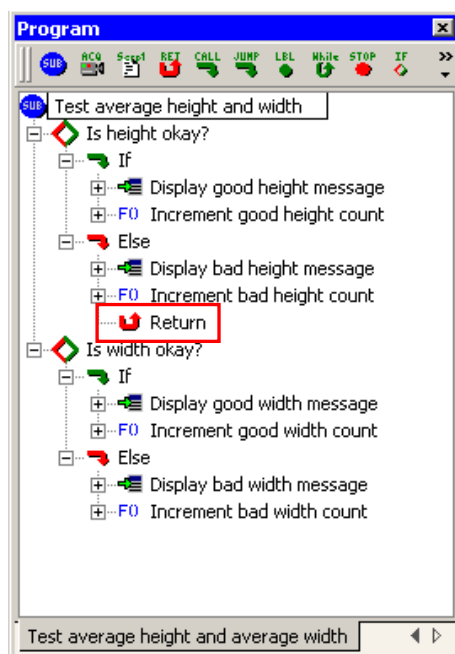
The subroutine **Good part** pulses a signal on digital output channel 4, and increments the count of good parts. The subroutine has reached its end, so control returns to...



... the instruction after **Good part? (AddB)** and execution continues.



The **Return** instruction causes an immediate return from a subroutine to its calling routine.



Control is automatically returned to its calling routine at the end of a subroutine. This **Return** instruction is unnecessary, but harmless.

This **Return** instruction causes an immediate return from the subroutine if the height test fails. The code that tests the width is not executed.

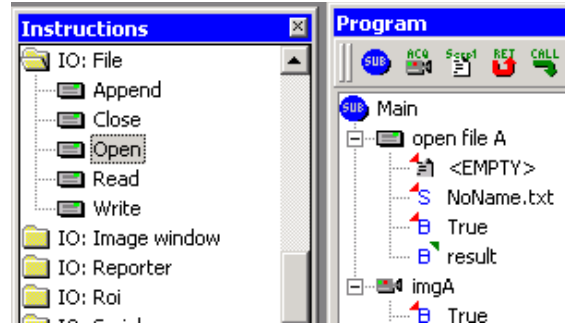


Program execution always starts in the **Main** routine. You cannot rename the **Main** routine, nor delete it.

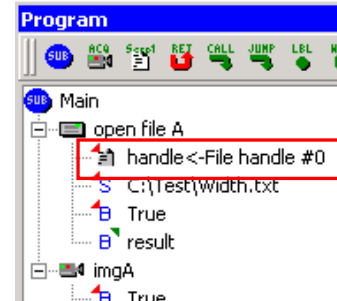
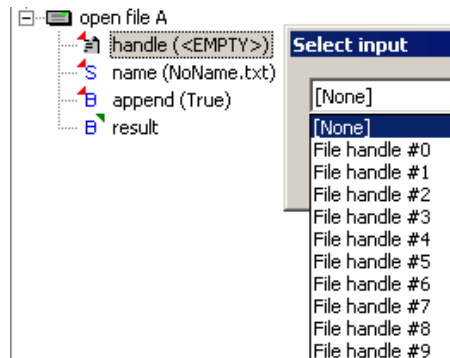
Event-driven subroutine execution

A subroutine can be executed automatically based on an event. Suppose you want to write the good average widths to a text file instead of executing the **Display success message** instruction.

The **IO:File Open** instruction is added to the program.

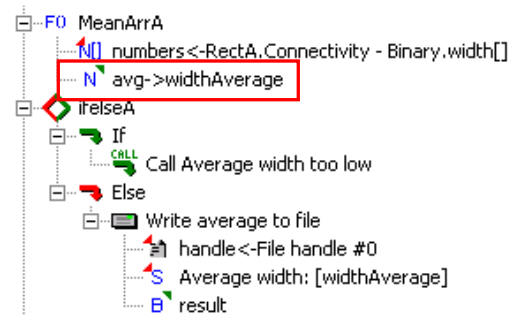


Every open file must be associated with a file handle; up to 10 handles (0 – 9) can be active at once. Select the file handle from the drop-down list that is displayed when you double-left click on the **handle** input.



The **Display good width message** instruction is replaced with the **IO:File Write** instruction. The same file handle with which the file was opened must be selected.

When defining a string to write, surrounding a variable name with square brackets [] means that the value of the variable is accessed. You cannot access readings, only variables, so the **avg** output of **MeanArrA** was saved to the number variable **widthAverage**.



Variables		
Name	Value	
N	widthAverage	0.00

Now when his investigation is run in continuous mode

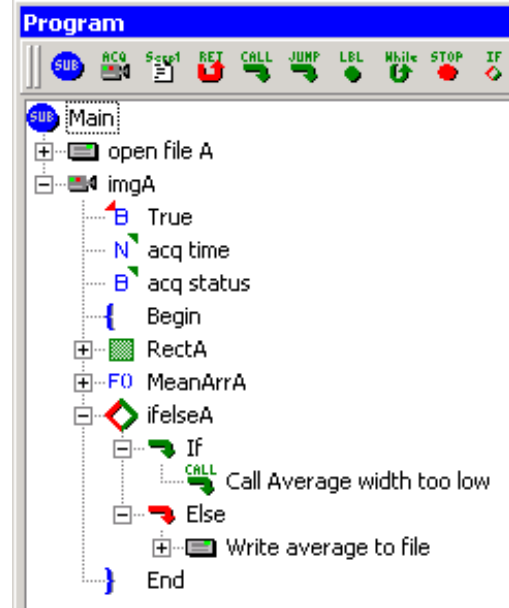
The file **C:\Test\Width.txt** is opened

ImgA acquires an image

RectA executes its preprocessors and algorithm
MeanArrA calculates the average of the values in **width[]**

If the average width is less than 10, the subroutine **Average width too low** is called; otherwise the average is written to the file

The investigation returns to the beginning and repeats the process

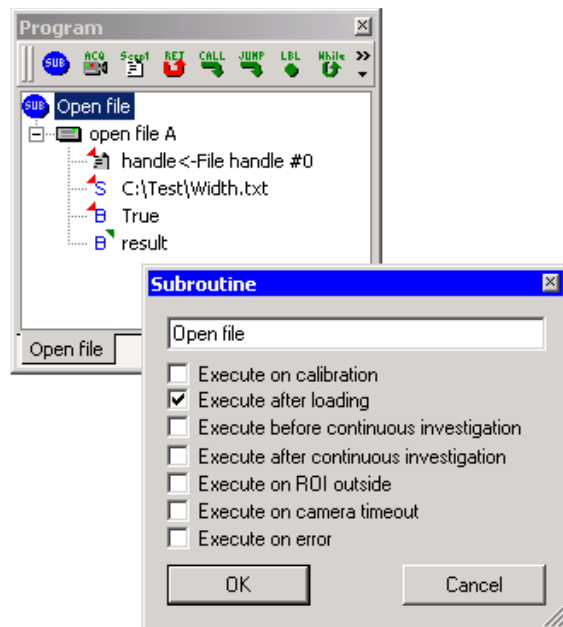


But... It doesn't make sense to repeatedly open the file at the start of every pass through the program. (In fact, the program will run correctly, but every execution of **Open** after the first one will generate a runtime warning.)

To open the file just once, the **Open** instruction is moved to a subroutine. Double-left-click the subroutine name to display its options dialog, and select **Execute after loading** or **Execute before continuous investigation**.

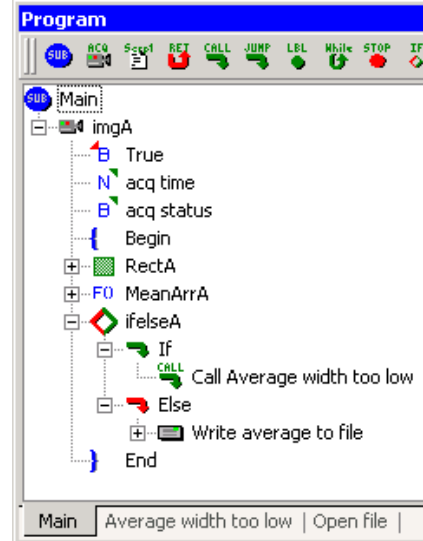
If you select **Execute before continuous investigation**, the subroutine is called automatically every time you click the **Run continuously** button on the main toolbar.

If you select **Execute after loading**, the subroutine is called automatically when you select **Program→Open** from the main menu and load the investigation.



Because the file is opened automatically after the investigation is loaded, the **IO:File Open** instruction in the **Main** routine is no longer necessary.

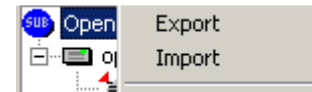
The file is closed automatically when the investigation is unloaded. (You could add code to explicitly close the file with an **IO:File Close** instruction based on some condition, such as the number of passes through the program, an error condition, etc.)



Exporting and importing subroutines

As you develop more and more Sherlock investigations, you may find yourself recreating the same or similar pieces of code over and over again. With some forethought, you can design useful subroutines that can be saved to separate files and inserted into new investigations as needed.

When you right-click on a subroutine name, the **Export** and **Import** items appear at the top of the pop-up menu.



Selecting **Export** displays the **Export subroutine** dialog, with which you can save the subroutine to an .ivb file.

To import a subroutine into an investigation, right-click on any of the investigation's routine names and select the **Import** item from the pop-up menu. From the **Import subroutine** dialog, select the .ivb file you want to import. The contents of the .ivb are imported as a new subroutine, not into the routine you clicked to display the pop-up menu.

Before the subroutine is imported, a dialog box asks you whether you want to create subroutine-specific variables. If you click the **Yes** button, variables that are referenced in the subroutine are created, prepended with the name of the subroutine and an underscore (e.g., **Open file_**), and references to the variables in the subroutines are given the new names. If you click the **No** button, no variables are created, and references to variables are undone.

This subroutine was exported to Init.ivb...

	Name	Value
N	numBadParts	0.00
N	numGoodParts	0.00
S	strTodaysDate	***Empty***

... and imported into a new investigation, with the question **Create subroutine specific variables?** answered **Yes**. Note the new names of the variables.

	Name	Value
N	Initialize_numBadParts	0.00
N	Initialize_numGoodParts	0.00
S	Initialize_strTodaysDate	***Empty***

After a subroutine is imported, you can rename the variables to their original names, provided there are not already variables in the investigation with the same names.



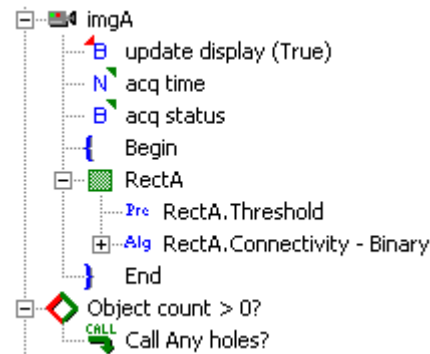
Variable names in text embedded in instructions – **If**, **If-Else** and **While** conditional expressions; the **text** for **IO : Reporter : Print** and **IO : File : Write**, etc. – and JavaScript modules are not updated to the new names. It is up to you to update these names.



References to non-variable program elements – image windows, ROIs, subroutines, alignment objects, calibration objects – are not maintained when you export and import subroutines. You will have to redefine these references.

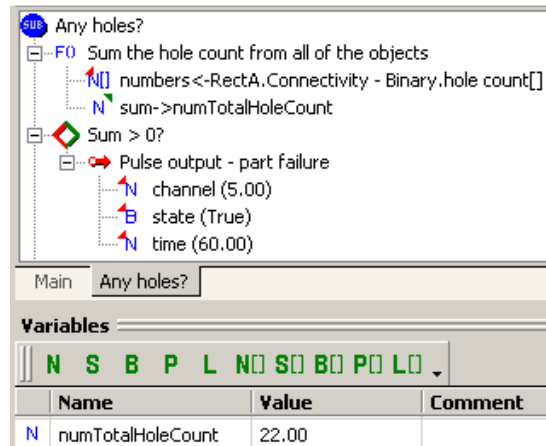
An .ivb file is a binary encoding of a subroutine; you cannot open it outside of Sherlock.

This routine executes the **Connectivity – Binary** algorithm from the ROI **RectA**. If at least one object was found, this routine calls the subroutine **Any holes?** ...



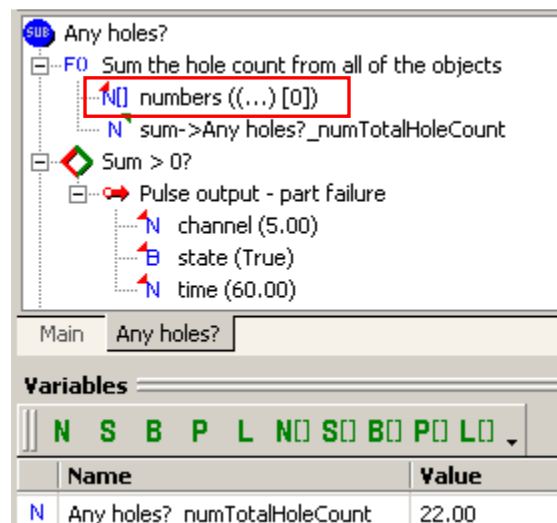
... which determines whether any of the objects has holes, by calling an instance of the **Statistics : SumArr** instruction with **RectA.Connectivity – Binary.hole count[]** assigned to its **numbers** input. A **sum** greater than 0 means indicates a bad part, in which case a pulse is sent out on digital output channel 2.

The **sum** output of the **SumArr** instruction is saved to the variable **numTotalHoleCount**.



When you export **Any holes?** to an .ivb file, import it into a new investigation, and answer **Yes** to **Create subroutine specific variables?**, **numTotalHoleCount** is renamed **Any holes?_numTotalHoleCount**.

The reference to **RectA.Connectivity – Binary.hole count[]** is undone. You must redefine the input to the **numbers** parameter.



If you import a subroutine into an investigation that already contains elements with the same names as elements in the subroutine, the elements in the subroutine are automatically renamed by appending **_A** or **_B** or **_C**, etc. to them.



Stop

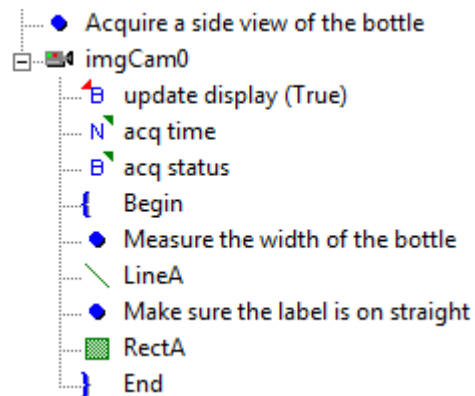
The **Stop** instruction causes an immediate halt of the investigation. The current iteration of the investigation is not completed, but subroutines marked **Execute after continuous investigation** are called if the investigation is running in continuous mode.

This behavior is slightly different than both the main toolbar button **Stop after completing current iteration**, which allows the current pass through the investigation to complete, and allows subroutines marked **Execute after continuous investigation** to be called; and the toolbar button **Abort**, which causes an immediate halt of the investigation, and suppresses calls to subroutines marked **Execute after continuous investigation**.



Remark

A good developer comments her code, with the thought that at some point someone may need to figure out how the program works. Use the **Remark** instruction with wild abandon. It is non-executing, so it will not increase investigation execution time.

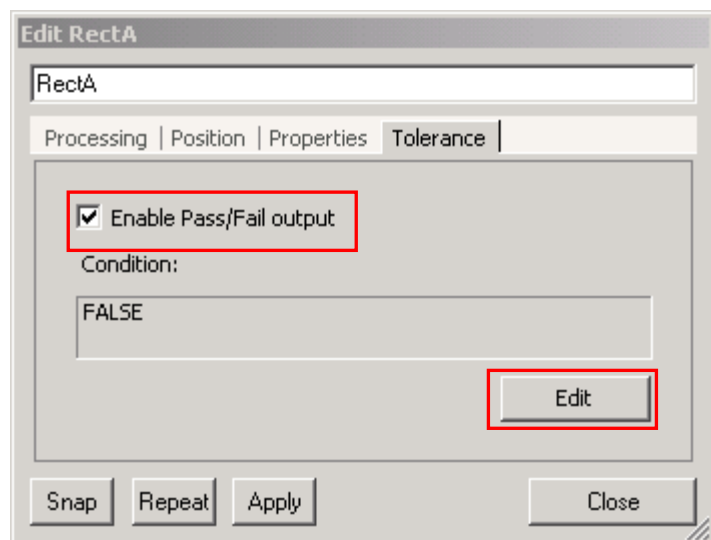


ROI Tolerance

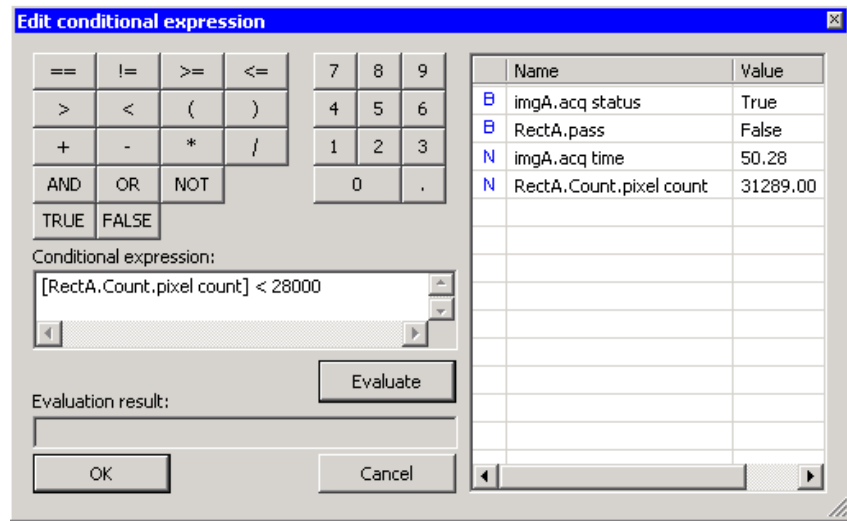
Every ROI has a **Tolerance** tab on which you can define a condition that determines whether the ROI passes or fails.

Check **Enable Pass/Fail output** to enable the **Edit** button.

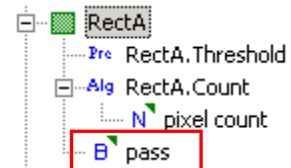
Click the **Edit** button to display the same **Edit conditional expression** dialog as for the **If**, **If-Else**, and **While** instructions.



The ROI **RectA** executes the algorithm **Count**. If the returned **pixel count** is less than 28000, the conditional expression evaluates to **TRUE**.

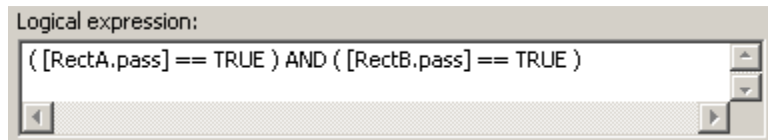


In the program, the ROI displays a new reading, **pass**, which will return **TRUE** or **FALSE**, the state of the conditional expression.



Either immediately after the ROI execution or later in the program, you can test the ROI's **pass** reading, either by itself or in combination with other readings.

Here, the **pass** readings from two ROIs are tested in the conditional expression of an **If** instruction.

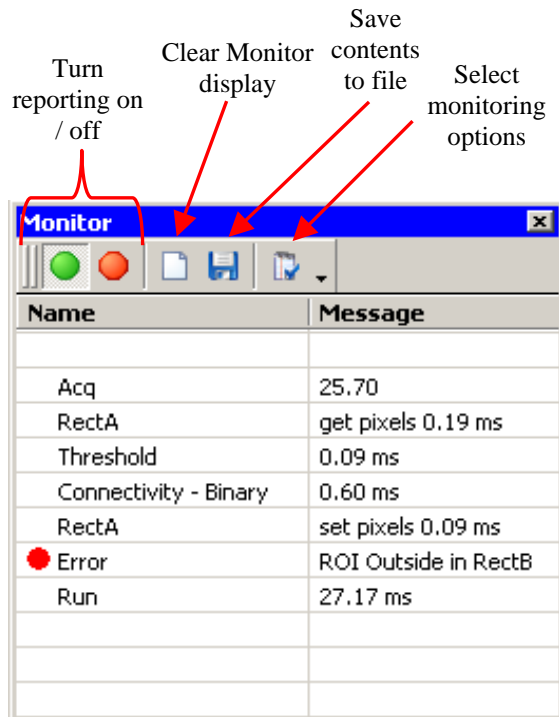


Although the **Name** list shows every single-value reading and variable that exists in the investigation, not just the readings from the ROI's algorithms, the idea is to create an expression that evaluates only results returned by the ROI's algorithms to determine whether the ROI met some pass/fail criteria. Otherwise, if you want to evaluate results returned by algorithms from several ROIs, you may just as well add **If-Else** or **Test** statements to the program.

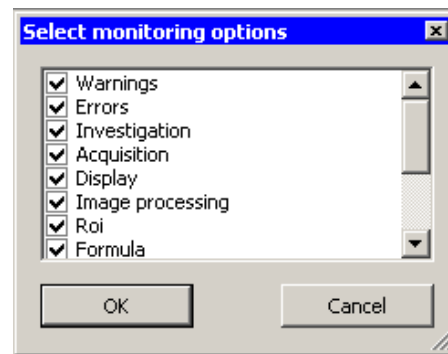
Monitor and Reporter

The Monitor

The Monitor displays runtime information, such as warnings, errors, and execution times. To display the Monitor, select View **View→Monitor** from the main menu.



You can select for display only the information that is of interest to you.



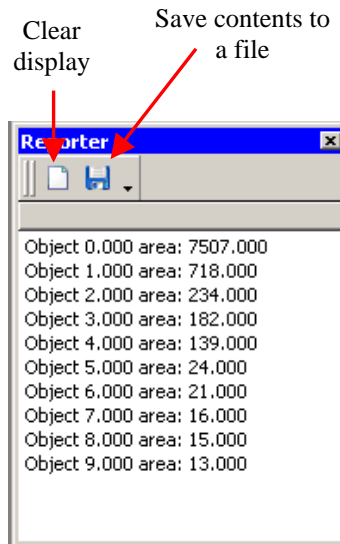
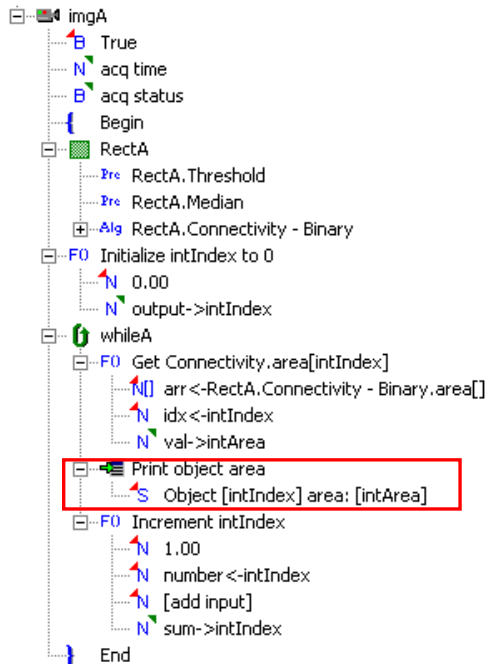
Observing execution times can sometimes aid you in finding an application bottleneck.



Updating the Monitor can add several milliseconds to the investigation execution time. The Monitor should not be displayed when an investigation is deployed.

The Reporter

The Reporter is a text window to which you can write with the **IO:Reporter Print** instruction. To display the Reporter, select **View→Reporter** from the main menu.



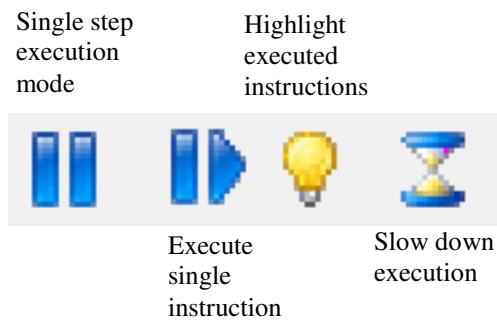
The area of each object found by the Connectivity algorithm is printed to the Reporter from within the While loop. The **IO:Reporter Print** instruction writes a single line followed by a carriage return and line feed.

When defining a string to print, surrounding a variable name with square brackets [] means that the value of the variable is printed. You cannot print readings, only variables.

Debugging

When developing and testing an investigation, it is sometimes desirable – or necessary – to step through the program a line at a time, or at least very slowly, to find the source of a problem.

Combinations of Sherlock's debug options, breakpoints, and execution options allow you to run an investigation in various debug modes.



Highlight executed instructions mode


Click the  **Highlight executed instructions** button.

Click either the  **Run once** or  **Run continuously** button.

The investigation runs at close to normal speed, but instructions are highlighted as they are executed.


Slow mode

Click the  **Slow down execution** button.

Click the  **Run once** or  **Run Continuously** button.

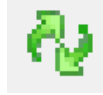
The investigation is executed at slow speed. It is much easier to see the update of the **Watch**, **Variables**, **Monitor**, and **Reporter** panes.

Single step mode

Click the  **Single step execution mode** button.



Click the **Run once** or



Run Continuously button.

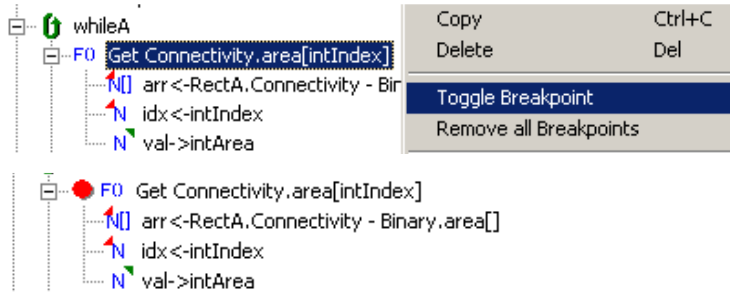


Repeatedly click the **Execute single instruction** button to execute one line at a time, or click the **Single step execution button mode** to resume normal execution.

Breakpoints

A breakpoint marks an instruction at which execution pauses when reached.

To add a breakpoint, right-click the instruction and select **Toggle Breakpoint** on the pop-up menu.



A breakpoint is marked with a red dot.

When you click the **Run once** or **Run continuously** button, the program executes until the first breakpoint. The investigation is automatically put into single step mode, and the **Run once** and **Run continuously** buttons are disabled. To continue execution to the next breakpoint, click the **Single-step execution mode** button. To execute one instruction at a time, click the **Execute single instruction** button.

To remove a breakpoint, right-click the instruction again and select **Toggle Breakpoint**, or click **Remove all Breakpoints**.



Breakpoints are not saved when you close the investigation.

Ending debug

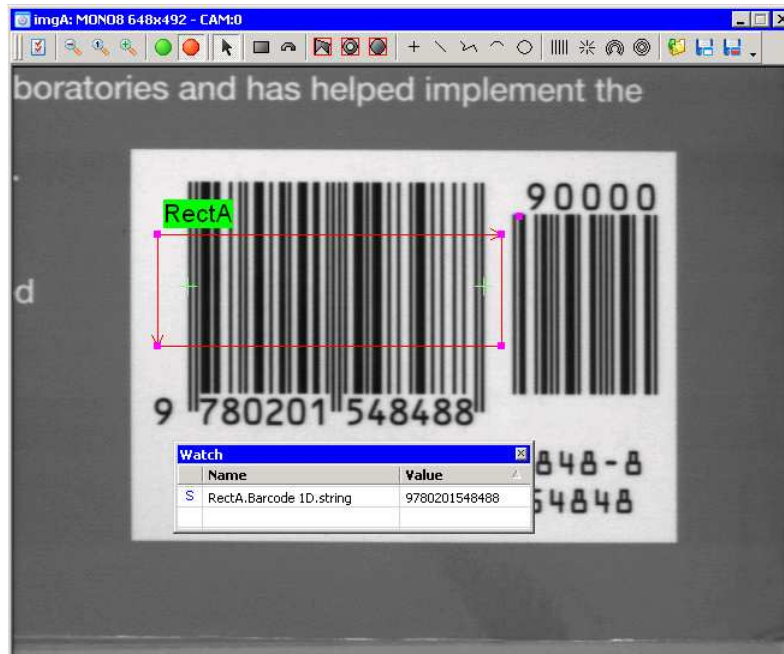


To end program execution while in the middle of a debug session, click the **Abort** button.

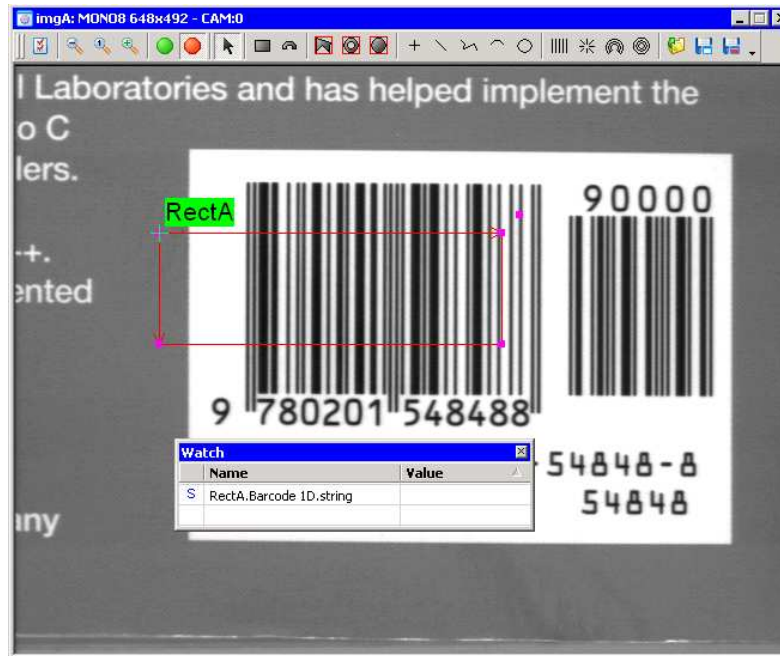
Landmarks and alignment

A typical machine vision application requires the analysis of the same features in a series of images. Because of the nature of motion control and image acquisition equipment, or of the target object itself, the features to be analyzed are sometimes in a slightly or considerably different place in each image. An ROI simply analyzes the pixels inside it; there is no way for it to “know” whether they’re the right pixels, so a shift or rotation of the object to be analyzed can result in a “false bad” result.

Here a rectangle ROI is positioned to read the barcode on a book with the **Barcode 1D** algorithm.

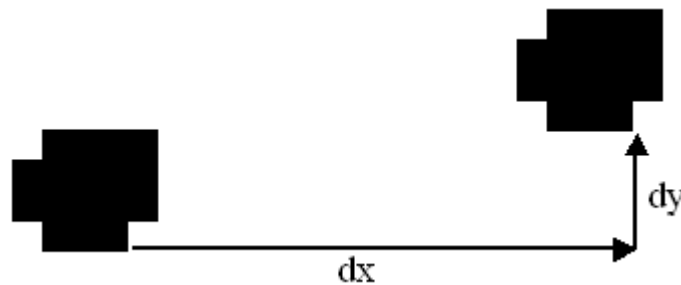


If the book shifts horizontally from acquisition to acquisition, the barcode may not be within the ROI, and therefore may not be read.

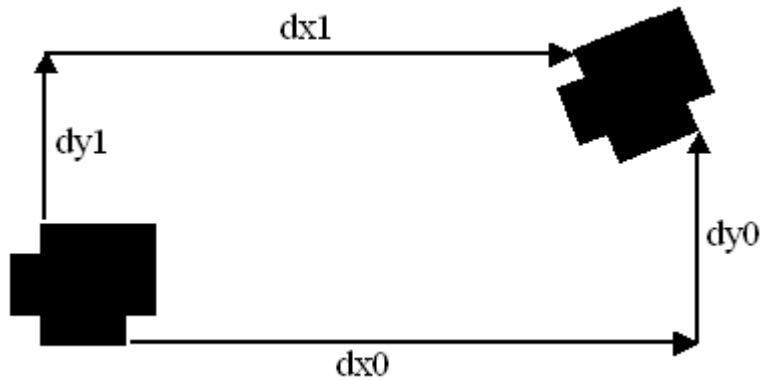


To ensure that ROIs analyze the correct pixels, you can create an alignment scheme to move the ROIs relative to one or more points that can be found reliably in every image.

If the pixels to be analyzed can shift only horizontally and/or vertically, one point is sufficient to measure the amount of shift.

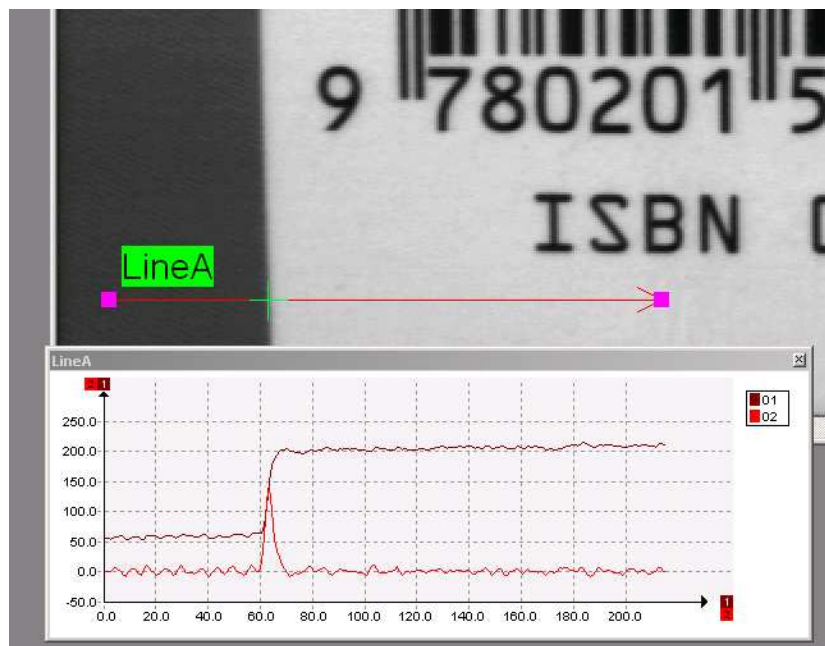


If the pixels to be analyzed can rotate as well as shift, two points are necessary to calculate the degree of rotation.

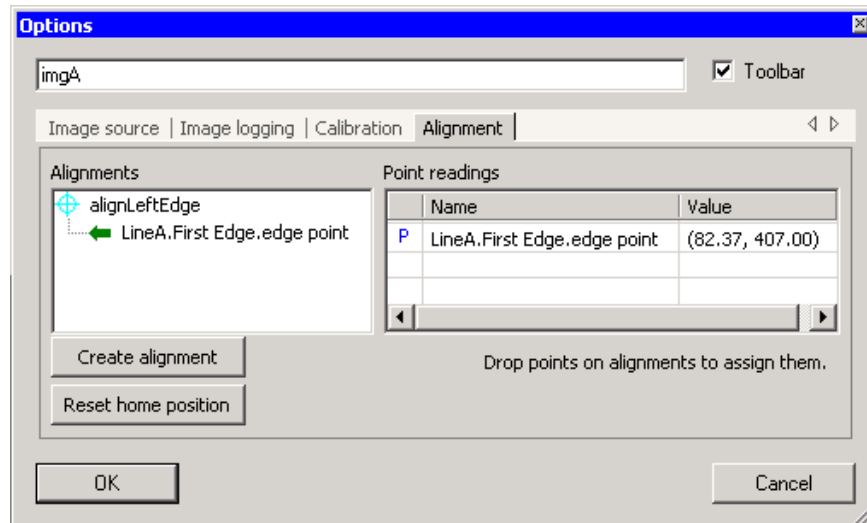


The first step in creating an alignment scheme is to identify the point or points – landmarks – that can be found in every image.

In the book image, the left edge of the barcode label can be found with a line ROI executing the **First Edge** algorithm. 100 was chosen for the algorithm's **edge strength** parameter after analyzing the magnitude and direction of the change in the pixel intensity at the label boundary in the ROI's **Profile** display.



An alignment scheme is created on the **Alignment** page of an Image Window's **Options** dialog.

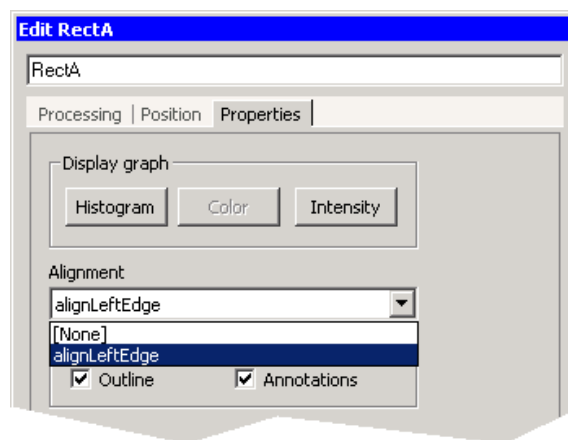


Click the **Create alignment** button to create an empty alignment scheme. By default, alignment schemes are named **alignmentA**, **alignmentB**, etc. Here the alignment scheme was renamed **alignLeftEdge**.

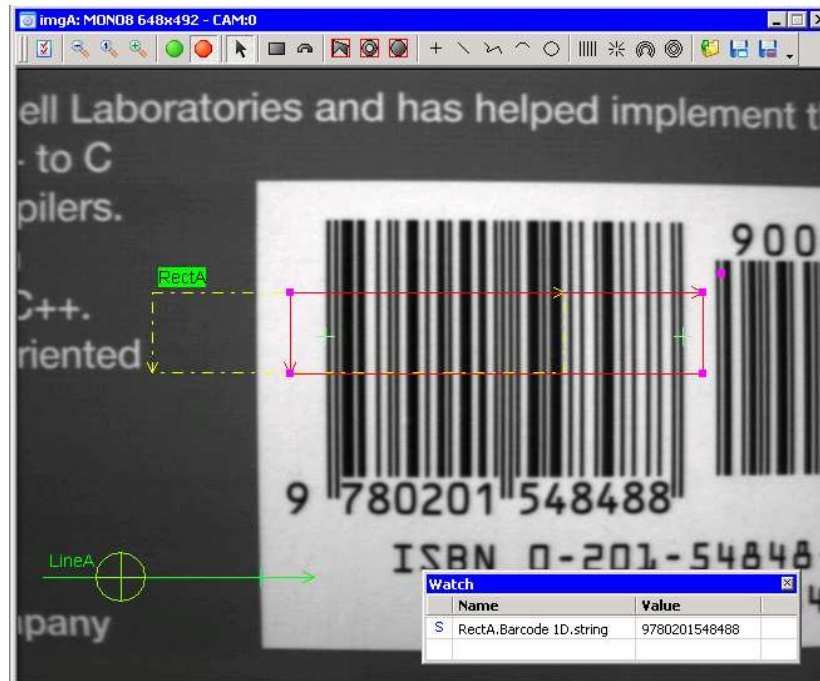
All single point readings are candidates for inputs to alignment schemes. The point found by the **First Edge** algorithm was dragged from the **Point readings** window to **alignLeftEdge** to make it a landmark

An alignment scheme by itself doesn't do anything other than determine the displacement of landmark points from their original positions; the alignment must be applied to an ROI so that the ROI moves relative to the landmark and analyzes the correct pixels.

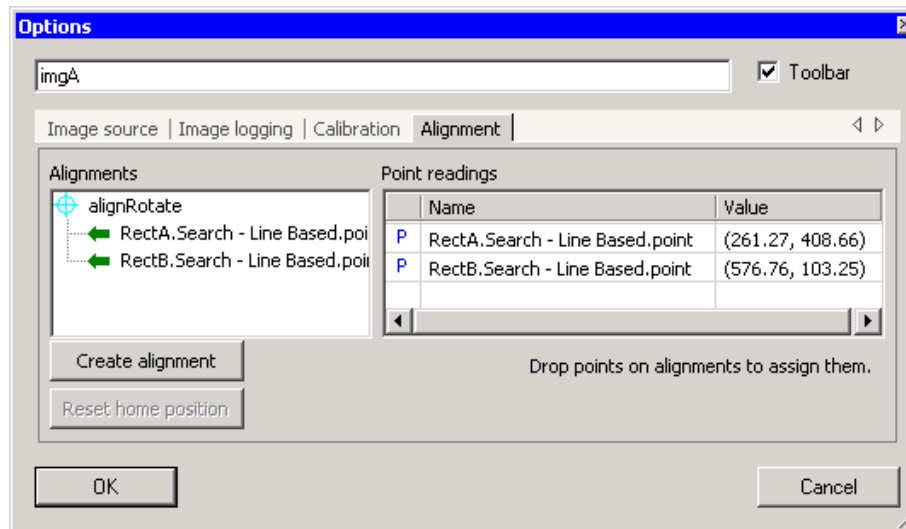
To apply an alignment to an ROI, open the ROI's **Edit** dialog and click the **Properties** tab. The Alignment drop-down list shows all the alignments in the investigation.



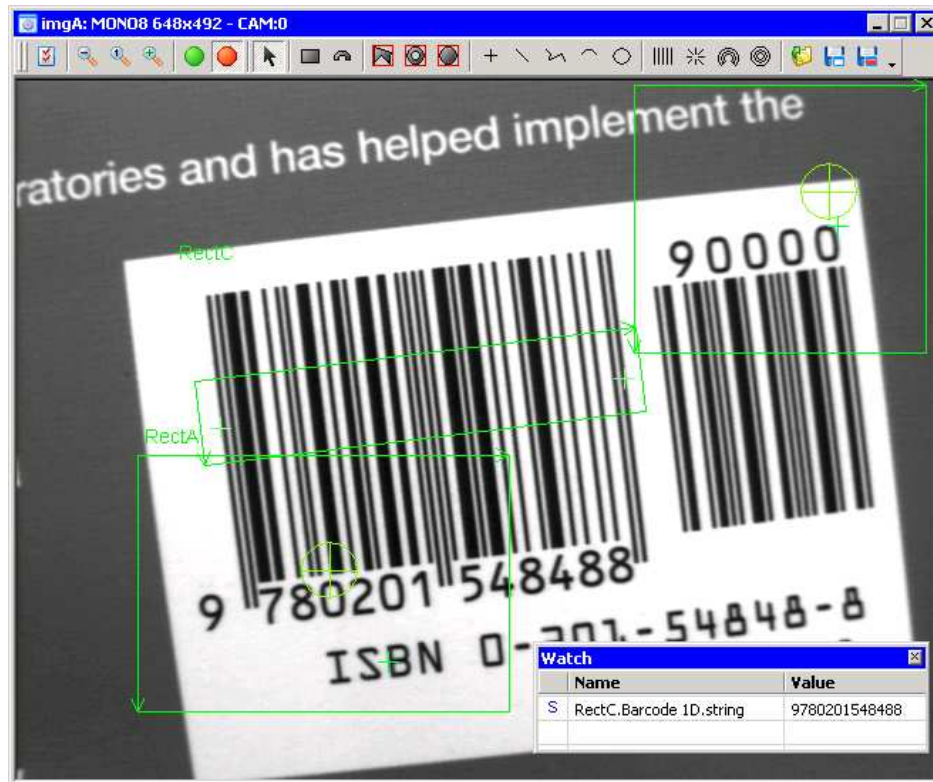
Now if the book shifts horizontally, the line ROI's **First Edge** algorithm finds the point at the left edge of the barcode label and calculates its displacement from its original position. The rectangle ROI executing the **Barcode 1D** algorithm shifts relative to this landmark. The dashed lines show the original position of the rectangle ROI.



If the object can rotate, a two-landmark alignment scheme is necessary. On the book, “ISBN” and the ‘90000’ above the second barcode were trained as patterns in two rectangle ROIs executing the **Search – Line Based** algorithm. The **point** readings from the two algorithms were turned into landmarks by adding them to **alignRotate**.



The ROI executing the **1D Barcode** algorithm was applied to the **alignRotate** alignment. Now if the book rotates, the barcode ROI rotates relative to the two landmarks, as long as the trained patterns do not move outside their search ROIs.



It is possible to have different alignment schemes applied to different ROIs within the same image window.

Calibration

By default, Sherlock returns point-to-point measurements in pixels.

To measure the distance between the outermost edges of the cutouts, a line ROI executing the **Outside Caliper** algorithm was created. The distance is 384.94 pixels



To return measurements in real-world units, an image window must be calibrated to correlate pixels in the image window to real-world locations.

Calibration method 1: points

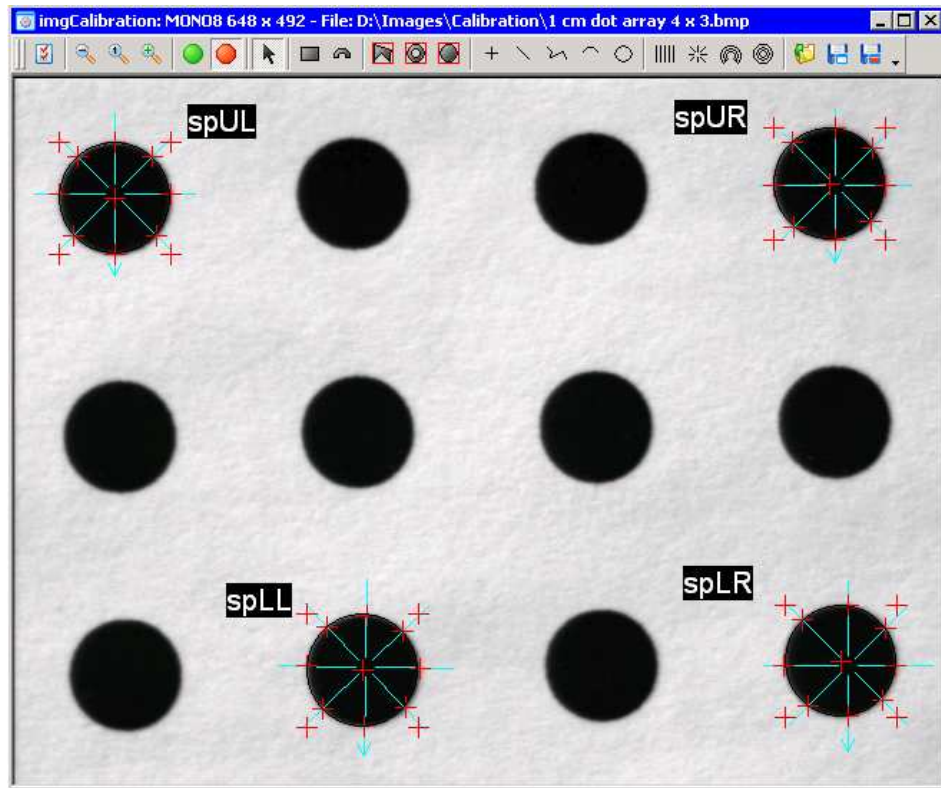
1. Identify four points in an image that are known real-world distances from each other
2. Create a transform that correlates the image window points to real-world locations
3. Create a calibration object and fill it with the transform
4. Apply the calibration object to an image window



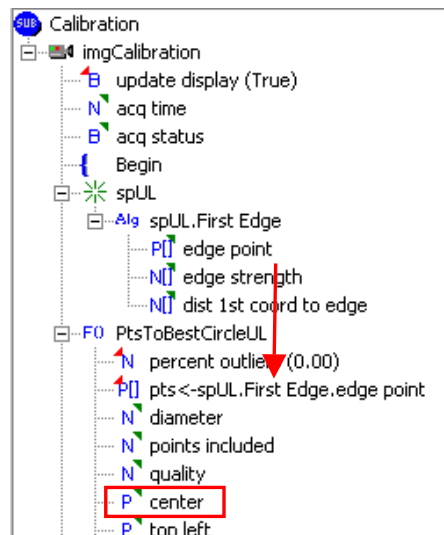
Since you generally need to calculate the transform only occasionally, programmatic creation of a transform is usually performed in a subroutine for which the **Execute on calibration** option is selected. A subroutine with this option selected is executed when the F11 key is struck. In the following example, the subroutine **Calibration** is marked for **Execute on calibration**.

1. Identify four points

In this image window, four spoke ROIs (spUL, spUR, spLL, spLR) executing the **First Edge** algorithm are positioned over four circles on a calibration target. In this target, the distance between the centers of horizontally or vertically neighboring circles is 10 millimeters.



The array of edge points found by each spoke ROI is passed as input to a **Geometric:PtsToBestCircle** instruction. One of the readings from this instruction is the **center** point of the calculated circle.



2. Correlate the four points

The four **center** readings returned by the four **PtsToBestCircle** instructions are passed as inputs to the **IO:Calibration:Calibrate Using Points** instruction. The corresponding real-world locations of the center points are entered as inputs in the same order as the center points.

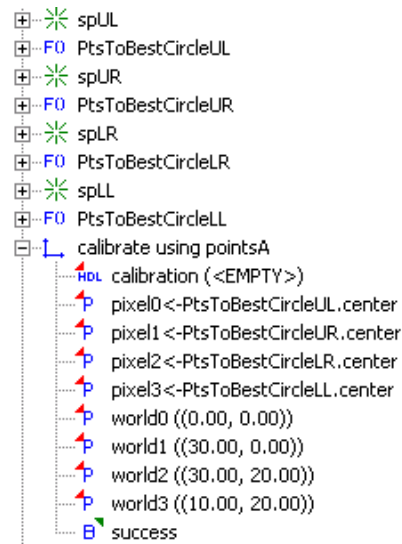
Upper-left circle center: (0.00, 0.00)

Upper-right circle center: (30.00, 0.00)

Lower-right circle center: (30.00, 20.00)

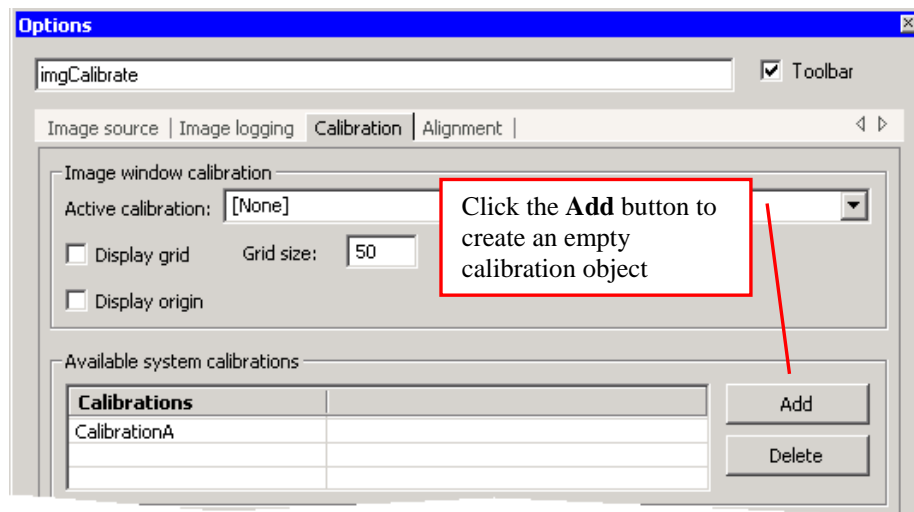
Lower-left circle center: (10.00, 20.00)

The unit of measurement is not specified.

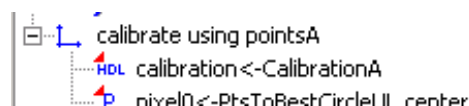


3. Create and fill a calibration object

An empty calibration object can be created on any image window's **Options** dialog **Calibration** tab.



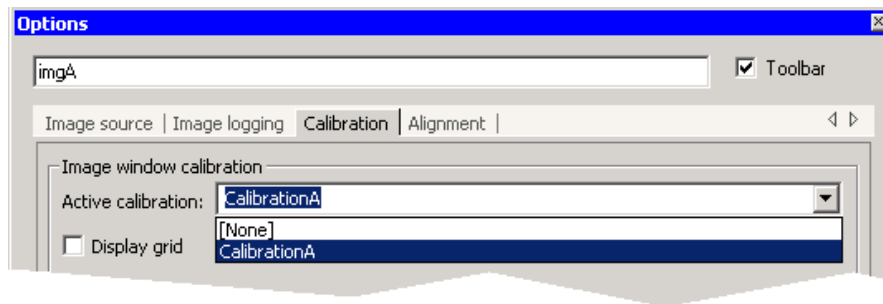
In the **Calibrate Using Points** instruction, double-left click the **HDL <Empty>** input to display the handles of the calibration objects. Select an object (**CalibrationA**) to save the calibration transform to.



Since the subroutine hasn't been executed yet, the transform hasn't been created, and the calibration object is still empty. To create the transform and fill the calibration object, execute the subroutine by hitting the F11 key.

4. Apply the calibration object

Calibration objects are applied to image windows individually. On an image window's **Options** dialog **Calibration** tab, select the calibration object from the drop-down list.



Now when point-to-point measurements are made in the calibrated image window, they are returned in real-world distances. The distance between the outermost edges of the cutouts is 23.09 millimeters.



Calibration method 2 : transformation values

1. Determine the origin and scaling factors of the image
2. Create a transform using the origin and scaling factors from step 1
3. Create a calibration object and fill it with the transform
4. Apply the calibration object to image windows

This method requires that you know or can calculate the ratios of horizontal and vertical pixel distances in the image window to distances in real-world units; these are supplied as scaling factors to the **IO:Calibration:Calibrate Using Transformation Values** instruction.

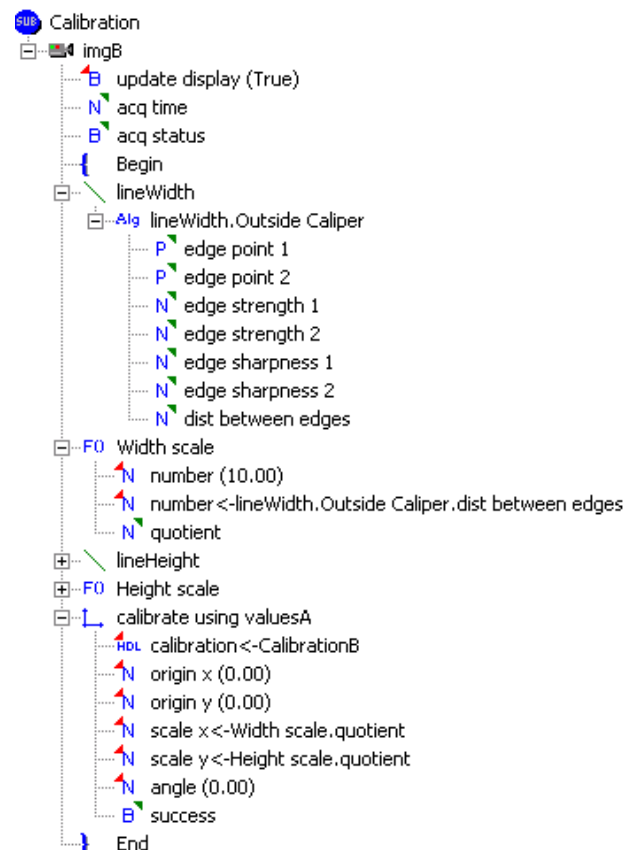


1. In this **Calibration** subroutine, the horizontal scale factor is calculated by dividing the real-world width of one of the squares (10 mm) by the pixel distance returned by the **lineWidth** ROI executing the **Outside Caliper** algorithm. The same is done for the vertical scale factor. The scale factors are 0.06 in both directions.

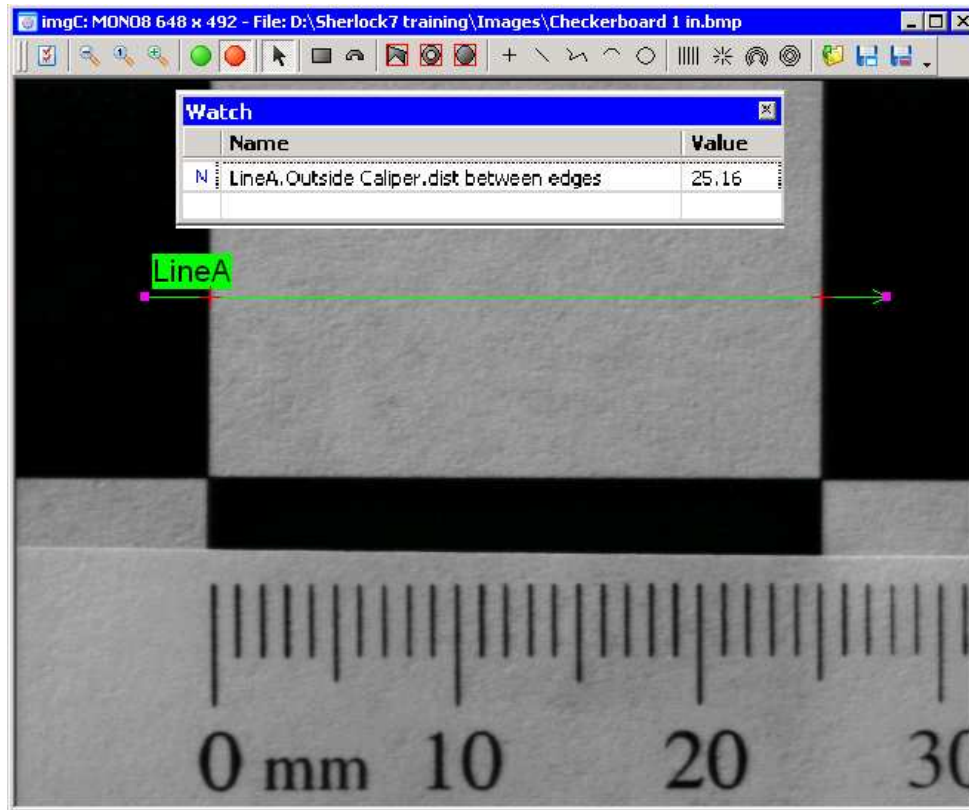
Watch		
	Name	Value
N	lineWidth.Outside Caliper.dist between edges	164.06
N	lineHeight.Outside Caliper.dist between edges	162.78
N	Width scale.quotient	0.06
N	Height scale.quotient	0.06

2. The scale factors are passed as inputs to the **Calibrate Using Transformation Values** instruction. The origin is left at (0,0), the upper-left corner of the image; no rotation is calculated.

3. The calibration object **CalibrationB** is filled with the transform.

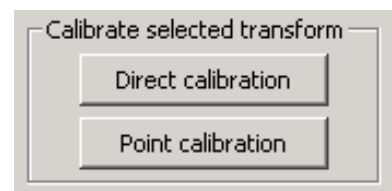


4. After the **Calibration** subroutine is executed by hitting the F11 key, point-to-point distance measurements made in another image window to which the calibration transform **CalibrationB** is applied are returned in calibrated units.



Manual calibration

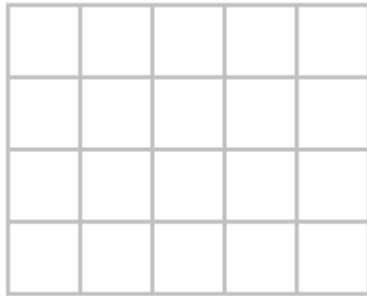
The calibration methods just described can also be performed manually, by clicking the **Direct calibration** (transformation values) or **Point calibration** (points) button on the image window's **Options** dialog **Calibration** tab.



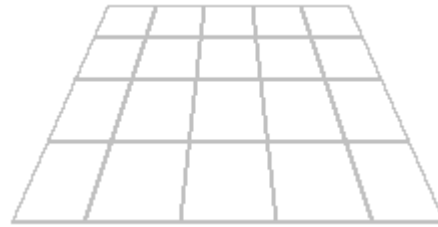
Area measurements, such as the **area[]** reading returned by the **Connectivity – Binary** algorithm, are always returned in pixels.

Calibration using a grid

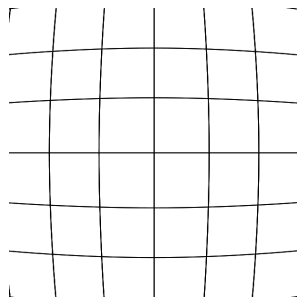
Camera placement and optics (lenses) can introduce distortion that makes image analysis, particularly point-to-point measurement, unreliable at best, and impossible at worst.



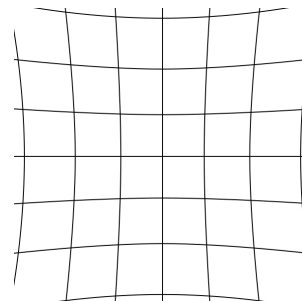
no distortion



perspective (skew) distortion



barrel distortion (from lens)



pincushion distortion (from lens)

The four-point calibration method can correct for perspective distortion (skew), which results when the camera's imaging plane is not parallel to the inspection plane. It cannot correct for the kind of distortion introduced by most lenses, especially at their extreme edges, or by other optical irregularities. (Calibration using transformation values ["scale x" and "scale y"] cannot correct for any type of distortion.)

Grid calibration can be used to convert from pixels to real-world units (millimeters, inches, etc.), or to remap pixels so that features appear as they would if there were no distortion introduced by camera placement or optics.

The calibration algorithms

The grid calibration utility includes four algorithms, each optimized for a particular type of distortion. **Arbitrary or random distortion** refers to distortion that is not the same throughout the image. **Radial distortion**, which is introduced to some extent by most lenses, is the same along any line drawn from the center of the image (the center of the lens) to the edge. Pincushion and barrel ("fisheye") distortion are examples of radial distortion.

- **Perspective** corrects for perspective distortion only; it does not correct for arbitrary or radial distortion.
- **Piecewise Bilinear** corrects for any kind of distortion using a localized technique.
- **Piecewise Perspective** corrects for any kind of distortion using a localized technique. Unlike **Piecewise Bilinear** the local approximation is based on perspective instead of linear interpolation. This method is adequate for perspective distortion combined with another kind of distortion (for example, radial).

- **Radial** corrects for radial distortion only; it does not correct for arbitrary or perspective distortion.

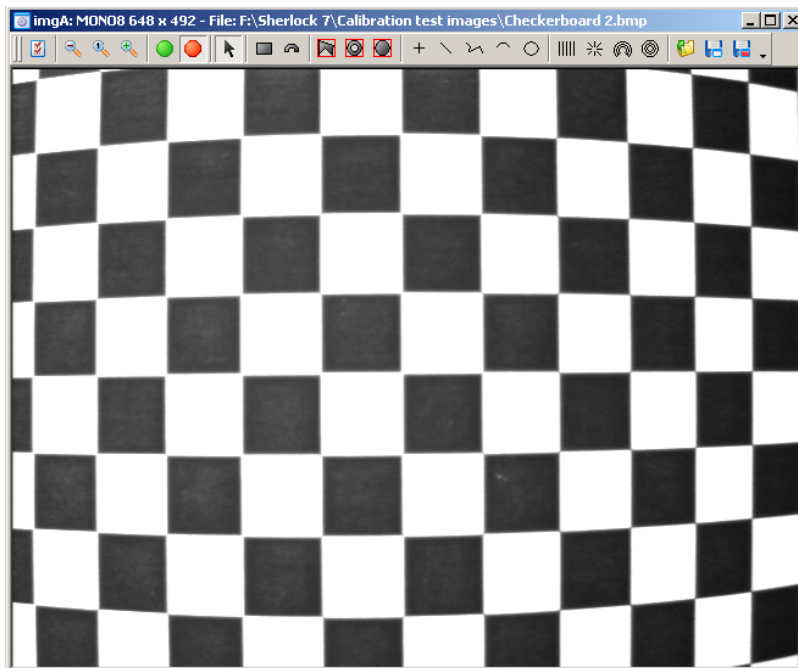
The calibration target

To calibrate using the grid method, you need one of the following targets:

- A black-and-white checkerboard in which the sizes of the squares are known and uniform
- A black-on-white or white-on-black line square grid in which the sizes of the squares created by the grid are known and uniform
- An array of black-on-white or white-on-black circular spots in which the spots' center-to-center distances are known and uniform

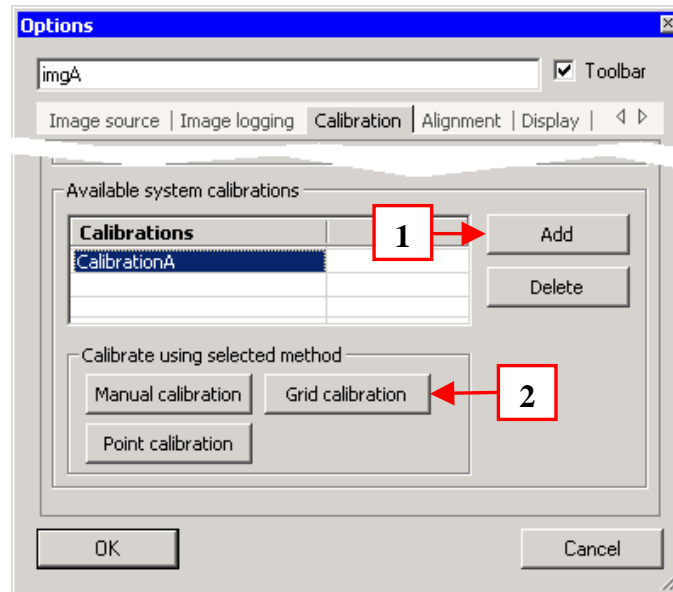
By finding these feature points, and knowing that the original physical location of the feature points must lie on a uniform grid with square cells, complex distortions can be corrected with minimal user input.

This image of a checkerboard target exhibits arbitrary distortion, especially at the left and right sides. There is no perspective distortion due to camera placement, nor is the distortion radial.

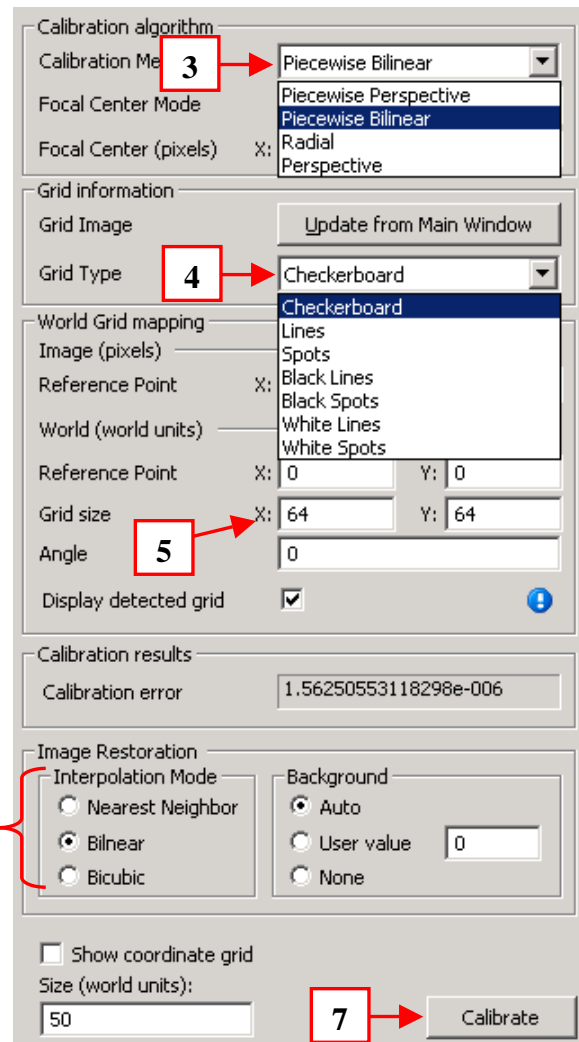


Once you have acquired an image of the target under the optical conditions in which processing will take place, click anywhere in the image window to display its **Options** dialog. Click the **Calibration** tab.

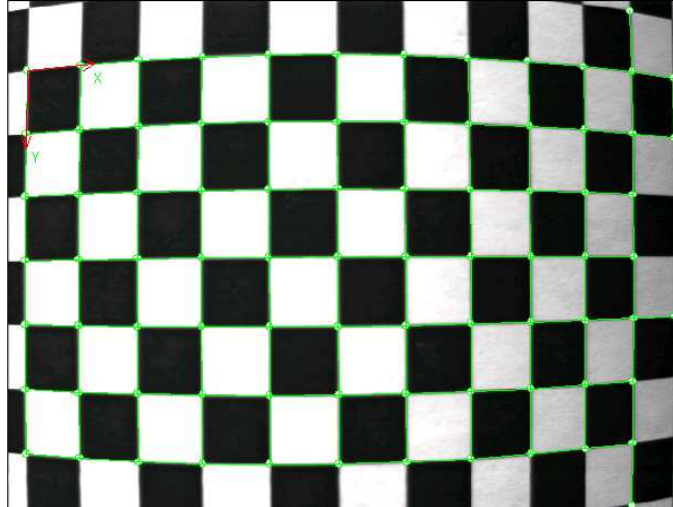
1. Click the **Add** button to create a new calibration object. You can rename the object, or leave its name as is.
2. Click the **Grid calibration** button.



3. Select the algorithm appropriate for the type of distortion you are correcting for.
4. Select the type of target you are using. If you select **Lines** or **Spots**, Sherlock will determine whether they are dark-on-light or light-on-dark.
5. If you want to redraw the image (remap the pixels), enter the grid size **in pixels** – that is, the width and height of the squares for a checkerboard or line grid; or the center-to-center distance for spots. If you want to convert from pixels to real-world units, enter the grid size in real-world units.
6. Select the method of image restoration you want to use. (Image restoration is performed only if you select it in step 8b.)
7. Click the **Calibrate** button. Calibration will take several seconds.

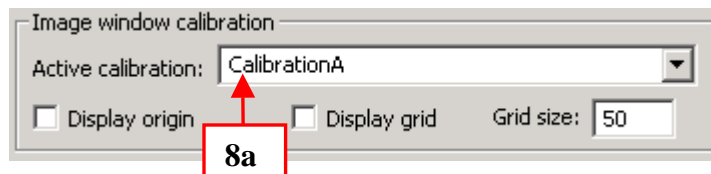


If **Display detected grid** is checked, the grid connecting the spots or running along the edges of the squares or grid lines is displayed.



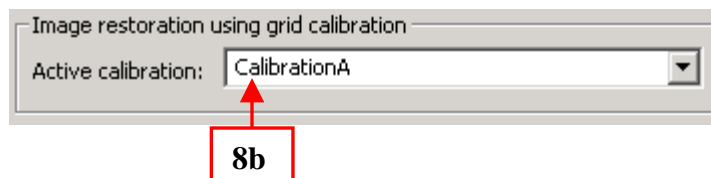
Close the calibration dialog. On the **Calibration** page of the image window's **Options** dialog, select how you want to use the calibration object.

8a. If you want to convert from pixels to real-world units, select the calibration object for the **Image window calibration active calibration**.



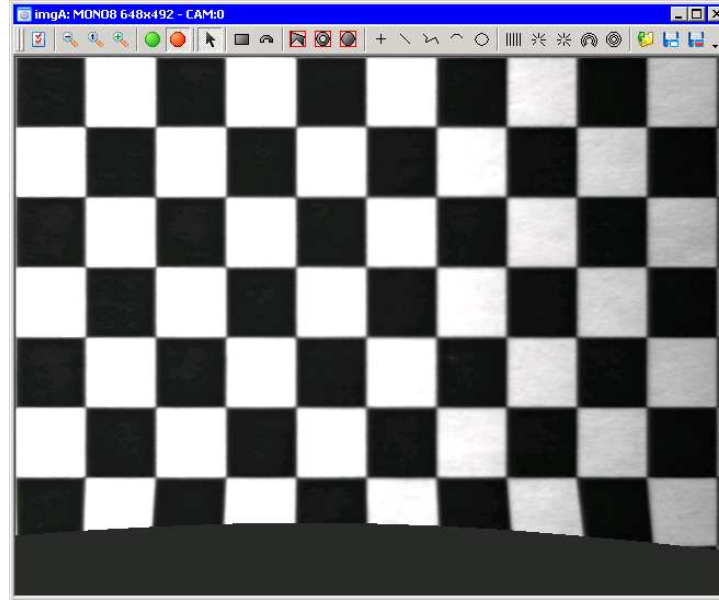
OR

8b. If you want to remap the pixels, select the calibration object for the **Image restoration active calibration**.



A single calibration object cannot be used to perform both image correction and pixel-to-real-world conversion. Selecting the same calibration object for both of the calibration options will lead to unexpected (and undesirable) results.

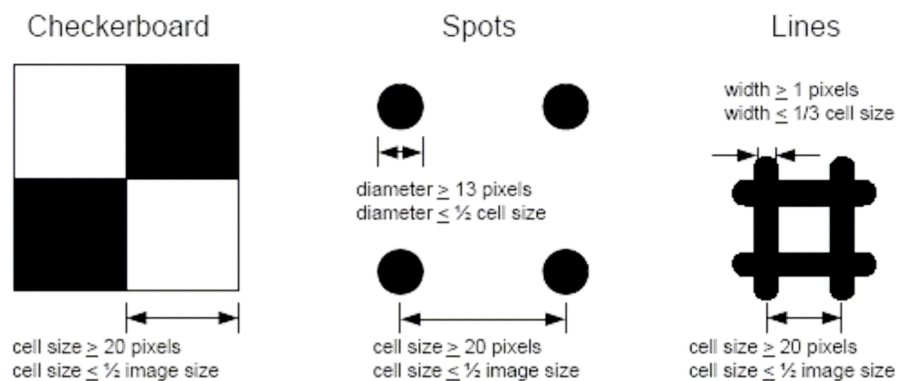
If you created a calibration object to remap the pixels, you can run the investigation to see the corrected image.



Minimal target dimensions

The minimum number of grid points is four, which is acceptable for pure perspective distortion (no arbitrary or radial distortion), as long as the four points form a rectangle. However, the greater the number of grid points, the more accurate the calibration.

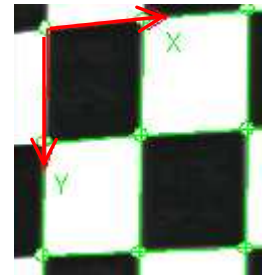
The checkerboard and the line grids result in the most accurate calibrations. The picture below shows the range of acceptable dimensions for the pattern elements.



The different calibration methods – Perspective, Piecewise Bilinear, Piecewise Perspective, and Radial – take different amounts of time to generate the contents of a calibration object. But once the contents have been generated, they all take the same amount of time to apply to an image at run time. The different image restoration methods – Nearest Neighbor, Bilinear, and Bicubic – take increasing amounts of time to remap the pixels, but are also increasingly accurate.



- For the checkerboard or line calibration targets, the algorithms require a 10-pixel buffer between any point on an edge and the edge of the image, for the square to be used in the calibration. For example, if the top edge of a square in a checkerboard is only 8 pixels from the top edge of the image, the square will not be used to create the correction map, even though the entire square is in the image
- Grid calibration corrects the source image starting at the point designated as the origin in the **detected grid** display in the calibration dialog.
By default, this is the upper-leftmost spot center or square corner in the grid; it is marked with two red arrows. This origin becomes the (0,0) pixel of the corrected image; pixels to the left and above the **detected grid** origin are “mapped out” of the corrected image




JavaScript

JavaScript is a scripting language developed by Netscape to enable Web authors to build interactive sites. Although it shares many of the features and structures of the full Java language, it was developed independently. **JavaScript is not Java.**

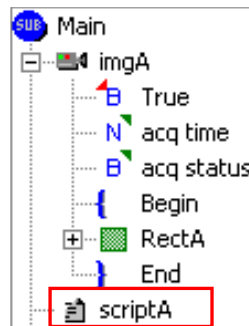
In a Sherlock investigation, a JavaScript code module can be used to perform tasks that “native” Sherlock code either cannot perform easily, or cannot perform at all.

Adding a JavaScript module

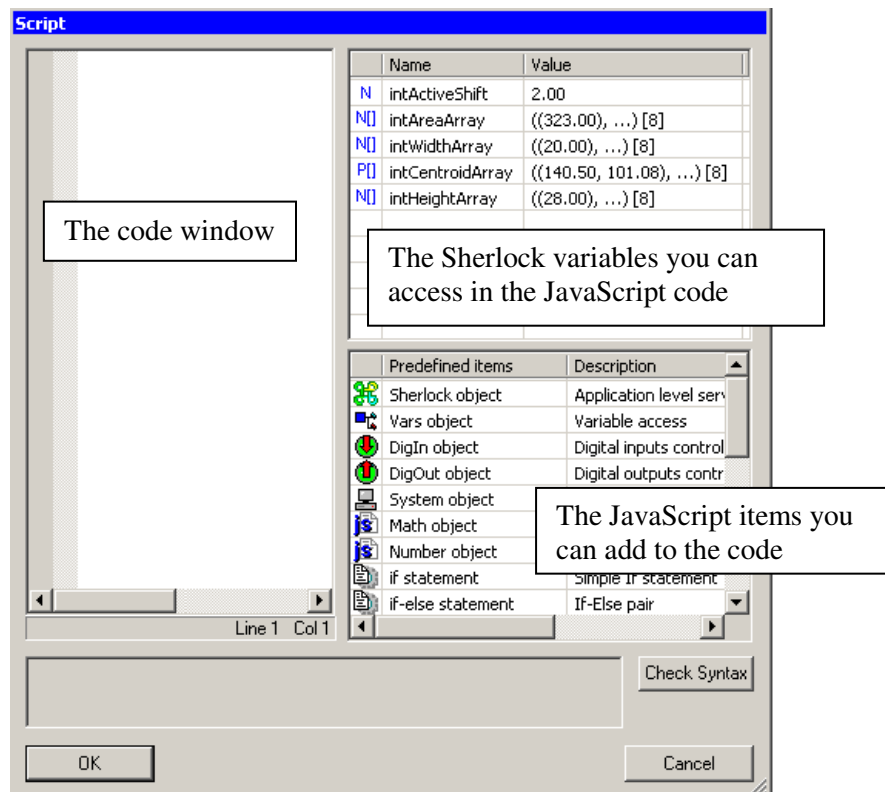
To add a JavaScript module to a program, click the **Create script** button  on the Program toolbar.

The module is added to the program after the element that had the focus when you clicked the **Create script** button.

Double-left-click on the module name in the program to display its edit dialog



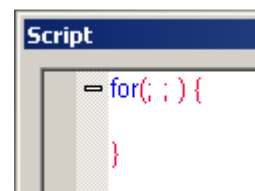
The JavaScript edit dialog



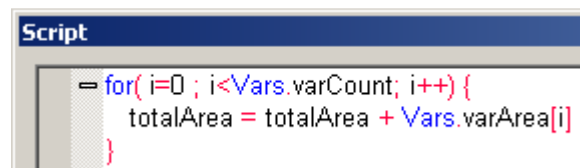
The JavaScript language includes many common programming constructs – **For**, **While**, **Do/While**, **If**, **If/Else**, **Switch**, etc. There are also several objects with built-in properties and methods – **Sherlock**, **Vars**, **DigIn**, **DigOut**, **System**, **Math**, **Number**, and **Date**.

To add code to the code window, you can either type it, or drag-and-drop an item from the **Predefined items** or Sherlock variables window.

When you drag-and-drop a programming construct from **Predefined items** into the code window, the construct's "skeleton" is created.



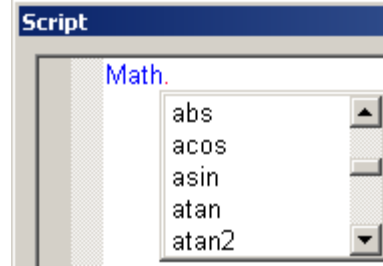
You then fill in the details.



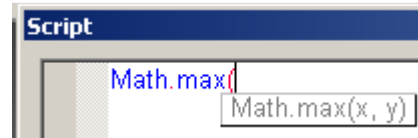
Using JavaScript objects

JavaScript is an object-oriented language. A JavaScript object encapsulates members and methods for different types of operations.

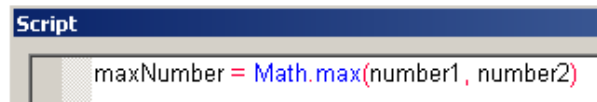
When you drag-and-drop an object from **Predefined items** and type a period (.), the object's members and methods are displayed in a drop-down list.



After you select an object's method and type a left parentheses, the method's parameters are displayed. You fill in the parameters to complete the method.



Here, the **max** method of the **Math** object returns the larger of the two numbers contained in the variables *number1* and *number2*. The value is stored in the variable *maxNumber*.



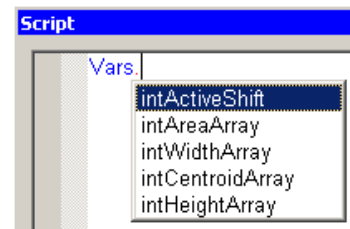
Variables

Sherlock variables

The main method of communication between Sherlock and JavaScript is through Sherlock variables. JavaScript cannot call Sherlock subroutines, manipulate ROIs, call instructions, or in any other way access a Sherlock investigation (except for a few methods described later).

Sherlock variables are accessed from the JavaScript **Vars** object. For example, a Sherlock variable *intActiveShift* would be accessed as *Vars.intActiveShift*.

If you type **Vars.** (with the period) in the code window, all of the variables in the Sherlock investigation are displayed in a drop-down list.



If you drag-and-drop **Vars** from the **Predefined items** list to the code window, then type a period, the same drop-down list is displayed.

If you drag-and-drop a variable from the variable window (the JavaScript variable window, not the Sherlock **Variables** window), **Vars.** is automatically prepended.

Point variables

A Sherlock point variable is accessed through two properties, **.x** and **.y**. To read the x and y values of a single Sherlock point variable *Point* in JavaScript, use *Vars.Point.x* and *Vars.Point.y*. To read the x and y values of one element in a point array variable *Points*, use *Vars.Points[array_index].x* and *Vars.Points[array_index].y*.

Line variables

A Sherlock line variable is accessed through two properties, **.a** (angle) and **.d** (distance). To read the angle and distance components of a single Sherlock line variable *Line* in JavaScript, use

Vars.Line.a and *Vars.Line.d*. To read the angle and distance values of one element of a line array variable *Lines*, use *Vars.Lines[array_index].a* and *Vars.Lines[array_index].d*.



A JavaScript module has access to Sherlock variables only – not readings from algorithms.

JavaScript variables

JavaScript variables do not usually have to be declared. A variable's type and size are determined at runtime based on context.

The 'Script' window displays the following code:

```
jsNumber = Vars.varNumber
jsNumbers = Vars.varNumbersArray

jsPointX = Vars.varPoint.x
jsPointY = Vars.varPoint.y

jsPointArray = Vars.varPointArray
```

To the right, a table shows the current state of variables:

	Name	Value
N	varNumber	321.000
N[]	varNumbersArray	((3.142), ...) [3]
P	varPoint	(123.000, 456.000)
P[]	varPointArray	((34.700, 263.880), ...) [2]

When you assign a Sherlock array variable to a JavaScript variable, the type and array size of the JavaScript variable are automatically set, as in *jsPointArray = Vars.varPointArray* above.

Because variables are dynamic in JavaScript, there is no type checking when passing values to Sherlock. Assigning the wrong type of JavaScript variable to a Sherlock variable will not generate a syntax error, but will generate a runtime error.

The 'Script' window displays the following code:

```
jsString = "Hello"
Vars.varNumeric = jsString
```

The variable table shows:

	Name	Value
N	varNumeric	0.00

The message area says "No errors". Buttons for "Check Syntax", "OK", "Run", "Stop", and "Cancel" are visible.

A JavaScript variable's type will change if you assign it different types of data. In the following code, the JavaScript variable *jsVariable* is first a number, then a string, then an array of points, and finally a number again. (But of course this is not good coding practice!)

The 'Script' window displays the following code:

```
jsVariable = 23.9
jsVariable = "Hello"
jsVariable = Vars.varEdgePoints
jsVariable = Vars.varIndex
```

The variable table shows:

	Name	Value
N	varIndex	0.000
P[]	varEdgePoints	((23.000, 55.000)...

One of the few variables you must declare is an array that is not created by assigning a Sherlock array to it. You do not have to declare the size of an array; arrays are automatically resized as necessary.

```
Script
// Copy Sherlock array to local array
jsNumberArray = Vars.varNumberArray

// Create array of undetermined size
var jsSmallNumbers = new Array()

k = 0

// The .length method returns the size of the array
for( i=0 ; i<jsNumberArray.length; i++ ) {
    // Copy only small ( < 100 ) numbers to new array
    if ( jsNumberArray[i] < 100 ) {
        jsSmallNumbers[k] = jsNumberArray[i]
        k++
    }
}
```

All JavaScript variables are local to the module in which they appear. They are destroyed when the code module is exited; their values are not maintained between calls to the module.

JavaScript variable names are case sensitive. *jsNumber* is not the same variable as *JSnumber*.

If you type a Sherlock variable name in the JavaScript code window without using the **Vars** object, you create a local JavaScript variable with the same name as the Sherlock variable, **but they are not the same**.

This sets a local JavaScript variable *varTotal* to 168.

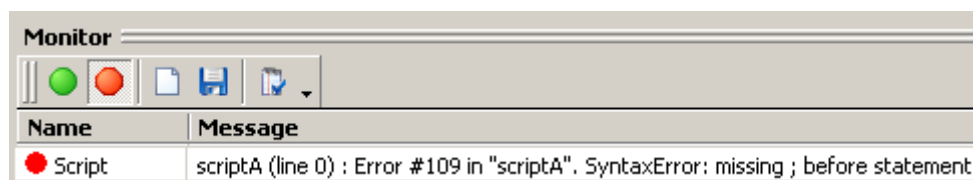
```
varTotal = 123 + 45
```

	Name	Value
N	varTotal	0.000

This sets the Sherlock variable *varTotal* to 3. The local JavaScript variable *varTotal* is still set to 168.

```
Vars.varTotal = 33/11
```

JavaScript cannot recognize variables with embedded spaces. For example, *varTotal Height* is a valid Sherlock variable; it will appear in the JavaScript **Vars** object as *Vars.varTotal Height*. But at runtime the JavaScript engine will generate an error:



(The engine will generate the same error in the JavaScript edit dialog if you click the **Check Syntax** button.)

The Sherlock object

The Sherlock object writes to the Monitor, the Reporter and IpeStudioLog.txt. The three methods that write to these – **.Monitor**, **.Reporter**, and **.DbgLog**, respectively – all take a single parameter, the string to write.

```
Sherlock.Monitor(  
    Sherlock.Monitor(String message)
```

When defining **String**

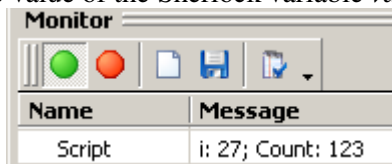
- To write a literal string enclose it in quotes (" ")
- To write a Sherlock variable, use the **Vars** object
- To write a JavaScript variable, enter its name without quotes

You can concatenate literal strings and variables in any combination with the plus sign "+".

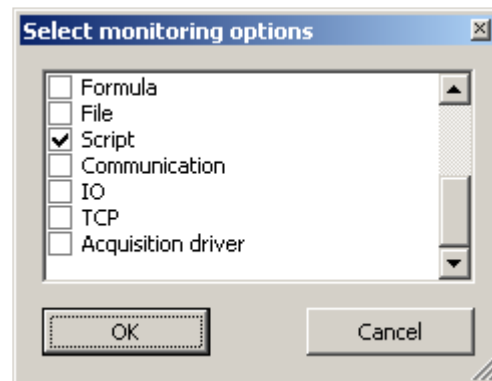
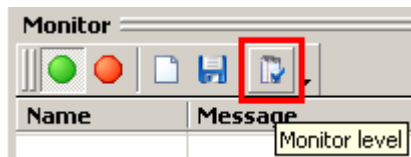
For example

```
Sherlock.Monitor("i: " + i + "; Count: " + Vars.varCount)
```

writes the literal string "i: ", the value of the JavaScript variable *i*, the literal string "; Count: ", and the value of the Sherlock variable *varCount*.



Sherlock.Monitor writes to the Monitor only if the **Script** option is enabled in the **Select monitoring options** dialog, which is displayed when you click the **Monitor level** button on the Monitor's toolbar.



Sherlock.DbgLog writes to <Sherlock>\bin\IpeStudioLog.txt only if the logger level in <Sherlock>\bin\IpeLog.config is set to DEBUG. If the logger level is set to ERROR, WARNING or INFO, **Sherlock.DbgLog** is ignored.

```
log4j.logger.ipeStudio.logger=DEBUG, RolFile
```

Sherlock.Reporter always writes to the Reporter, even if it is closed.

The DigIn and DigOut objects

The **DigIn** object reads the digital inputs

The **DigOut** object reads and writes the digital outputs.

The System Object

The **System** object can manipulate text files (Open, Close Read, Write, Delete, Move, etc.) and folders (Create and Delete), and pause the script for a number of milliseconds (Sleep).

You must create hierarchies of folders one level at a time.

For example, if neither Folder1 nor Folder2 exists, these two statements will create them:

```
System.FolderCreate("D:\\Folder1")
System.FolderCreate("D:\\Folder1\\Folder2")
```

This statement alone will not:

```
System.FolderCreate("D:\\Folder1\\Folder2")
```

You can delete a folder at any level of the hierarchy.

For example, given the hierarchy D:\\Folder1\\Folder2\\Folder3, the statement

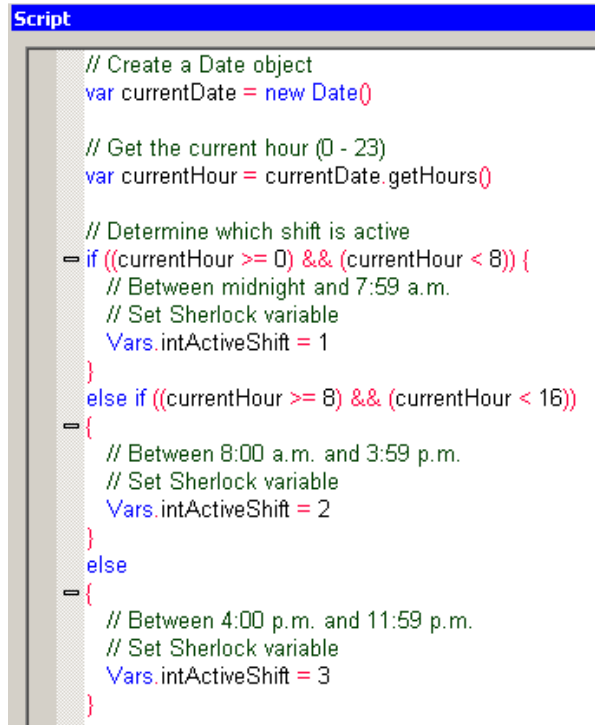
```
System.FolderDelete("D:\\Folder1\\Folder2")
```

deletes Folder2 and Folder3, but not Folder1.

The Date Object

The **Date** object returns the current date and time. The month, day, year (since 1900), hours (in a 24-hour clock), minutes, and seconds can be extracted with the methods **getMonth**, **getDay**, **getYear**, **getHours**, **getMinutes**, and **getSeconds**.

This code gets the current hour, determines which 8-hour shift is active, and sets the Sherlock variable *intActiveShift* to 1, 2 or 3, accordingly.



```
Script
// Create a Date object
var currentDate = new Date()

// Get the current hour (0 - 23)
var currentHour = currentDate.getHours()

// Determine which shift is active
= if ((currentHour >= 0) && (currentHour < 8)) {
    // Between midnight and 7:59 a.m.
    // Set Sherlock variable
    Vars.intActiveShift = 1
}
= else if ((currentHour >= 8) && (currentHour < 16)) {
    // Between 8:00 a.m. and 3:59 p.m.
    // Set Sherlock variable
    Vars.intActiveShift = 2
}
= else {
    // Between 4:00 p.m. and 11:59 p.m.
    // Set Sherlock variable
    Vars.intActiveShift = 3
}
```



- The array property **.length** returns the number of elements of both Sherlock and JavaScript arrays. This line of code

```
ArrayLength = Vars.varAreas.length
```

sets the JavaScript variable *ArrayLength* to the number of elements in the Sherlock array *varAreas*.
- If you need to read more than a few elements from a Sherlock array variable, it is more efficient to copy the entire variable to a JavaScript variable than it is to read the elements of the Sherlock variable one at a time. (In these examples, *Vars.varAreas* is a Sherlock number array variable, and *Vars.varTotalArea* is a Sherlock number variable; *jsArrayAreas* and *jsTotalArea* are JavaScript variables.)

This...

```
jsArrayAreas = Vars.varAreas
jsTotalArea = 0
for (i=0; i<jsArrayAreas.length; i++) {
    jsTotalArea += jsArrayAreas[i]
}
Vars.varTotalArea = jsTotalArea
```

...not this

```
jsTotalArea = 0
for (i=0; i<Vars.varAreas.length; i++) {
    jsTotalArea += Vars.varAreas[i]
}
Vars.varTotalArea = jsTotalArea
```

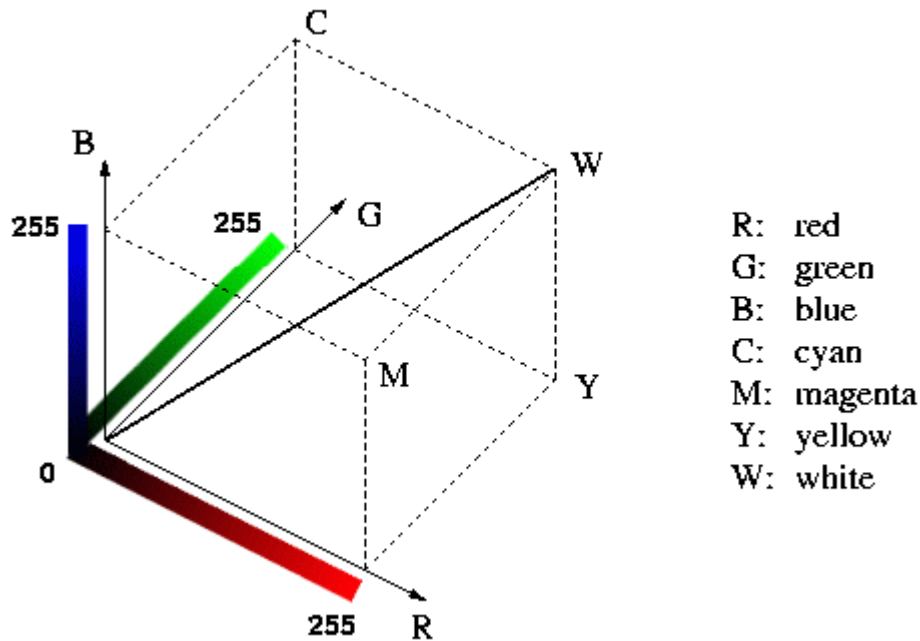


- **true** and **false** are Booleans; 0 and 1 are numbers.
- Variable names and predefined literal names are case sensitive. **True** is a variable; **true** is a predefined literal.
- Sherlock and JavaScript cannot access the same text file at the same time. A text file that has been opened with Sherlock's **IO:File Open** instruction cannot be accessed by the JavaScript System object file operations (**FileClose**, **FileAppend**, etc.) until Sherlock closes the file, and vice-versa.

For more information on JavaScript, click on **Start → Programs → Teledyne DALSA**

→Sherlock →JavaScript Help. There are also many online resources for learning more about JavaScript.

Color classifiers



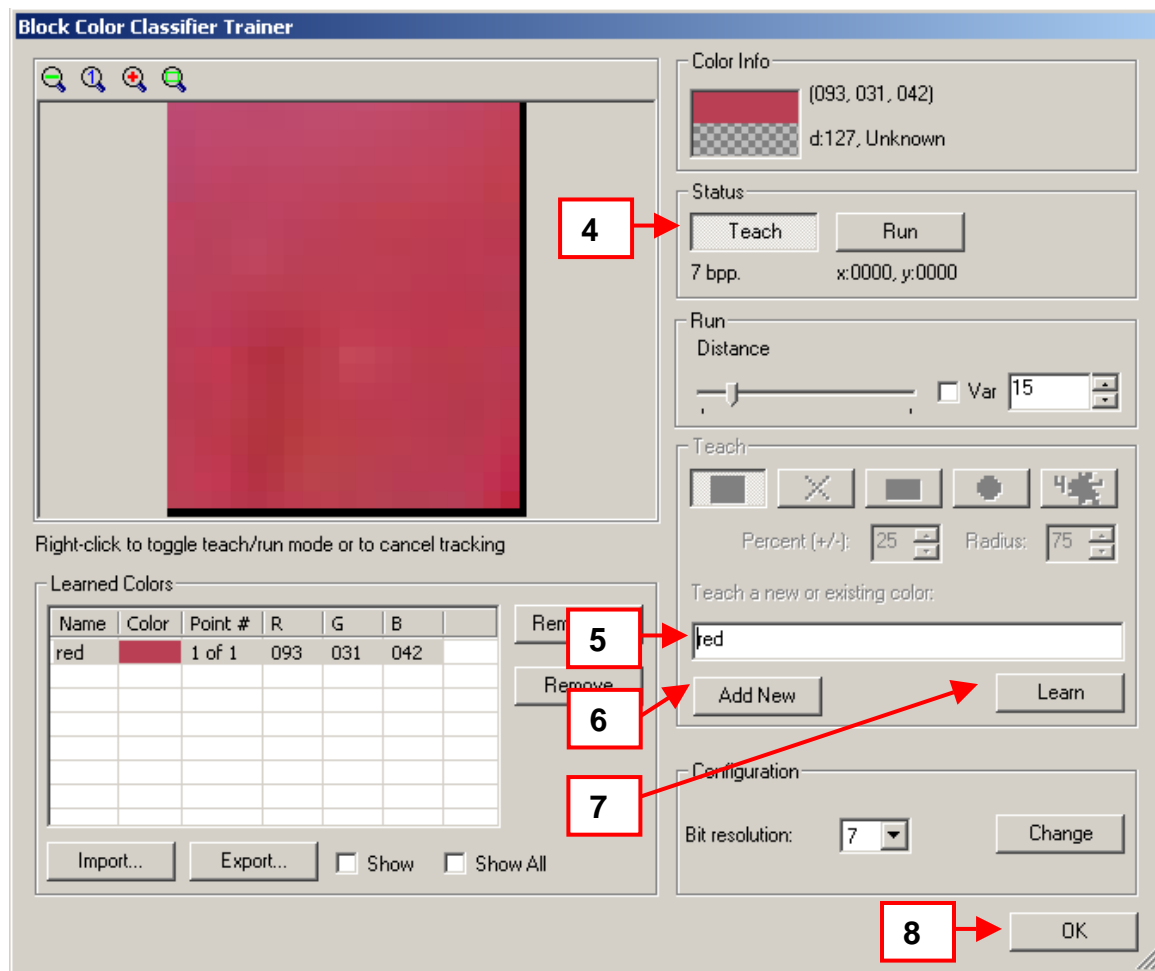
There are 16,777,216 (256^3) colors in the 24-bit RGB color space. In an image acquired by a color camera, it is unlikely that more than a few pixels will have exactly the same (r,g,b) values. What may appear to the eye as two identical dark blue pixels probably vary by at least a few values in each component – (10, 7, 223) and (8, 11, 220), for example.

For algorithms that compare pixels in an acquired image to a set of trained colors, using a single (r,g,b) value as the basis of comparison would not result in satisfactory results, since very few, if any, pixels being analyzed would match the single (r,g,b) value. For these algorithms, Sherlock uses color classifiers that define a trained color not as a single (r,g,b) value, but rather as one or more (r,g,b) “seed” values and their “volumes of influence”. A volume of influence is the three-dimensional space around the seed value or values that defines the extent of the color within the RGB color space.

Sherlock has two types of color classifiers, which are used in different algorithms. (See the sections on the individual algorithms for which classifier an algorithm uses.) The basic steps for training colors for both classifiers are the same.

1. Use a rectangle ROI to select a group of pixels. This can include pixels you do not want to include in the defined color.
2. Select an algorithm from the ROI's list. Click the algorithm's **Parameters** button.
3. On the parameters dialog, click the **Configure** button. Either the **Statistical Color Classifier Trainer** or the **Block Color Classifier Trainer** dialog will appear, depending on the algorithm you selected.
4. On the trainer dialog, click the **Teach** button.
5. Enter a name for the color.
6. Click the **Add New** button. The color name is added to the list of learned colors, but the color is not yet learned.

7. Click the **Learn** button. The average of the pixels within the ROI is calculated and displayed in the list of learned colors. The average and the color displayed next to it are for reference only. Depending on the algorithm, the classifier will use the average of the (r,g,b) values in the ROI as the seed, or it will use each (r,g,b) value in the ROI as an individual seed and assign each the same color name.
8. Click the **OK** button on the trainer dialog. Click the **OK** button on the algorithm's parameters dialog. Select a new group of pixels with the ROI, and repeat steps 3 thru 8.



The two classifiers – block and statistical – differ in how they classify colors, and in their options for determining whether an (r,g,b) value falls matches a learned color.

Block Color Classifier (shown above)

Classifier creation

The block color classifier uses the individual (r,g,b) values of the pixels in the ROI as seeds. When the first color is learned, a volume of influence expands around the seeds such that the color “owns” all of the (r,g,b) values in the color space. As more colors are learned, the volumes of influence of the learned colors are resized and reshaped so that they do not overlap. This prevents an (r,g,b) value from falling into more than one color’s volume of influence. To each (r,g,b) value in the RGB color space, the classifier assigns the identifier of the learned color whose volume of influence contains it, and its distance from the color’s closest seed.

The following diagram gives an extremely simplified idea of what a color space classified by the block classifier might look like. For the sake of readability, this color space has only two axes, red and blue. Both axes extend for 256 units (0 thru 255), but only a small portion of the space is shown. Every entry in the color space has a red and a blue component.

In this diagram, two colors have been learned, “red” and “blue”; their single seed values are in **bold**. (In reality, block classifier learned colors usually have more than one seed.) Their volumes of influence are demarcated by a bold line. The distances of the (r,b) values from their colors’ seeds are shown as “d = n”. **These distances do not reflect how distances are really assigned in the block classifier.**

blue d = 8	blue d = 7	blue d = 6	blue d = 5	blue d = 4	blue d = 3	blue d = 2	blue d = 1	blue d = 1	blue d = 1
blue d = 8	blue d = 7	blue d = 6	blue d = 5	blue d = 4	blue d = 3	blue d = 2	blue d = 1	blue seed	blue d = 1
red d = 3	red d = 3	blue d = 6	blue d = 5	blue d = 4	blue d = 3	blue d = 2	blue d = 1	blue d = 1	blue d = 1
red d = 2	red d = 2	red d = 2	red d = 2	red d = 2	red d = 3	blue d = 2	blue d = 2	blue d = 2	blue d = 2
red d = 2	red d = 1	red d = 1	red d = 1	red d = 2	red d = 3	blue d = 3	blue d = 3	blue d = 3	blue d = 3
red d = 2	red d = 1	red seed	red d = 1	red d = 2	red d = 3	red d = 4	red d = 5	red d = 6	blue d = 4
red d = 2	red d = 1	red d = 1	red d = 1	red d = 2	red d = 3	red d = 4	red d = 5	red d = 6	red d = 7

Classifier options

Bit resolution - The number of bits of each eight-bit color component of an (r,g,b) value that are used during training and analysis. Only the *n* highest bits of each color component are used. The lower the bit resolution, the more colors can differ and still be considered the same. You must set this before you start training colors, and click the **Change** button.

Distance – How much the (r,g,b) value of a pixel being analyzed at run time can differ from a one of the color’s seed values and still be recognized as the color. A pixel’s (r,g,b) value may fall within a color’s volume of influence, but if its distance from all of the color’s seed values is greater than *distance*, it will not be recognized as that color. The distance is applied equally to all colors in the classifier.

Point # – You can assign the same name to more than one learned color. Each new instance of a learned color is assigned a number (1 of 5, 2 of 5, 3 of 5, etc.).

Runtime

At run time, an algorithm checks the properties of the classifier’s (r,g,b) value to which a single pixel or the average of a group of pixels (depending the algorithm) maps, to determine its assigned color identifier and its distance from the seeds of the learned color. If the distance is less than the distance limit, the algorithm returns the name of the learned color. If the distance is

greater than the distance limit, the (r,g,b) value's color is "unknown". In the preceding simplified two-dimensional diagram, the highlighted (r,b) value falls within the volume of influence of the **red seed**, and its distance from the seed is 5. If the distance limit is 5 or more, the color of the highlighted (r,b) value is "red". If the distance limit is 4 or less, the color of the highlighted (r,b) value is "unknown".

Statistical Color Classifier

Classifier creation

The statistical color classifier calculates the average of all of the pixels in the ROI to create a single seed color, and calculates a volume of influence around the seed. As more colors are learned, their volumes of influence may overlap.

Classifier options

Used channels – Which of the (r,g,b) components are used to learn and compare colors. You must set this before you start training colors, and click the **Change** button.

Bit resolution – The number of bits of each eight-bit color component of an (r,g,b) value that are used during training and analysis. Only the n highest bits of each color component are used. The lower the bit resolution, the more colors can vary and still be considered the same. You must set this before you start training colors, and click the **Change** button..

Threshold – How much an (r,g,b) value being analyzed at run time can differ from a color's seed value and still be recognized as the color. The threshold can be set differently for different colors in the classifier.

Runtime

At run time, an algorithm using the statistical classifier calculates which seed the (r,g,b) value being analyzed is closest to. If the (r,g,b) value falls outside the closest learned color's threshold, the (r,g,b) value's color is "unknown".

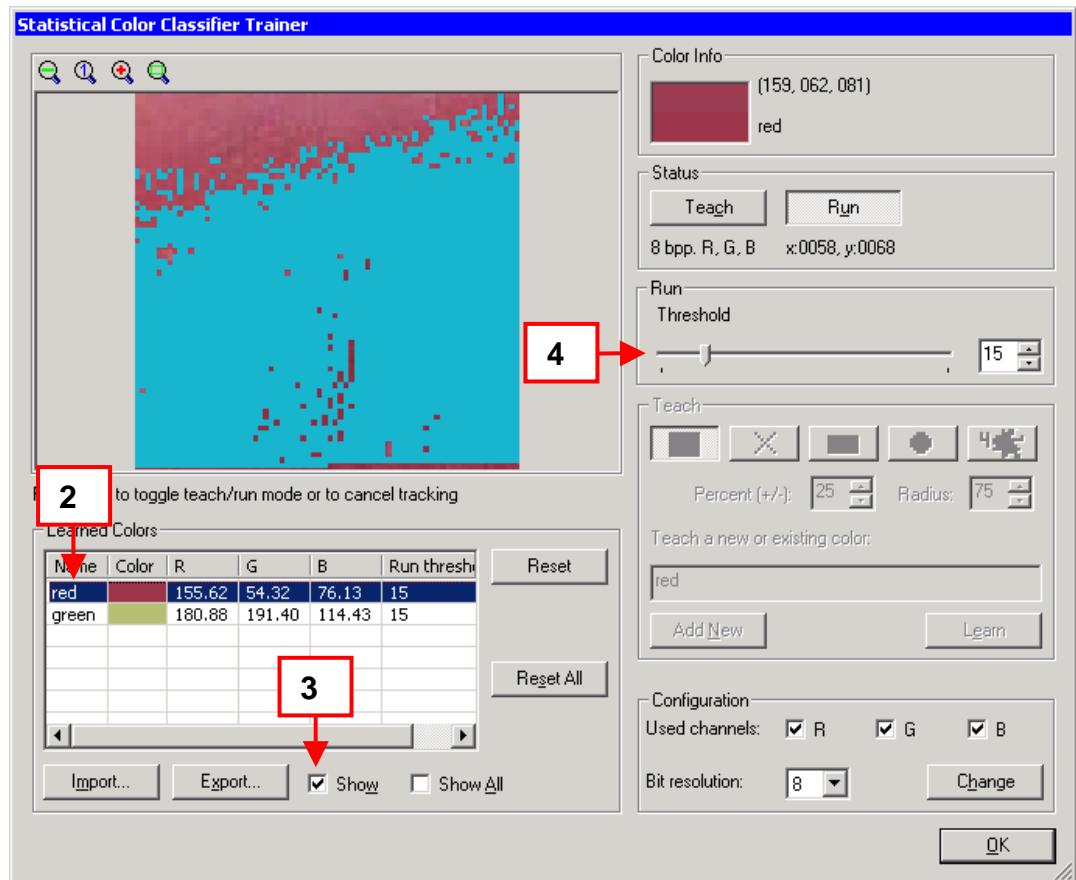
Fine-tuning a classifier

Once you have trained one or more colors, you can test the distance or threshold setting to determine whether it is too high or too low, and adjust it accordingly.

1. Select an area of pixels with the ROI.
2. Select a color from the list of learned colors.
3. Enable **Show**. All of the pixels in the ROI that are within the selected color's distance or threshold limit are highlighted with a uniform color.
4. Raise or lower the distance or threshold.

(Enable **Show All** to highlight all of the pixels in the ROI that fall within any learned color's volume of influence, and within the color's distance or threshold.)

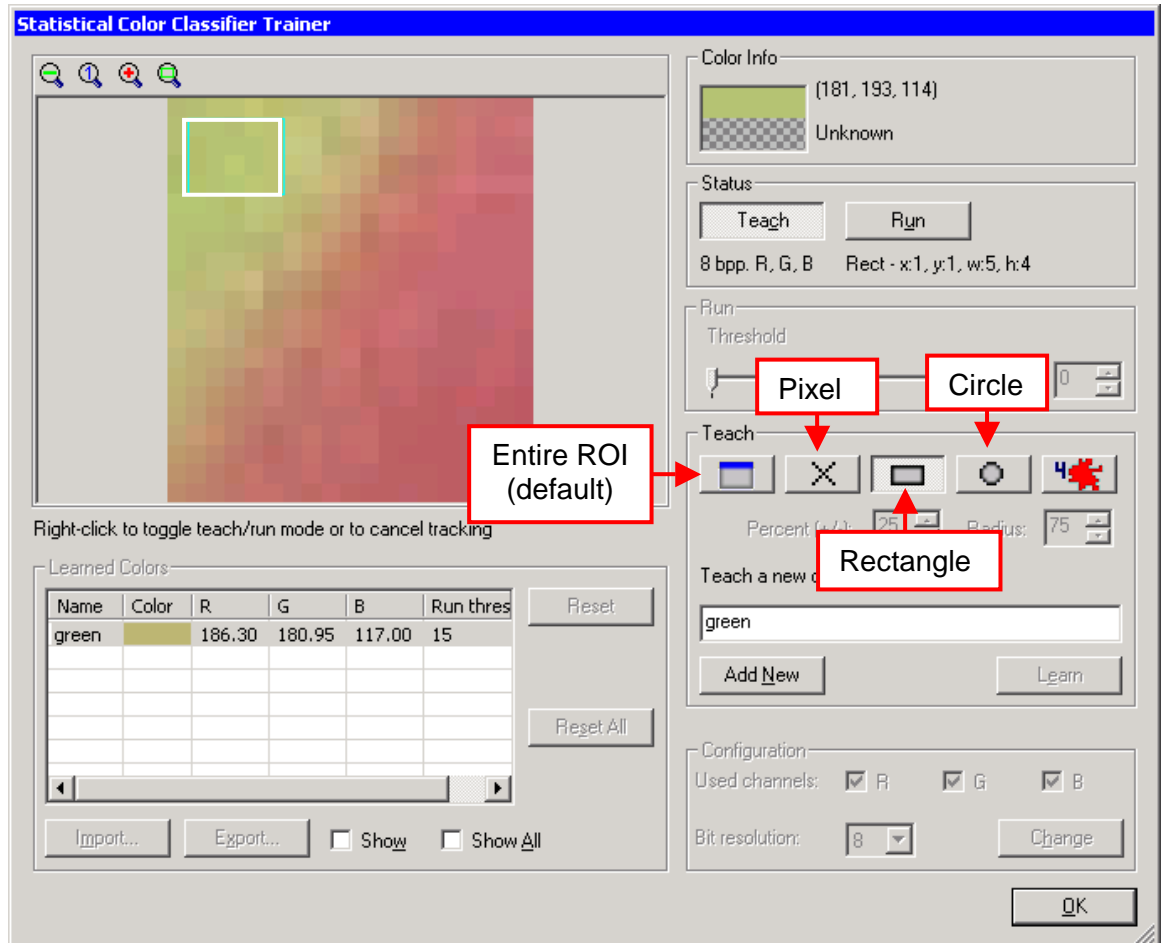
Here all the “red” pixels are highlighted.



The pixels that are selected for training a color should be about the same (r,g,b) value, or at least belong to the same color “family”. Training a color from an ROI that contains widely different (r,g,b) values – for example, (210, 80, 24) (dark orange) and (100, 200, 15) (medium green) – will have unexpected and probably undesired results. The same is true for assigning the same name to widely different (r,g,b) values with multiple **Add New/Learn** sequences in either classifier.

If you cannot easily isolate the few pixels you want to classify by drawing an ROI, you can select a portion of the ROI when training the color. You can select a single pixel, a rectangular portion, or a circular portion.

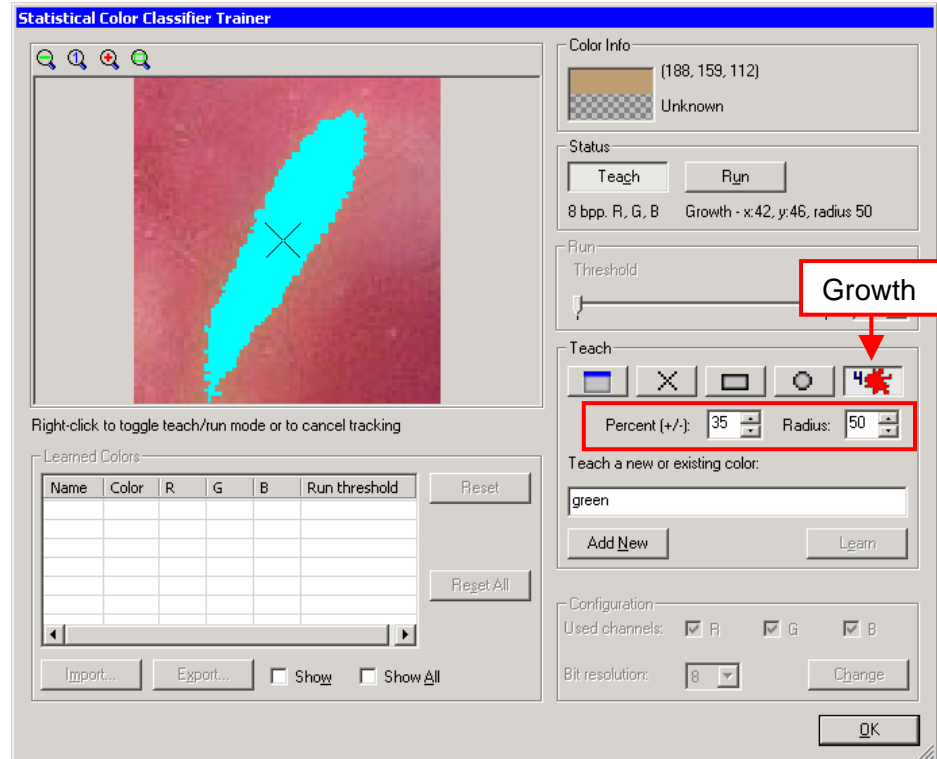
Here a rectangular portion of the ROI was selected to train “green”.



You can use the “growth” option to select pixels in the ROI based on their distance from a starting point, and their percent difference from the (r,g,b) value of the starting point.

1. Click the **Teach** button.
2. Enter a color name.
3. Click the Growth button.
4. Click a starting point in the ROI display. The pixels that are within the set distance (radius) and percent difference of the starting point are highlighted.
5. Change the radius and/or percent, click on the ROI display to clear the highlighted pixels, and click on the ROI display again to reselect the starting point. The changes to the radius and/or percent are reflected in the highlighted pixels.

Here the growth option was used to select a group of green pixels within a red area.

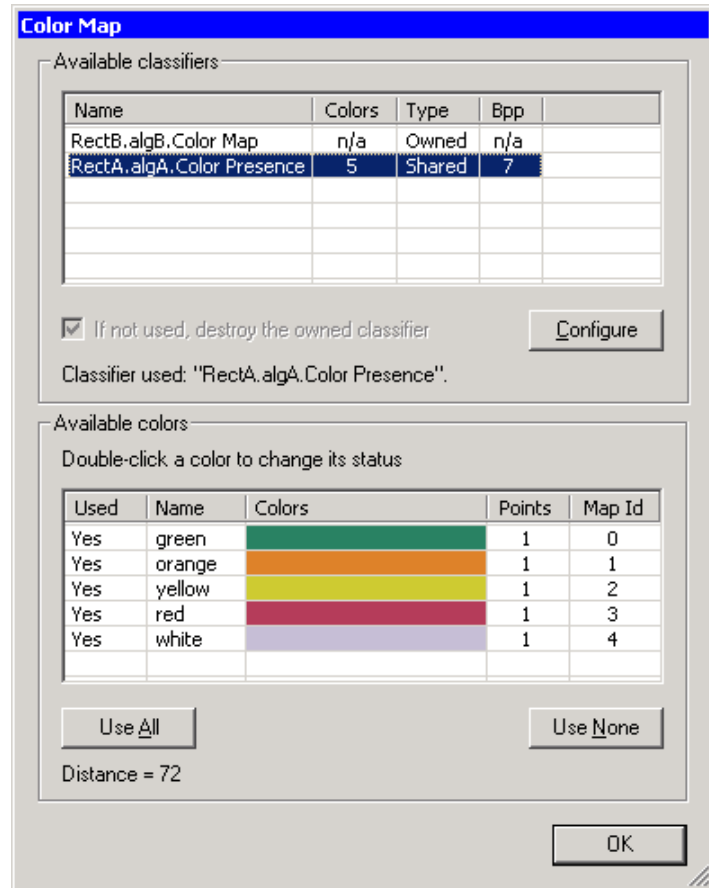


Sharing classifiers

Within an investigation, classifiers can be shared among algorithms that use the same type (block or statistical). The default behavior is to share a classifier upon algorithm instantiation.

A new ROI **RectB** was added to an investigation that already contained the ROI **RectA** executing the **Color Presence** algorithm. **Color Map** was selected as **RectB**'s algorithm. Because the two algorithms use the block classifier, by default **Color Map** shares the classifier configured for and owned by **Color Presence**.

To configure **Color Map**'s own classifier, click on **RectB.algB.Color Map** in the **Available classifiers** list, then click the **Configure** button.



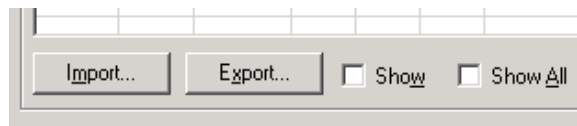
The **Color Map** dialog box is shown. It has a title bar with the text "Color Map". Inside, there are two main sections: "Available classifiers" and "Available colors".

Available classifiers: This section contains a table with columns: Name, Colors, Type, Bpp, and Name. The first row is "RectB.algB.Color Map" with values "n/a", "Owned", "n/a", and an empty Name column. The second row is "RectA.algA.Color Presence" with values "5", "Shared", "7", and an empty Name column. Below the table is a checkbox labeled "If not used, destroy the owned classifier" which is checked. To the right of the checkbox is a "Configure" button. Below the checkbox is the text "Classifier used: 'RectA.algA.Color Presence'".

Available colors: This section contains a table with columns: Used, Name, Colors, Points, and Map Id. The rows are: "Yes", "green", a green bar, "1", "0"; "Yes", "orange", an orange bar, "1", "1"; "Yes", "yellow", a yellow bar, "1", "2"; "Yes", "red", a red bar, "1", "3"; "Yes", "white", a white bar, "1", "4". Below the table are two buttons: "Use All" and "Use None". Below these buttons is the text "Distance = 72". At the bottom right of the dialog is an "OK" button.

Classifier files

A classifier can be exported to a file by clicking the **Export** button at the bottom left of the trainer dialog. A classifier file can be imported into algorithms in other investigations that use the same type of classifier (block or statistical) by clicking the **Import** button.



A small dialog box with two buttons: "Import..." and "Export...". To the right of the buttons are two checkboxes: "Show" and "Show All".



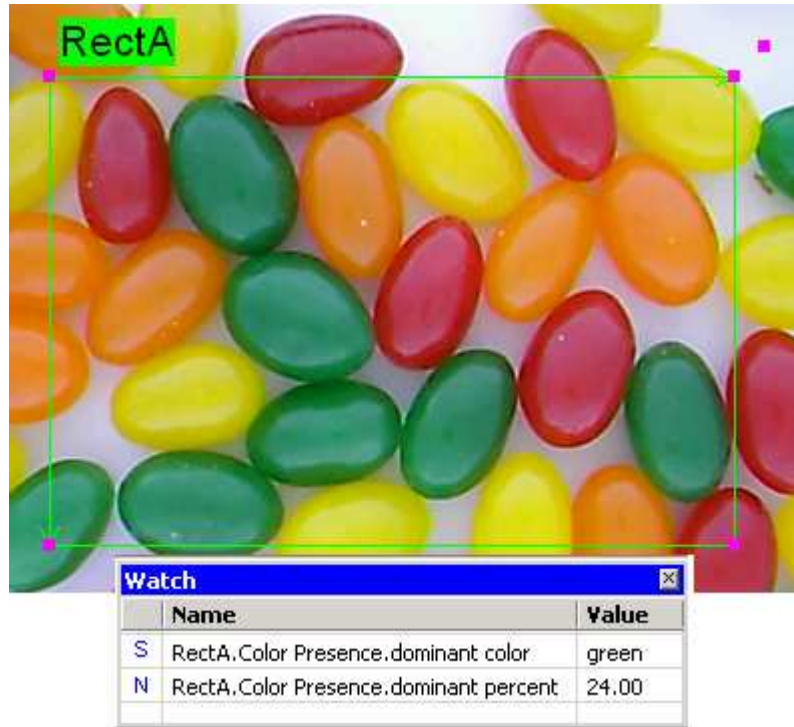
Block classifiers and statistical classifiers contain different information. You cannot import a block classifier file into an algorithm that uses the statistical classifier, and vice-versa.

Color presence

The **Color Presence Meter** counts the occurrence of trained colors within an ROI. It uses the block color classifier.

Pixels within an ROI are analyzed one-by-one by the classifier. If a pixel is within a trained color's distance limit, the count for the color is incremented. After all the pixels in the ROI have been analyzed, statistics on the colors are returned.

The classifier was trained with the colors green, red, orange, and yellow.






RectA.Color Presence.colors	RectA.Color Presence.percents
4 values:	4 values:
[000]green	[000]24.69
[001]orange	[001]20.71
[002]yellow	[002]17.80
[003]red	[003]15.90
Referenced objects:	Referenced objects:
None	None

About 20% of the pixels in the ROI were outside the distance limits of all of the trained colors.

Color spot meter

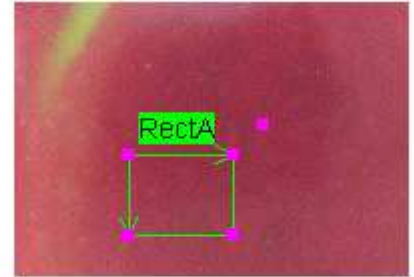
The **Spot Meter** compares the average (r,g,b) value of the pixels in an ROI to the colors in a classifier. If the average value falls within a color's threshold, the name of the color in the classifier is returned, as well as the average (r,g,b) values and standard deviation of the individual pixels' (r,g,b) values from the average. The Spot Meter uses the statistical classifier. If the average (r,g,b) value does not fall within any color's threshold, the returned color name is "Unknown".

The classifier was trained with the colors dark red, green yellow, and brown.

Name	Color	R	G	B	Run threshold
dark red		150.83	52.58	74.28	15
green yellow		180.30	188.70	113.97	15
brown		118.13	90.00	65.75	15

The average (r,g,b) value within RectA is within the threshold limit for the color "dark red".

RectA.Spot Meter.avg val->varAvgValue
3 values:
[000]152.62
[001]54.11
[002]75.92
Referenced objects:
varAvgValue



Watch		
	Name	Value
S	RectA.Spot Meter.color name	dark red



If the average (r,g,b) value of the pixels in an ROI does not fall within any color's threshold, the returned average (r,g,b) values are (0,0,0), not the average (r,g,b) values of the "Unknown" color.

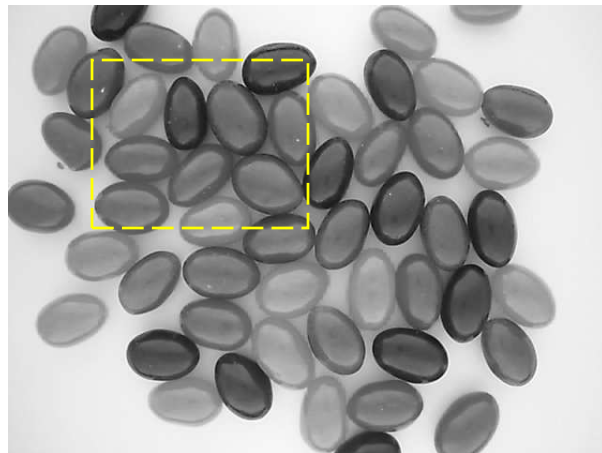
Color mapping

Most of Sherlock's algorithms work on monochrome images only. In order to apply these algorithms to a color image, the image must be converted to monochrome.

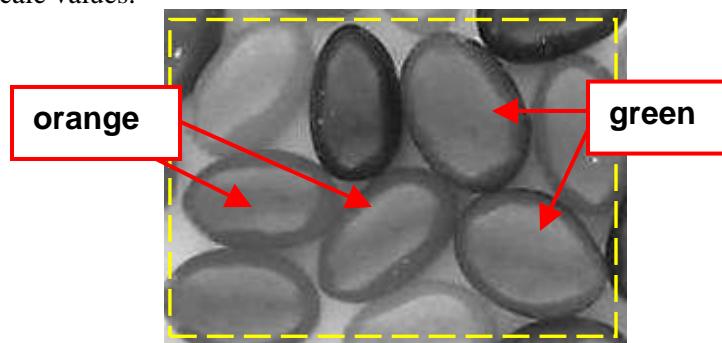
Consider the task of isolating the objects in this image based on their color so that they can be counted and measured.



A straightforward RGB-to-monochrome conversion yields what may seem to be a useable image.



But careful examination reveals that the green and orange objects share similar, overlapping ranges of grayscale values.



It would not be possible to set a threshold to separate the green objects from the orange, so that the **Connectivity – Binary** algorithm could be applied to them separately.

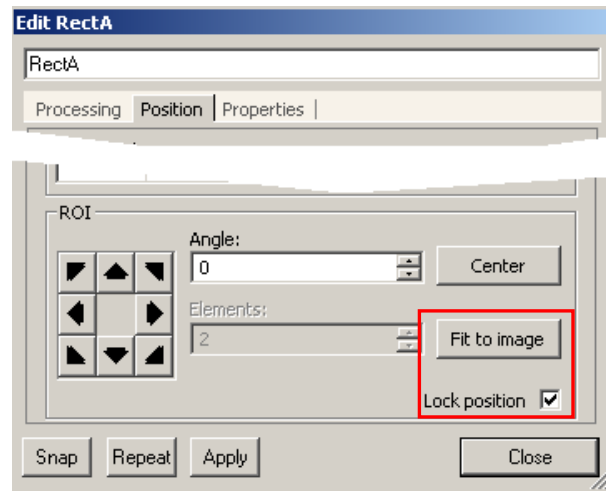
Color mapping assigns a grayscale value to the colors in a classifier. The **Color Map** algorithm uses the block classifier. As you train each color, it is assigned the next number in the sequence {0, 1, 2, 3,..., 255}.

In this example, the colors in the source image – green, red, yellow, and orange – were mapped to the values 0, 1, 2 and 3.

Used	Name	Colors	Points	Map Id
Yes	green		1	0
Yes	red		1	1
Yes	yellow		1	2
Yes	orange		1	3

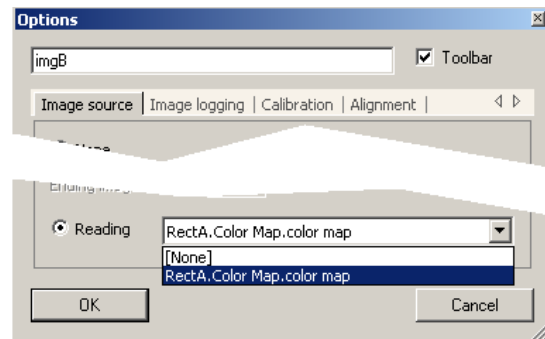
The next step is to create a grayscale image from the source color image based on the color map. You can map any rectangular section of the source image, or the entire image.

To map the entire source image, click the **Fit to image** button on the **Position** tab of the color source image window ROI's **Parameters** dialog. It's a good idea to also check the **Lock position** checkbox.

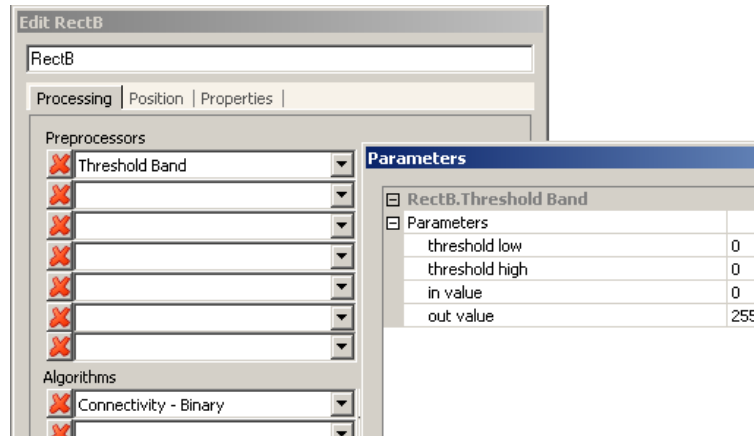


Create a destination image window, and add a rectangle ROI to it.

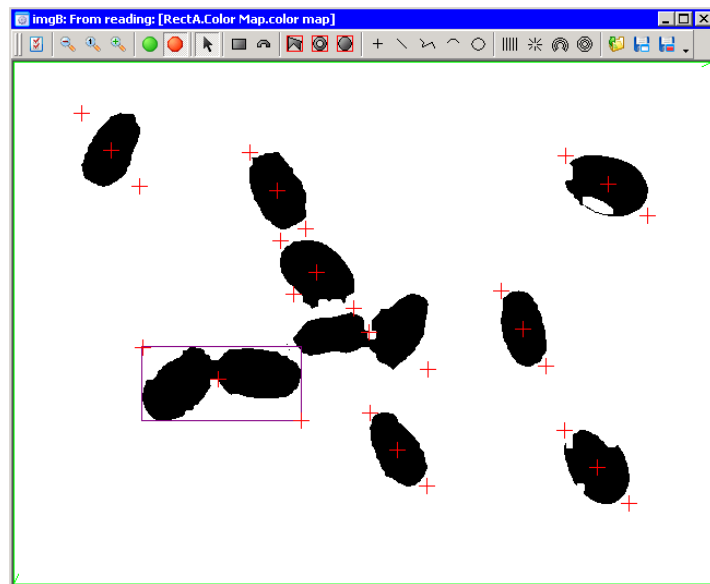
In the destination image window's **Options** dialog, select **Reading** as the image source, and the color map created by the **Color Map** algorithm as the reading.



To analyze only the green objects, select the **Threshold Band** preprocessor for the ROI, set its **threshold low** and **threshold high** parameters to 0, **in value** to 0, and **out value** to 255. This will set all pixels from **threshold low** to **threshold high** (0 to 0) to 0, and all other pixels to 255. For the algorithm, select **Connectivity – Binary**. (By default, the **Connectivity – Binary** algorithm processes black [0] pixels.)



When the investigation executes, the grayscale image is thresholded so that only the “green” pixels (value 0) are mapped to black. The **Connectivity – Binary** algorithm returns information about the green objects.



To analyze the red objects, set the **Threshold Band** parameters **threshold low** and **threshold high** to 1 and 1; for yellow, 2 and 2; etc.

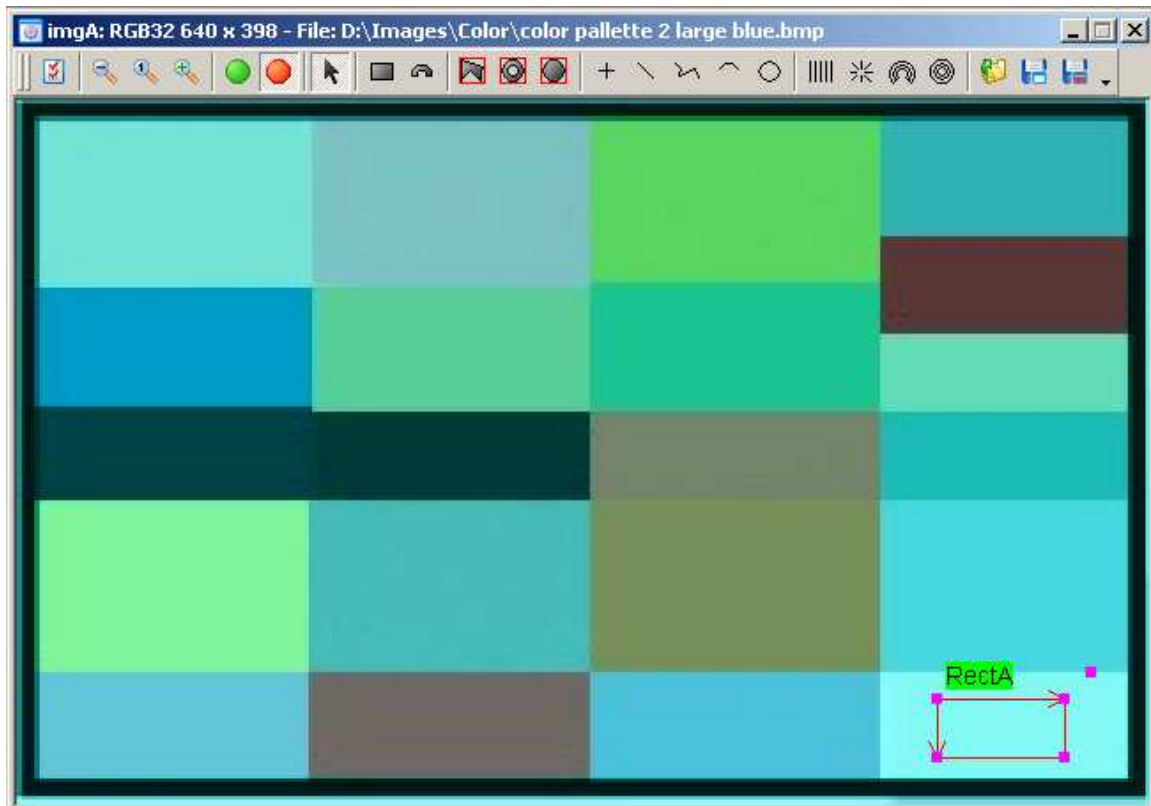
Color Correction

Images acquired by color cameras sometimes exhibit subtle color shifts that can make color analysis unreliable. Ideally, color problems should be corrected before the image is acquired, by adjusting the lighting and the camera's white balance. In cases where either or both of these is insufficient, difficult, or impossible, Sherlock can make color adjustments after the image is acquired.

Correction for color problems requires the use of two image windows executing a complementary algorithm and preprocessor pair.

- In the first image window, the algorithm calculates the coefficients that must be applied to the red, green and blue color planes to change the pixels in an ROI to white.
- In the second image window, the preprocessor applies the coefficients calculated by the algorithm.

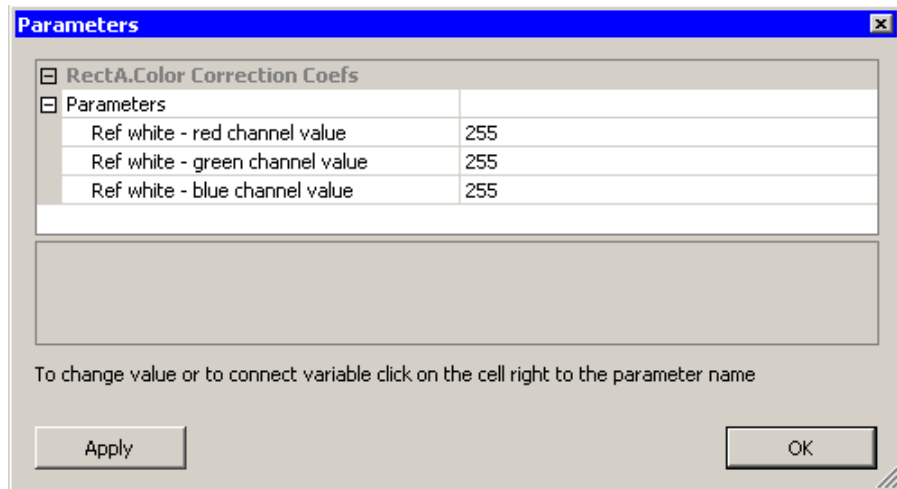
This image displays a noticeable shift in red; the lower-right rectangle is supposed to be white, but it is blue-green.



Correction calculation

RectA executes the **Color Correction Coefs** algorithm.

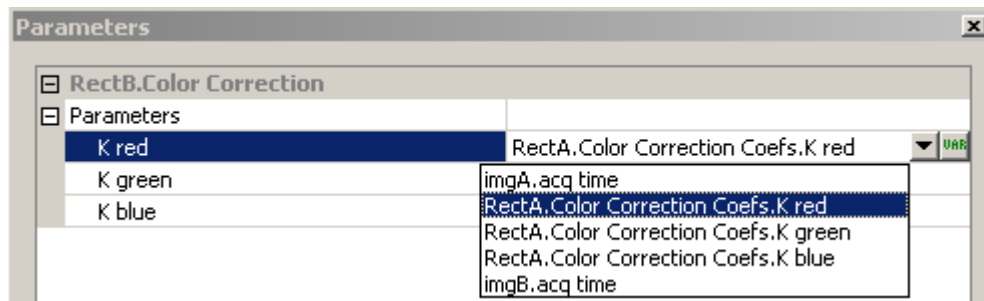
The default target values for the red, green, and blue components of a white pixel are 255, 255, and 255. You can change these values in the algorithm's **Parameters** dialog. The algorithm calculates the coefficients necessary to make the pixels in the ROI attain these target values.



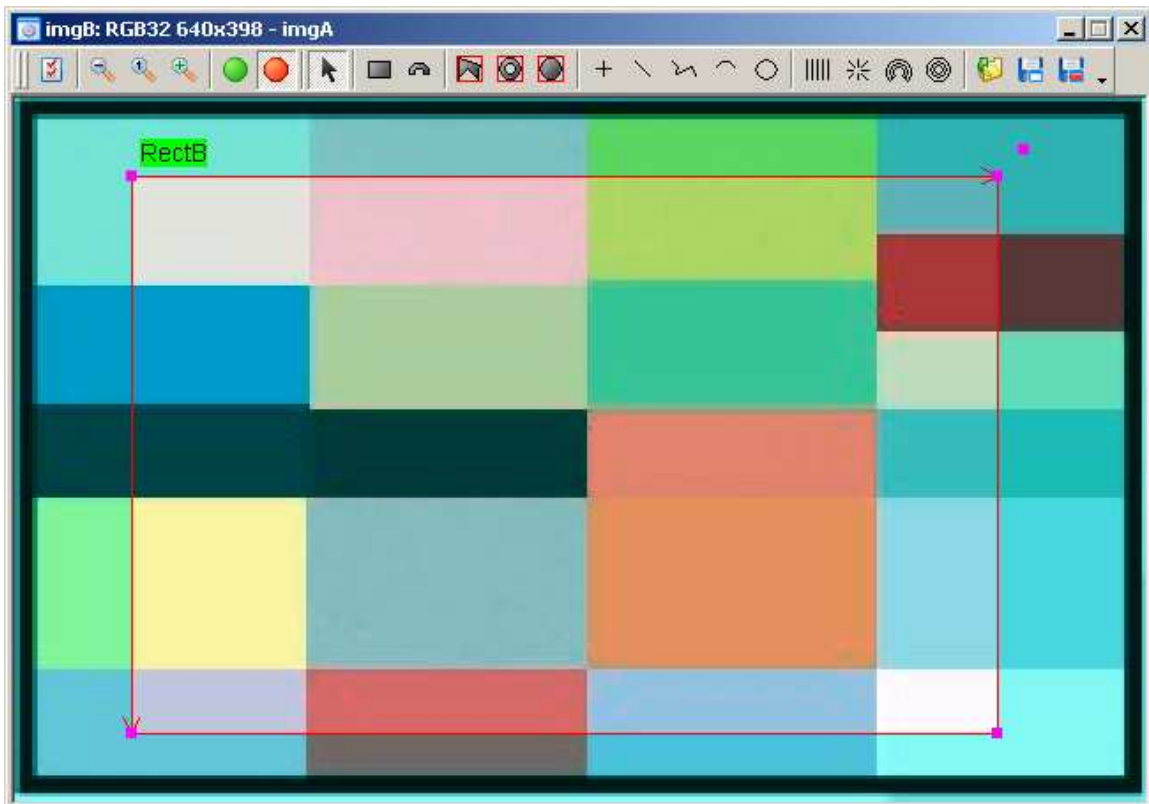
Correction application

An ROI executing the **Color Correction** preprocessor is created in the image to be corrected.

In the preprocessor's **Parameters** dialog, the coefficients calculated by the **Color Correction Coefs** algorithm are assigned to the proper parameters.



Here is the image with the color correction coefficients applied to the pixels in **RectB**. Note especially the change in the lower-right rectangle.

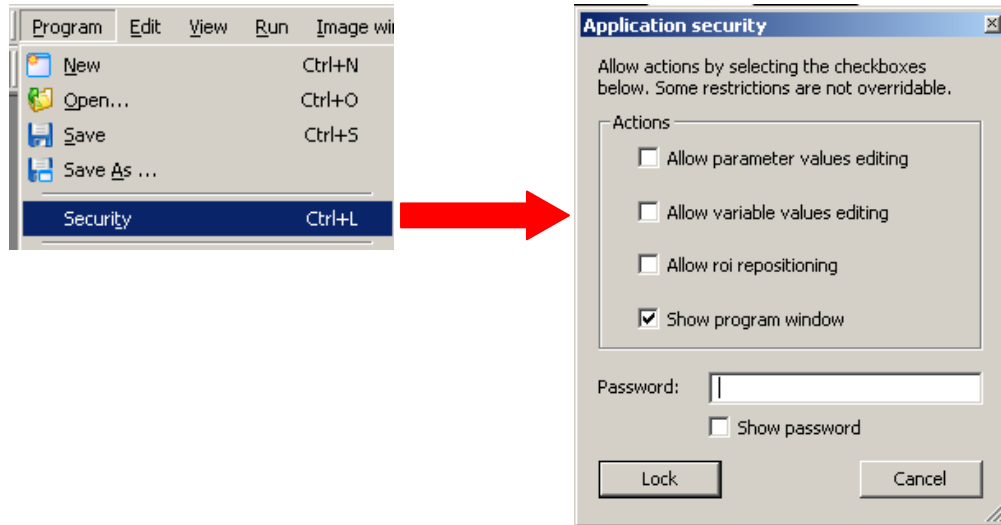


Security

Sherlock provides two levels of security, one to prevent anyone – including you – from accidentally modifying an investigation, and the second to prevent unauthorized people from loading and examining an investigation.

Preventing modifications

A security dialog available from Sherlock's **Program** menu lets you control which aspects of an investigation can be modified.



Set the level of protection by enabling (checking) or disabling (unchecking) the **Actions**, entering a password, and clicking the **Lock** button.

Actions

Allow parameter values editing – If disabled, preprocessor and algorithm parameters cannot be edited.

Allow variable values editing – If disabled, variable values cannot be edited, nor can variables be created, deleted, renamed, or modified in any way.

Allow roi repositioning – If disabled, ROIs cannot be moved or resized interactively, nor can ROIs be created, deleted, renamed, or modified in any way. (Except that their preprocessors' and algorithms' parameters can be edited if **Allow parameter values** is enabled.)

Show program window – If disabled, the Program window is not displayed.

If you do not enable any of the **Actions**, the investigation cannot be modified at all. No matter which **Actions** are enabled or disabled, a locked investigation can still be run.

When the investigation is locked, the **Lock** button is changed to **Unlock**. You must supply the correct password to unlock the investigation. In an unlocked investigation, the **Actions** settings are ignored.



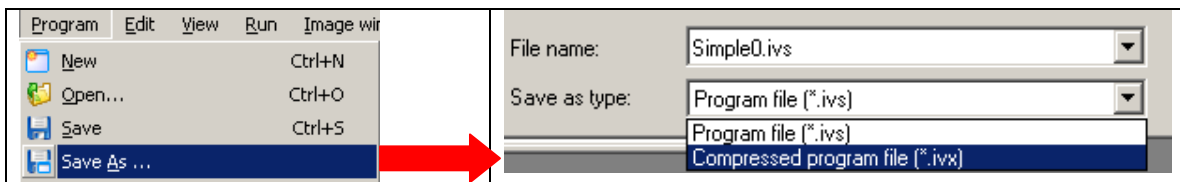
Even if the action **Allow roi repositioning** is disabled, ROIs can be moved programmatically, either through landmarking and alignment, or by calls to the **IO : Roi** instructions **Offset**, **Set Coordinate**, and **Set Rotation**.



The security settings have no effect on access from a front end application created with Visual C++, Visual C#, or Visual Basic. Even if an investigation is locked with none of the allowable actions enabled, a front end application can modify variables and reposition ROIs. It is up to the designer of the front end application to expose only those elements of an investigation that she wants the end user to be able to modify. (However, the protection selections remain active in the investigation file and are in effect when the investigation is accessed directly through the Sherlock GUI.)

Preventing loading

You can save an investigation in a compressed format by specifying file type “.ivx” in the **Program → Save As..** dialog and supplying a password.



An investigation saved as type .ivx cannot be loaded, either directly into the Sherlock GUI or by a Visual Basic, Visual C++ or Visual C# front end, without providing the correct password.

This password does not have to be the same password as for the security options described in Preventing modifications.

This option makes the most sense when you deploy an application with a compiled (.exe) front end created with Visual Basic, Visual C++ or Visual C#. In such a front end, you must call the Sherlock engine method **InvLoadComp(InvestigationName, Password)** to load the investigation. The password can be hard-coded, or the user can be prompted to enter the password.

Passwords

For the **Actions** options and compressed files:

- The maximum password length is 32 characters. The minimum length is one character.
- A password can contain any printable keyboard character – A thru Z, a thru z, 0 thru 9, !, @, #, etc.
- Passwords are **not** case-sensitive: “sahb74” is the same as “SaHb74”.

For the **Actions** options only:

- The password is not maintained when an investigation is unlocked; you must enter a password every time you lock the investigation.
- A history of used passwords is not kept; you can use the same password over and over again, or change it whenever you want to.

- There is no facility for retrieving a forgotten password. If you forget the password for a locked investigation, open the investigation .ivs file with a text editor (e.g., Notepad), look for lines like these at the beginning (the token values will differ)

```
<Investigation
  Version="7010009"
  Lock="15"
  Pwd="0ZzkJPnFDnuCU9dOENFFepa6D8tBOy58">
```

and carefully change the **Lock** and **Pwd** values.

```
<Investigation
  Version="7010009"
  Lock="0"
  Pwd=" ">
```

There is no space between the two quote marks following Pwd=.



Neither security method prevents an investigation from being deleted from the hard drive, nor from being corrupted by the operating system. It is a good idea to keep backup copies of your investigations, but “the best-laid schemes o’ mice an’ men...”

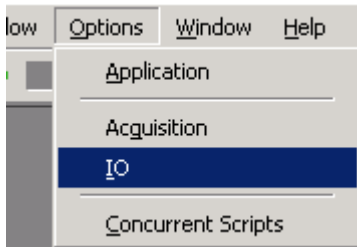
Program switching

Sherlock's program switching feature enables automatic switching among two or more investigations based on digital input signals.

In the **Program Switch** dialog (see **Configuring program switching**) you assign program numbers to the investigations you want to switch among, and select the digital inputs that will cause the switch. Program switching requires that enough digital inputs be dedicated to represent in binary the program numbers of the investigations, plus one more for a strobe to actuate the switch. For example, if you want to switch among four investigations, three digital inputs will have to be dedicated – two for the program number (00, 01, 10, 11), and one for the strobe.

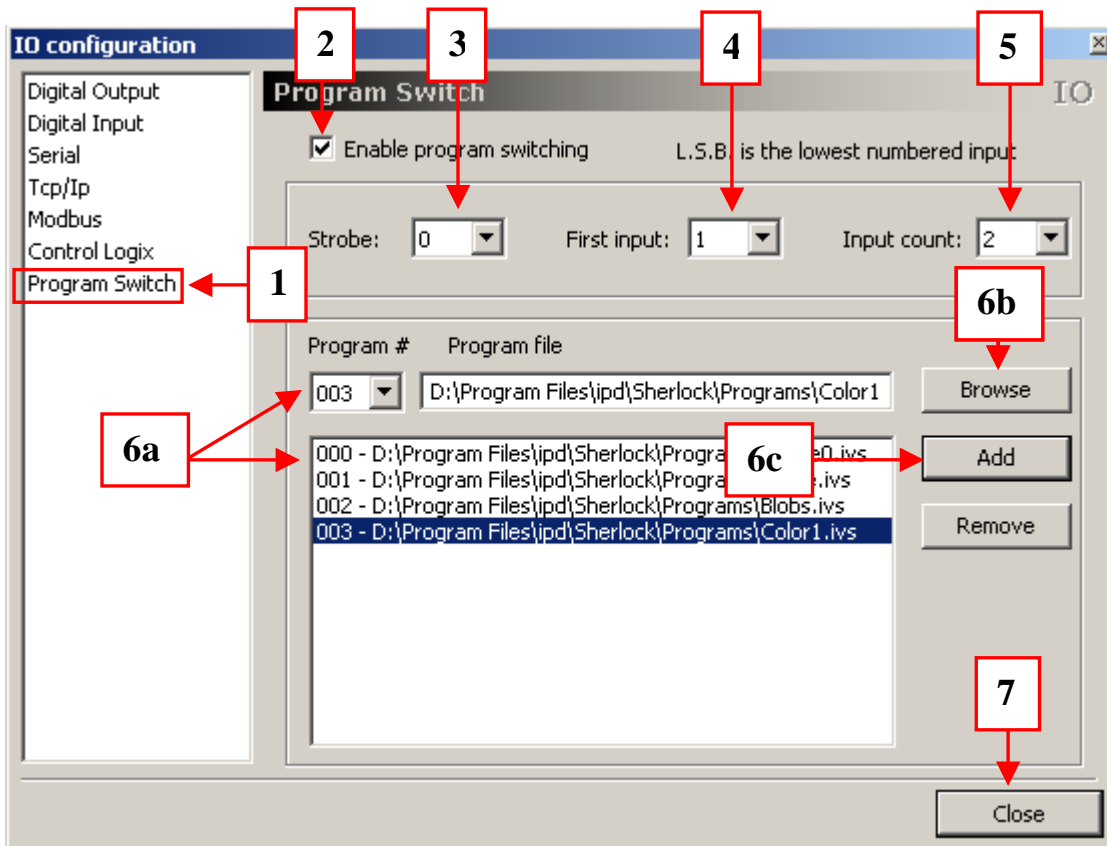
Configuring program switching

From Sherlock's main menu select **Options → IO**



In the **IO Configuration** dialog

1. Select **Program Switch** from the list of options.
2. Enable program switching.
3. Select a digital input to be the strobe to actuate program switching.
4. Select a digital input to be the least-significant digit of the binary representation of the program number.
5. Set the number of digital inputs that are needed to define the program number. The number of investigations that you can switch among will be $2^{\text{Input count}}$.
6. For every investigation that you want to switch to
 - a. Select a program number, either by clicking on an entry in the program list or from the **Program #** drop-down list
 - b. **Browse** to the location of the investigation and select it
 - c. Click the **Add** button
7. Click the **Close** button.



The digital inputs selected for program switching (program numbers and strobe) should not be used for other purposes, **but there is nothing to prevent you from doing this**. Careless selection of the program switching inputs could result in unpredictable behavior – for example, if the strobe is assigned to the same input as the external trigger!

Program switch settings are saved in <Sherlock>\bin\IpePgmSw.dat. The settings are active only when the **Enable program switching** checkbox on the **IO Configuration → Program Switch** dialog is checked.

Program switching at runtime

When program switching is enabled and the strobe input is triggered

- the currently-loaded investigation is stopped after it completes its current iteration
- the digital inputs that define the program number are read, and the investigation assigned to the program number is loaded
- the investigation is automatically put into **Run continuous** mode



If the number of investigations you want to switch among is not a power of 2, the program numbers assigned to the investigations do not have to be contiguous, nor start at zero.

For example, if you want to switch among three investigations, they could be assigned the program numbers 00, 10 and 11. **Do not leave ***Empty*** assignments in the program list.** Because Sherlock cannot switch to an ***Empty*** investigation, the currently-loaded investigation will continue to run, and you may never know that an attempt was made to switch to an unassigned program number. Instead, create and assign an investigation that does something, such as inform the user that an invalid combination of signals (and therefore an invalid program number) was sent to the digital inputs.



If a switch fails for a reason other than the ***Empty*** assignment case – for example, if a program number is associated with an investigation that has been deleted, moved or renamed

- the currently-loaded investigation is stopped after it completes its current iteration
- a new (empty) investigation is created
- Sherlock is halted

Severe program switch errors may cause an immediate halt of Sherlock.



The program switching software cannot detect and adapt to changes in a system's hardware (digital input) configuration. You must be careful to reconfigure the program switching setup if the hardware changes.

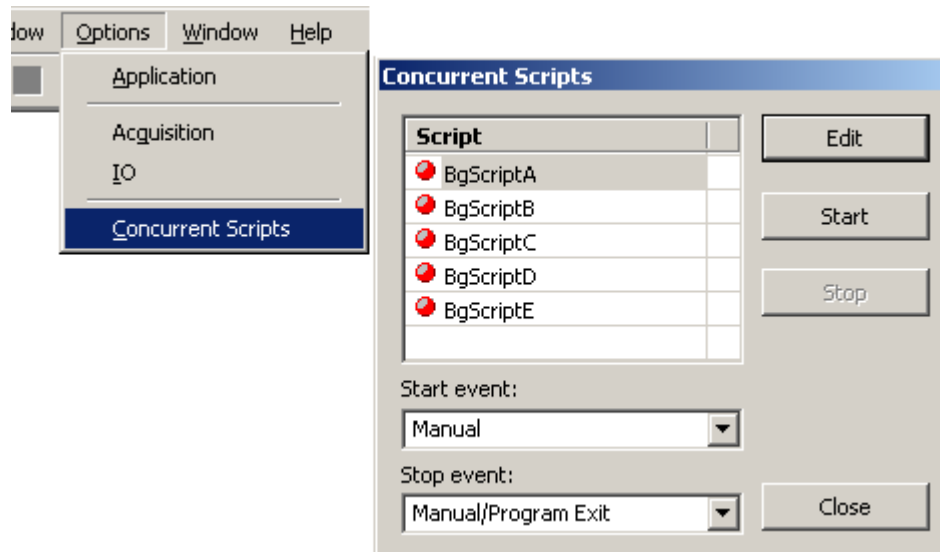
Concurrent scripts

A concurrent JavaScript module runs in the background, asynchronously from the main program.

A concurrent script is useful for

- performing computations independent from those in the main program
- calculating statistical data over time
- custom logging – normal or exceptional conditions
- monitoring and changing hardware status

The **Concurrent Scripts** dialog is available from the main menu **Options** entry.



To rename a script, left-click once on the script name, wait briefly, then left-click on the name again.

Select a script and click the **Edit** button to open the script editor. The editor is exactly the same as that for creating in-line JavaScript modules.

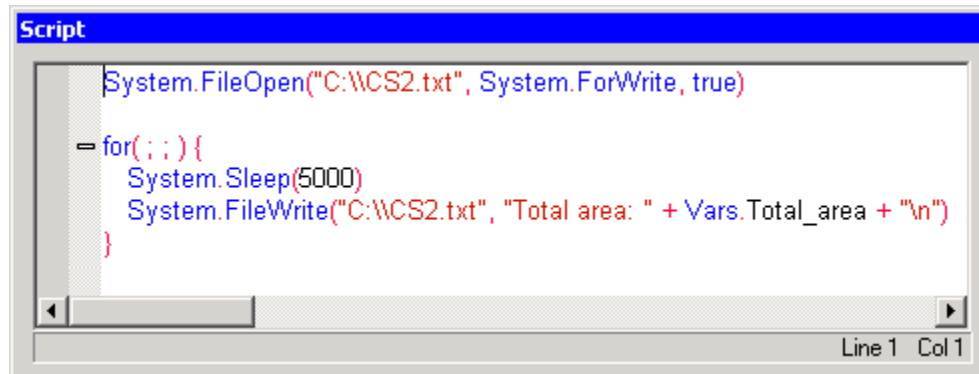
From the events listed in the **Start Event** and **Stop Event** drop-down lists on the **Concurrent Scripts** dialog, select the events that will launch and terminate the script. A script is usually launched by the **Investigation Load** or **Continuous Investigation Start** event.

A concurrent script is executed only once. To make any or all of the code in a script execute repeatedly, the code must be placed within a **for**, **while**, or **do...while** statement that will loop “forever” or until a condition is met.

In order to have the background thread yield control (especially in the case of tight loops) and improve application responsiveness, a **Sleep()** call should appear somewhere inside the loop.

The scripts are synchronized on Sherlock resources. If the main program and a script need to access the same resource -- variables, digital IO lines etc. -- they will **block/wait** until access is yielded by the thread/script currently using that resource. This behavior is **not** user configurable. No explicit synchronization is available to the user.

This script opens the text file **CS2.txt**. Within the **for** loop, the script goes to sleep for 5000 milliseconds, wakes up, and writes a line of text, including the value of the Sherlock variable **Total_area**, to the file. If the **Sleep** and **FileWrite** statements were not placed within the **for** loop, only one line would be written to the file, and the script would terminate. Because there are no condition arguments in the **for** statement, it will loop “forever” (until the script is terminated).



```
Script
System.FileOpen("C:\CS2.txt", System.ForWrite, true)

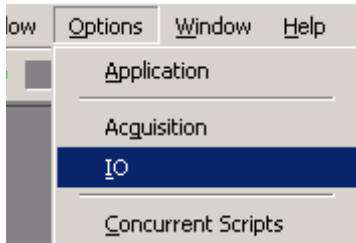
for(;;) {
    System.Sleep(5000)
    System.FileWrite("C:\CS2.txt", "Total area: " + Vars.Total_area + "\n")
}
```

Line 1 Col 1

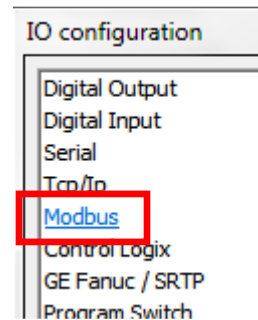
Modbus

Creating a Modbus connection

You define and activate a Modbus connection from the **IO Configuration** dialog, which is available from the main menu **Options** entry.



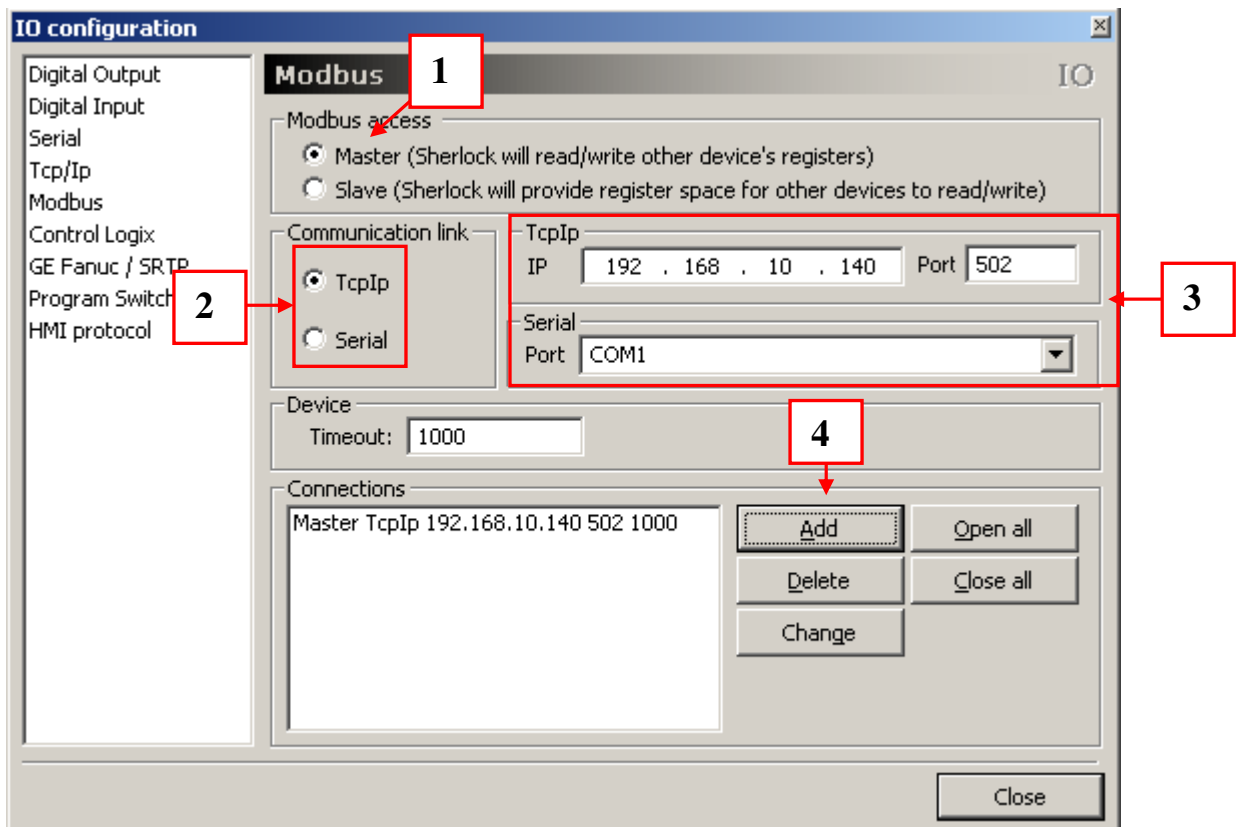
In the **IO configuration** dialog, select **Modbus** from the list of options.



Sherlock as Modbus Master

To create a Modbus connection in which Sherlock is the master

1. Select **Master** as the access mode
2. Select **TcpIp** or **Serial** as the communication link
3. a. If you selected **TcpIp** as the communication link, enter the IP address and port number of the remote computer which will be the slave to Sherlock
b. If you selected **Serial** as the communication link, select the serial port through which you want to communicate
4. Click the **Add** button to add the connection configuration to the list of connections



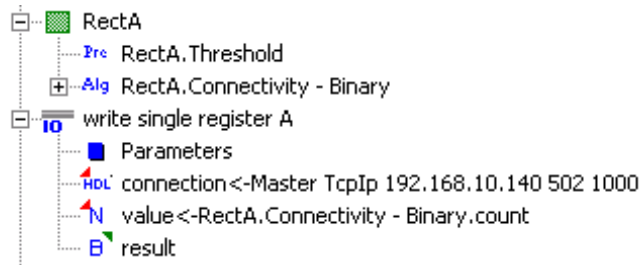
Using Sherlock as Modbus Master in a program

Sherlock provides eight Modbus Master instructions, available from the **IO: Modbus Master** folder:

1. Read multiple discrete
2. Read multiple registers
3. Read single discrete
4. Read a single register
5. Write multiple coils
6. Write multiple registers
7. Write single coil
8. Write a single register

The instructions are described in detail in Sherlock's online Help.

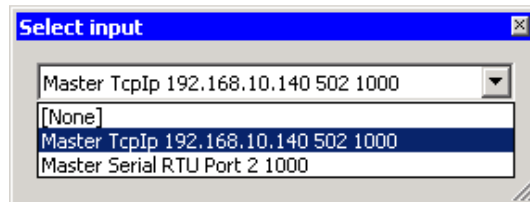
This example writes a value – the **count** reading from the **Connectivity – Binary** algorithm – to a register of the Modbus slave.



The **Parameters** settings depend on the operating characteristics of the slave device. Here, a single 16-bit value will be written to the register at address 40001 (the first register in the fourth register table).

Parameters	
write single register A	
Modbus	
device id	1
register	1
register table	4:0000
register type	short integer (16 bits)
register swap	False

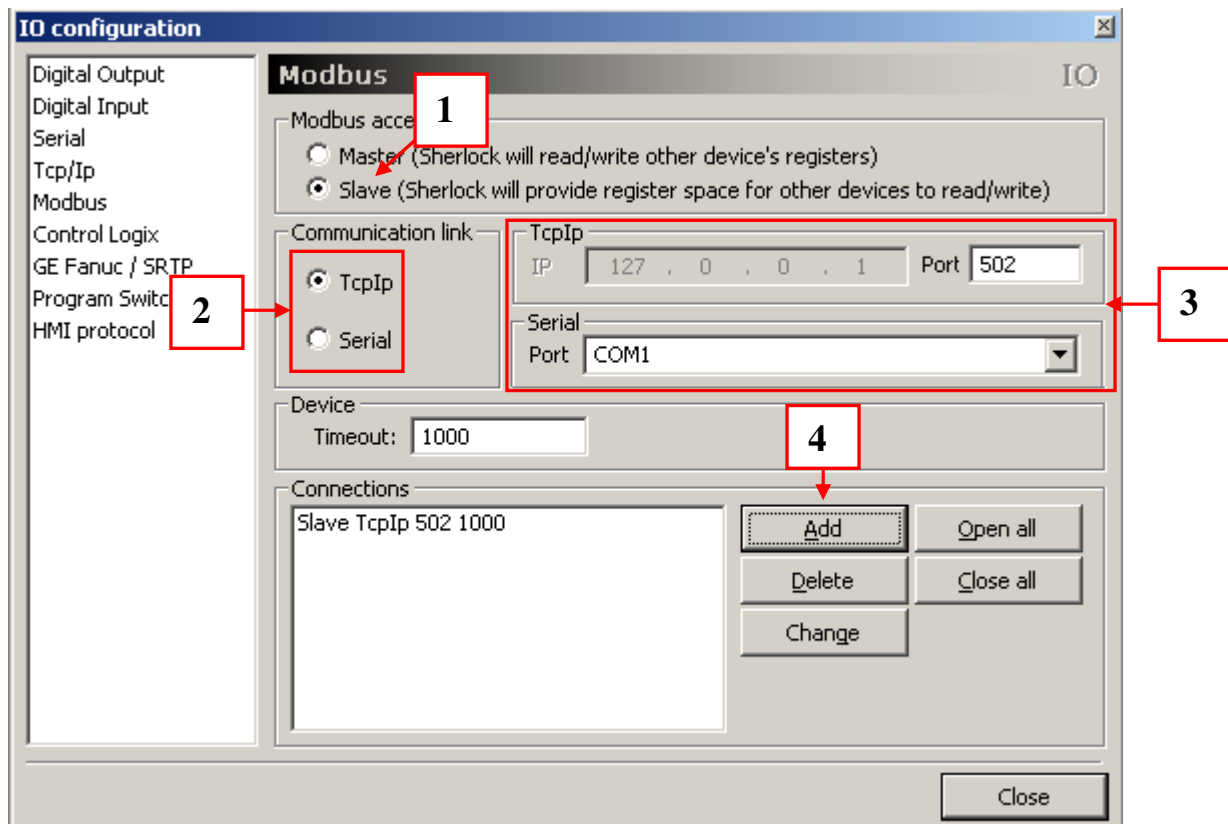
For the connection handle **HDL**, select the Modbus master connection definition.



Sherlock as Modbus Slave

To create a Modbus connection in which Sherlock is the slave

1. Select **Slave** as the access mode
2. Select **TcpIp** or **Serial** as the communication link
3. a. If you selected **TcpIp** as the communication link, enter the port number through which Sherlock will communicate with the Modbus Master
b. If you selected **Serial** as the communication link, select the serial port through which Sherlock will communicate with the Modbus Master
4. Click the **Add** button to add the connection configuration to the list of connections



Using Sherlock as Modbus Slave in a program

Sherlock provides four Modbus Slave instructions, available from the **IO: Modbus Slave** folder:

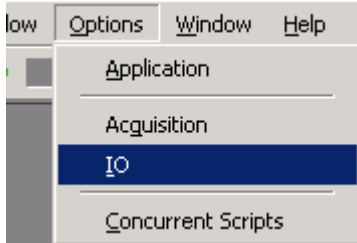
1. Read discrete
2. Read a single register
3. Write discrete
4. Write a single register

The instructions are described in detail in Sherlock's online Help.

Serial communication

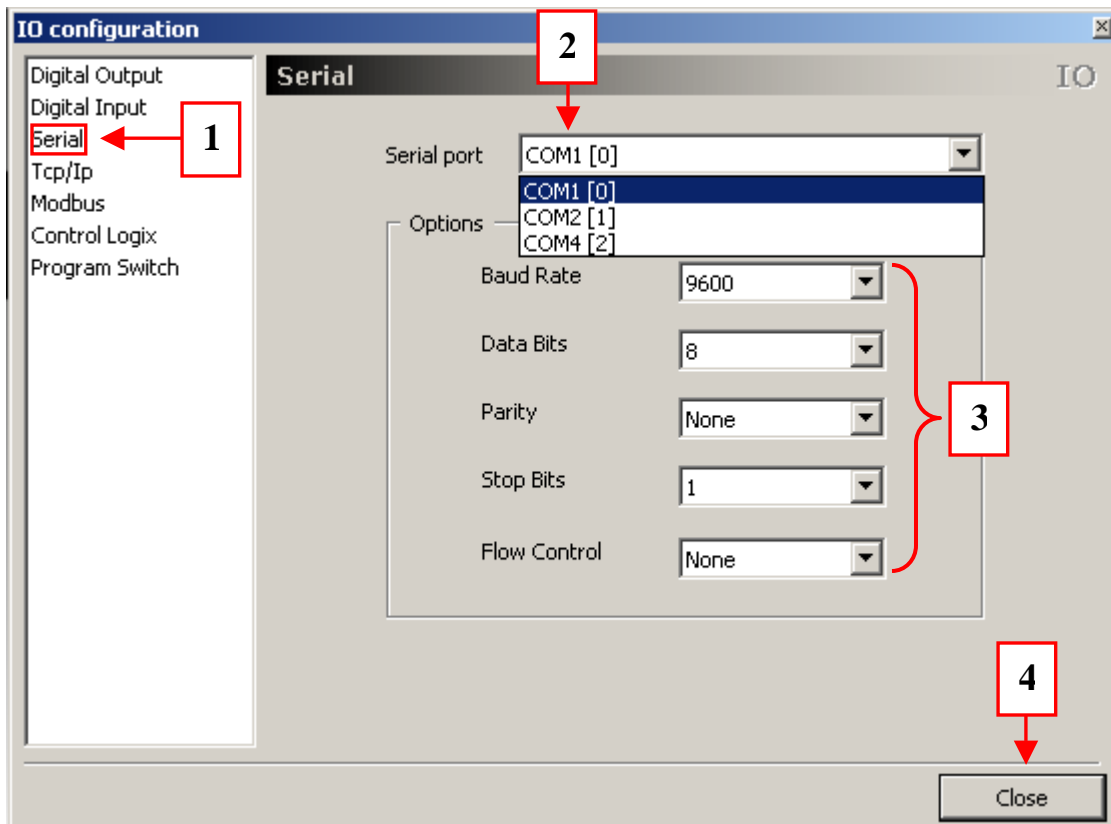
Defining a serial port's properties

From Sherlock's main menu select **Options** → **IO**



In the **IO Configuration** dialog

1. Select **Serial** from the list of communication protocols
2. Select the COM port from the list of available ports. The ports are listed by name, followed by [index number].
3. Set the port's properties
4. Click the **Close** button



Instructions

Sherlock provides four serial communications instructions, available in the **Instructions IO : Serial** folder:

- **Receive Character:** Read characters sent from another application; no termination character expected.
- **Receive Line:** Receive a string of characters sent from another application; a specific character must terminate the string.
- **Send String:** Send a string to the port for immediate transmission.
- **Purge Buffer:** Purge the specified buffer of the specified port.

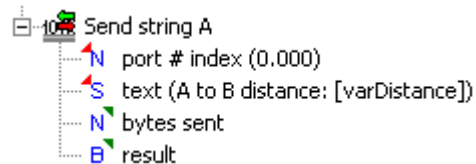
Instruction examples



Ports are referenced by index as shown in the **IO Configuration** dialog, not by port number. In these examples, **COM1** is index 0.

IO : Serial Send String

Send string A, an instance of **IO: Serial Send String**, sends the string "A to B distance: [varDistance]" to COM1. "[varDistance]" is replaced with the value of the variable *varDistance*.



Since the instruction cannot know whether any application read the sent string, the **result** reading will almost always return **True**.

Use the "C" language escape sequences to embed binary data into the string:

\odd where dd = decimal digit

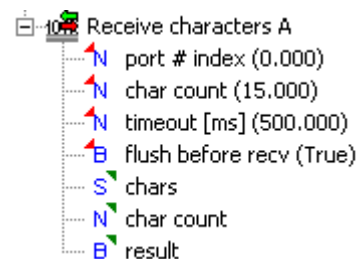
\xff where ff = hexadecimal digit

For example, the string "Send value 45: \045" will send the ASCII string "Send value 45: " followed by the decimal value 45 (not an ASCII character string "45").

IO : Serial Receive Character

Receive characters A, an instance of **IO: Serial Receive Character**, waits 500 milliseconds for 15 characters to arrive in COM1's input buffer.

If 15 characters do not arrive within 500 milliseconds, the **result** reading returns **False**.



flush before recv (True): COM1's input buffer is flushed of any unread characters before the instruction starts waiting for 15 characters.

Whether or not the instruction times out, the **char count** reading returns the number of characters received. The characters are removed from the port's input buffer and made available from the **chars** reading.

If more characters than were specified by the **char count** parameter are in the port's input buffer when the instruction is executed (and if **flush before recv** is set to False), only **char count** characters are removed from the buffer and made available from the **chars** reading; the remaining characters are left in the buffer.

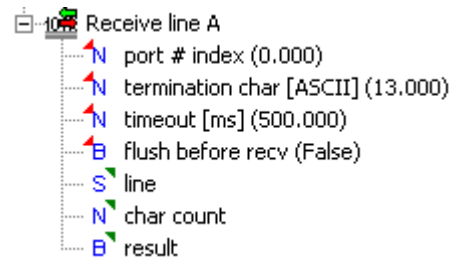
IO : Serial Receive Line

Receive line A, an instance of **IO: Serial Receive Line**, waits 500 milliseconds for a string of characters of any length terminated with a carriage return (ASCII 13) to arrive on COM1's input buffer.

If a carriage return does not arrive within 500 milliseconds, the **result** reading returns **False**.

flush before recv (False) : COM1's input buffer is **not** flushed of any characters before the instruction waits for a line.

Whether or not the instruction times out, **char count** returns the number of characters received, and any characters received are available from the **line** reading. The port's input buffer is cleared **up to and including the termination character**.



The termination character is not returned in the **line** reading, nor is it counted in **char count**. For this example, if the sending application sends the line “abcde<carriage return>” (<carriage return> = ASCII 13), **line** will contain “abcde” and **char count** will return 5. However, many applications automatically append a carriage return **and** a line feed to a sent string. In this case if the sending application sends the string “abcde<carriage return><line feed>” twice, two calls to this **Receive line** instruction will result in:

First call to **Receive Line**:

char count = 5

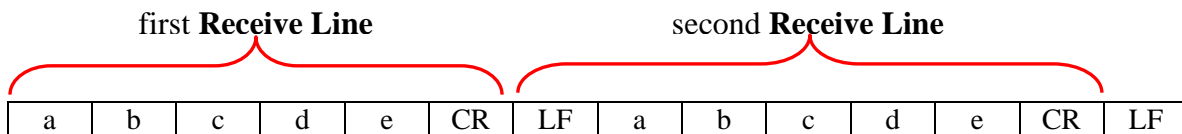
line = abcde

Second call to **Receive Line**:

char count = 6

line = {0xa}abcde

The line feed (0xa = ASCII 10 = <line feed>) is left over from the first call to **Receive Line**.



This is one reason – a very good one – to set **flush before recv** to True.

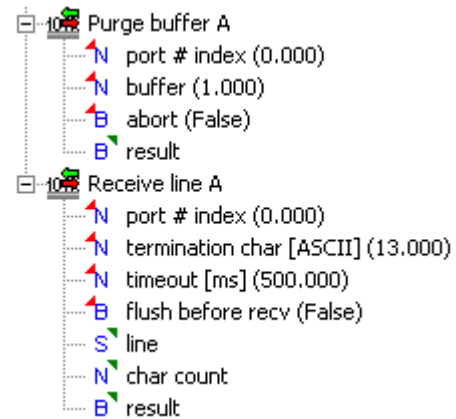
IO : Serial Purge Buffer

For the **buffer** parameter:

- 0 = TX / output / transmit buffer
- 1 = RX / receive / input buffer
- 2 = both TX and RX buffers

COM1's input buffer is purged of any characters before the call to **Receive line A**.

The same could be achieved by setting **Receive line A**'s **flush before recv** parameter to True. **Purge Buffer** is usually used to flush a port's input and/or output buffers from any point in an investigation, not just before a call to send or receive.



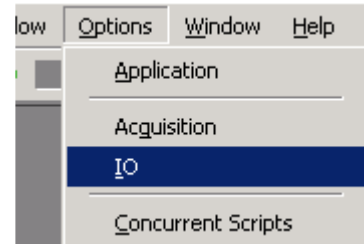
TCP/IP communication

TCP/IP (Transmission Control Protocol/Internet Protocol) is a suite of communications protocols used to connect computers on a network.

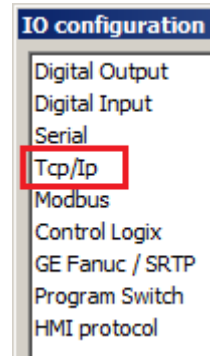
Sherlock implements TCP/IP with a server/client model. In this model, a server makes one of its ports available to send and receive data, and a client communicates to the server through this port. The client must know the IP Address or name of the server, and the port number that the server has made available for communication. Sherlock can act as a TCP/IP server or client, or both within the same investigation.

Selecting TCP/IP

From Sherlock's main menu, select **Options → IO**.



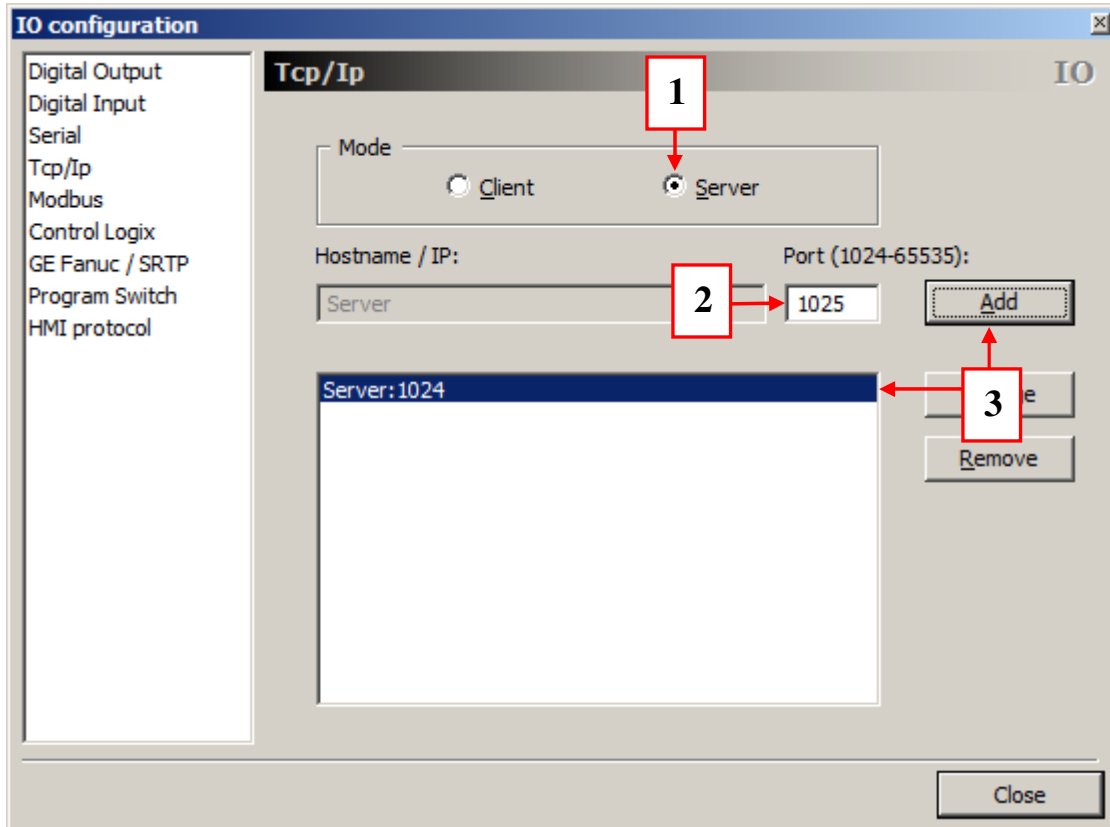
In the **IO configuration** dialog, select **Tcp/Ip** from the list of options



Sherlock as a server

As a server, Sherlock makes one of the ports on its host computer available for communications. Sherlock sends data to this port to be read by a client, and reads data sent to this port by a client.

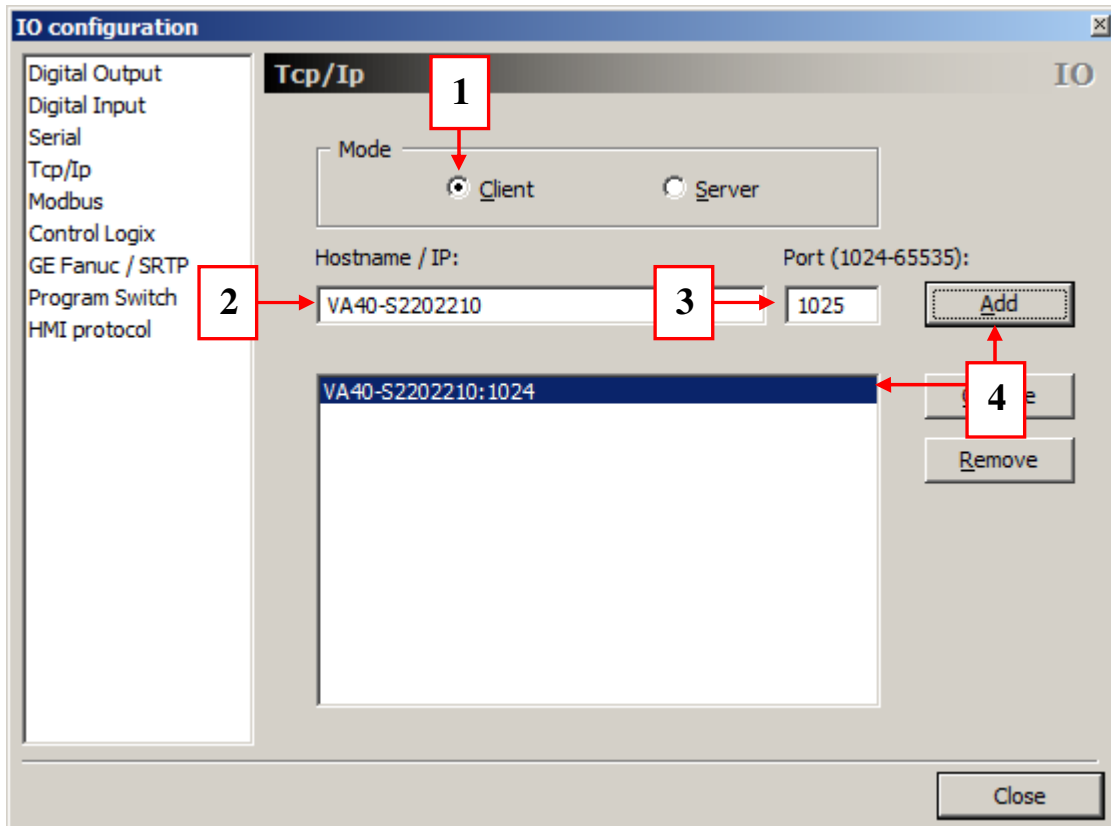
1. Select **Server** as the mode
2. Enter the number of the port to use for communication.
3. Click the **Add** button to add the connection to the list. (Note that the port number is automatically incremented when you click the **Add** button.)



Sherlock as a client

As a client, Sherlock reads data from and sends data to a server's port.

1. Select **Client** as the mode
2. Enter the name or IP Address of the server computer with which Sherlock will communicate. (By default, Sherlock's host computer name is shown as the **Hostname / IP**; you will need to change this.)
3. Enter the port number on the server computer that will be used for communication.
4. Click the **Add** button to add the connection to the list. (Note that the port number is automatically incremented when you click the **Add** button.)



Instructions

Sherlock provides five TCP/IP communication instructions, available in the **Instructions IO : Tcp/Ip** folder. All instructions can be invoked by a Sherlock server or client.

- **Receive Buffer:** Receive a specified number of ASCII characters sent from another application; no termination character is required. If more than the specified number of characters is sent by the other application, the extra characters are buffered; they will be the first characters read the next time the instruction is invoked.
- **Receive Byte Array:** Receive a specified number of bytes (integer numbers) and save them to an array; no termination character is required or expected. Positive values are truncated to the range 0 thru 255: 256 is set to 0; 257 is set to 1; etc. Negative values are converted to positive: -1 is set to 255; -2 is set to 254; etc.
- **Receive Line:** Receive an unspecified number of ASCII characters sent from another application; a specific character must terminate the string. The default termination character is ASCII 10, the line feed character. You can change this to any single character.
- **Send Byte Array:** Post an array of bytes (integer numbers) to the selected port for immediate transmission. Positive values are truncated to the range 0 thru 255: 256 is set to 0; 257 is set to 1; etc. Negative values are converted to positive: -1 is set to 255; -2 is set to 254; etc.
- **Send Line:** Post a string of ASCII characters with a termination character to the selected port for immediate transmission. The application that receives the line looks for the termination character to determine the completion of the transmission.

Examples

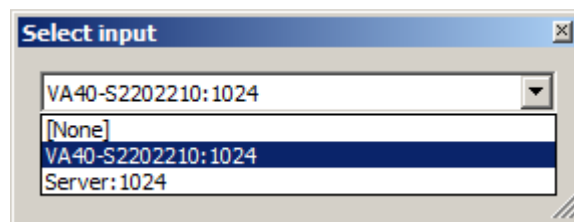
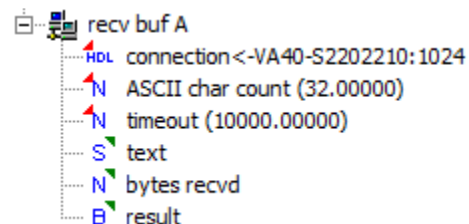
Receive Buffer

The Sherlock TCP/IP client waits up to 10 seconds (10,000 milliseconds) to receive 32 characters from port 1024 on the server computer named VA40-S2202210.

If 32 characters are received within 10 seconds, **result** returns **True**, and the characters are available in **text**.

If the instruction does not receive 32 characters within 10 seconds, **result** returns **False**, and **bytes recvd** returns 0. Any bytes received before the timeout are not available in **text**.

For the connection handle **HDL**, the TCP/IP client connection definition was selected.

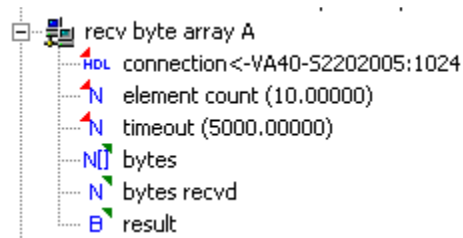


Receive Byte Array

The Sherlock TCP/IP client waits up to 5 seconds (5,000 milliseconds) to receive an array of 10 bytes from port 1024 on the server computer named VA40-S2202005.

If an array of 10 bytes is received within 5 seconds, **result** returns **True**; the numbers are available in **bytes**; and the number of bytes received is available in **bytes recvd**.

If the instruction does not receive an array of 10 bytes within 5 seconds, **result** returns **False**; **bytes** is empty; and **bytes recvd** returns 0.



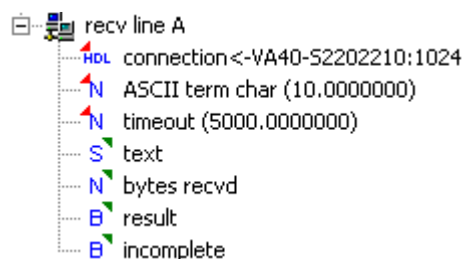
Receive Line


The Sherlock TCP/IP client waits up to 5 seconds (5000 milliseconds) to receive a string of characters terminated with a line feed character (ASCII decimal 10) from port 1024 on the server computer named VA40-S2202210.

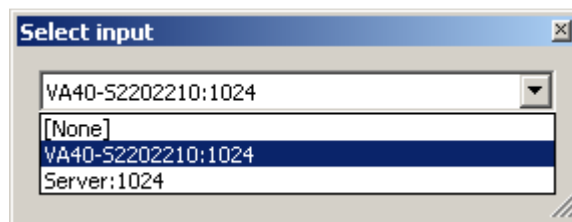
If a line feed termination character is received within 5 seconds, **result** returns **True**; **incomplete** returns **False**; the received characters, including the line feed character, are available in **text**; and **bytes recvd** returns the number of characters, including the line feed character. For example, if the server sends the line "ABC123n", **text** will contain ABC123{0xd} and **bytes recvd** will contain 7. (0xd is the hexadecimal representation of decimal 10. The braces {} denote that it is an ASCII value, not the three characters '0,' 'x' and 'd'.)

If a line feed character is not received within 5 seconds, but other characters are, the instruction times out, **result** returns **True**; **incomplete** returns **True**; and **bytes recvd** returns the number of characters received. The characters received before the timeout are available in **text**.

If absolutely no characters are received within 5 seconds, including a line feed character, **result** returns **False**; **incomplete** returns **True**; **bytes recvd** returns 0; and **text** is empty.

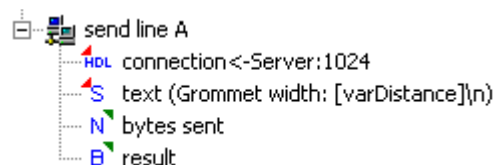


For the connection handle , the TCP/IP client connection definition was selected.




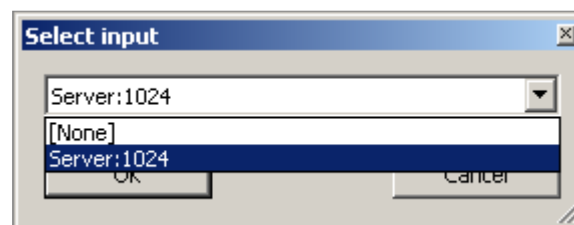
Send Line

The Sherlock TCP/IP server sends the string “Grommet width: [varDistance]” to port 1024. [varDistance] is replaced with the value of the variable *varDistance*. The string is terminated with “\n”, the ASCII line feed character.



If no application is listening to the specified port when the instruction is called, **result** returns **False**, and **bytes sent** returns 0.

For the connection handle , the TCP/IP server connection definition was selected.



Within a single Sherlock investigation, multiple servers and clients can be defined and active at the same time. However, a Sherlock server can be connected to only a single client at a time.



The Tcp/Ip server must be available when the client is created or initialized, or the client will generate a “failed connection” error. This is true of all Tcp/Ip clients, not just those in Sherlock investigations. For example, if you try to connect to a non-existent server from Hyperterminal, Hyperterminal displays a message box with the error “Unable to connect to 192.168.0.100 port 1024.”

In Sherlock, as soon as you create the server in the **Options → IO → Tcp/Ip** dialog, or load an investigation that includes a server specification, the server is available to be connected to a client.