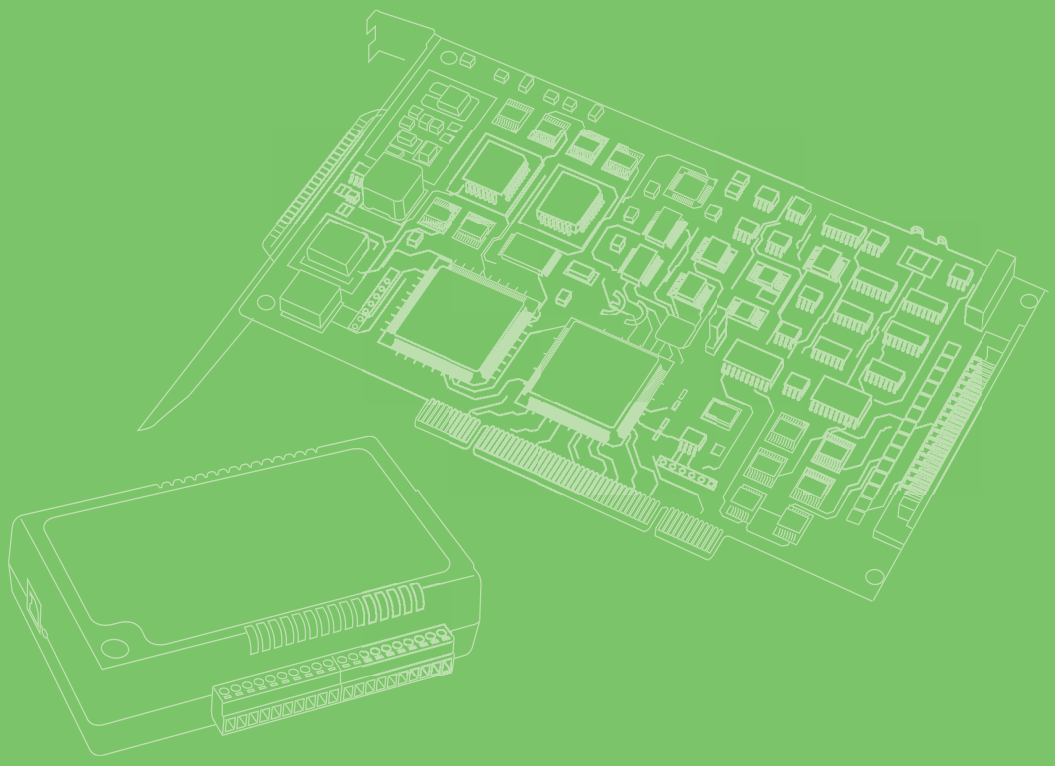
User Manual



Inspector Express

Scripting User Manual

ADVANTECH

Enabling an Intelligent Planet

Copyright

Acknowledgements

Intel and Pentium are trademarks of Intel Corporation.

Microsoft Windows and MS-DOS are registered trademarks of Microsoft Corp.

All other product names or trademarks are properties of their respective owners.

Product Warranty (2 years)

Advantech warrants to you, the original purchaser, that each of its products will be free from defects in materials and workmanship for two years from the date of purchase.

This warranty does not apply to any products which have been repaired or altered by persons other than repair personnel authorized by Advantech, or which have been subject to misuse, abuse, accident or improper installation. Advantech assumes no liability under the terms of this warranty as a consequence of such events.

Because of Advantech's high quality-control standards and rigorous testing, most of our customers never need to use our repair service. If an Advantech product is defective, it will be repaired or replaced at no charge during the warranty period. For out-of-warranty repairs, you will be billed according to the cost of replacement materials, service time and freight. Please consult your dealer for more details.

If you think you have a defective product, follow these steps:

1. Collect all the information about the problem encountered. (For example, CPU speed, Advantech products used, other hardware and software used, etc.) Note anything abnormal and list any onscreen messages you get when the problem occurs.
2. Call your dealer and describe the problem. Please have your manual, product, and any helpful information readily available.
3. If your product is diagnosed as defective, obtain an RMA (return merchandize authorization) number from your dealer. This allows us to process your return more quickly.
4. Carefully pack the defective product, a fully-completed Repair and Replacement Order Card and a photocopy proof of purchase date (such as your sales receipt) in a shippable container. A product returned without proof of the purchase date is not eligible for warranty service.
5. Write the RMA number visibly on the outside of the package and ship it prepaid to your dealer.

Part No. 2003102410 Edition 1

Printed in Taiwan March 2013

Declaration of Conformity

CE

This product has passed the CE test for environmental specifications when shielded cables are used for external wiring. We recommend the use of shielded cables. This kind of cable is available from Advantech. Please contact your local supplier for ordering information.

CE

This product has passed the CE test for environmental specifications. Test conditions for passing included the equipment being operated within an industrial enclosure. In order to protect the product from being damaged by ESD (Electrostatic Discharge) and EMI leakage, we strongly recommend the use of CE-compliant industrial enclosure products.

FCC Class A

Note: This equipment has been tested and found to comply with the limits for a Class A digital device, pursuant to part 15 of the FCC Rules. These limits are designed to provide reasonable protection against harmful interference when the equipment is operated in a commercial environment. This equipment generates, uses, and can radiate radio frequency energy and, if not installed and used in accordance with the instruction manual, may cause harmful interference to radio communications. Operation of this equipment in a residential area is likely to cause harmful interference in which case the user will be required to correct the interference at his own expense.

FCC Class B

Note: This equipment has been tested and found to comply with the limits for a Class B digital device, pursuant to part 15 of the FCC Rules. These limits are designed to provide reasonable protection against harmful interference in a residential installation. This equipment generates, uses and can radiate radio frequency energy and, if not installed and used in accordance with the instructions, may cause harmful interference to radio communications. However, there is no guarantee that interference will not occur in a particular installation. If this equipment does cause harmful interference to radio or television reception, which can be determined by turning the equipment off and on, the user is encouraged to try to correct the interference by one or more of the following measures:

- Reorient or relocate the receiving antenna.
- Increase the separation between the equipment and receiver.
- Connect the equipment into an outlet on a circuit different from that to which the receiver is connected.
- Consult the dealer or an experienced radio/TV technician for help.

FM

This equipment has passed the FM certification. According to the National Fire Protection Association, work sites are classified into different classes, divisions and groups, based on hazard considerations. This equipment is compliant with the specifications of Class I, Division 2, Groups A, B, C and D indoor hazards.

Technical Support and Assistance

1. Visit the Advantech web site at http://support.advantech.com.tw/ where you can find the latest information about the product.
2. Contact your distributor, sales representative, or Advantech's customer service center for technical support if you need additional assistance. Please have the following information ready before you call:
   - Product name and serial number
   - Description of your peripheral attachments
   - Description of your software (operating system, version, application software, etc.)
   - A complete description of the problem
   - The exact wording of any error messages

Warnings, Cautions and Notes

Warning! *Warnings indicate conditions, which if not observed, can cause personal injury!*

Caution! *Cautions are included to help you avoid damaging hardware or losing data. e.g.*

There is a danger of a new battery exploding if it is incorrectly installed. Do not attempt to recharge, force open, or heat the battery. Replace the battery only with the same or equivalent type recommended by the manufacturer. Discard used batteries according to the manufacturer's instructions.

Note! *Notes provide optional additional information.*

Document Feedback

To assist us in making improvements to this manual, we would welcome comments and constructive criticism. Please send all such - in writing to:
http://support.advantech.com.tw/

Packing List

Before setting up the system, check that the items listed below are included and in good condition. If any item does not accord with the table, please contact your dealer immediately.

- Item XXXXXXXX
- Box XXXXXXXXX

Safety Instructions

1. Read these safety instructions carefully.
2. Keep this User Manual for later reference.
3. Disconnect this equipment from any AC outlet before cleaning. Use a damp cloth. Do not use liquid or spray detergents for cleaning.
4. For plug-in equipment, the power outlet socket must be located near the equipment and must be easily accessible.
5. Keep this equipment away from humidity.
6. Put this equipment on a reliable surface during installation. Dropping it or letting it fall may cause damage.
7. The openings on the enclosure are for air convection. Protect the equipment from overheating. DO NOT COVER THE OPENINGS.
8. Make sure the voltage of the power source is correct before connecting the equipment to the power outlet.
9. Position the power cord so that people cannot step on it. Do not place anything over the power cord.
10. All cautions and warnings on the equipment should be noted.
11. If the equipment is not used for a long time, disconnect it from the power source to avoid damage by transient overvoltage.
12. Never pour any liquid into an opening. This may cause fire or electrical shock.
13. Never open the equipment. For safety reasons, the equipment should be opened only by qualified service personnel.
14. If one of the following situations arises, get the equipment checked by service personnel:
15. The power cord or plug is damaged.
16. Liquid has penetrated into the equipment.
17. The equipment has been exposed to moisture.
18. The equipment does not work well, or you cannot get it to work according to the user's manual.
19. The equipment has been dropped and damaged.
20. The equipment has obvious signs of breakage.
21. DO NOT LEAVE THIS EQUIPMENT IN AN ENVIRONMENT WHERE THE STORAGE TEMPERATURE MAY GO BELOW -20° C (-4° F) OR ABOVE 60° C (140° F). THIS COULD DAMAGE THE EQUIPMENT. THE EQUIPMENT SHOULD BE IN A CONTROLLED ENVIRONMENT.
22. CAUTION: DANGER OF EXPLOSION IF BATTERY IS INCORRECTLY REPLACED. REPLACE ONLY WITH THE SAME OR EQUIVALENT TYPE RECOMMENDED BY THE MANUFACTURER, DISCARD USED BATTERIES ACCORDING TO THE MANUFACTURER'S INSTRUCTIONS.
23. The sound pressure level at the operator's position according to IEC 704-1:1982 is no more than 70 dB (A).

DISCLAIMER: This set of instructions is given according to IEC 704-1. Advantech disclaims all responsibility for the accuracy of any statements contained herein.

Safety Precaution - Static Electricity

Follow these simple precautions to protect yourself from harm and the products from damage.

- To avoid electrical shock, always disconnect the power from your PC chassis before you work on it. Don't touch any components on the CPU card or other cards while the PC is on.
- Disconnect power before making any configuration changes. The sudden rush of power as you connect a jumper or install a card may damage sensitive electronic components.

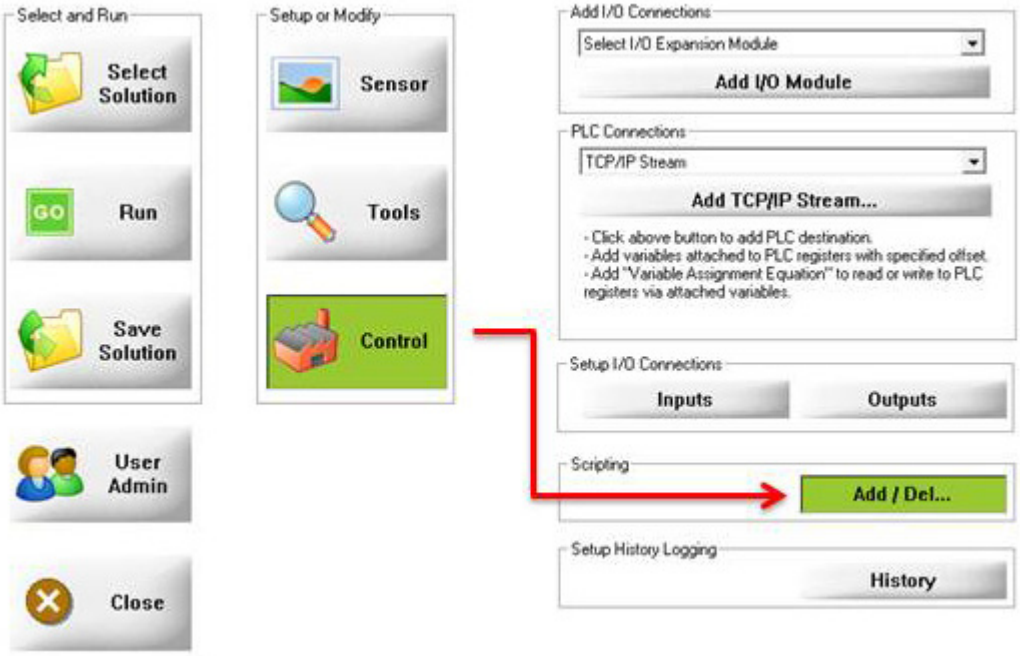
Contents

Chapter 1

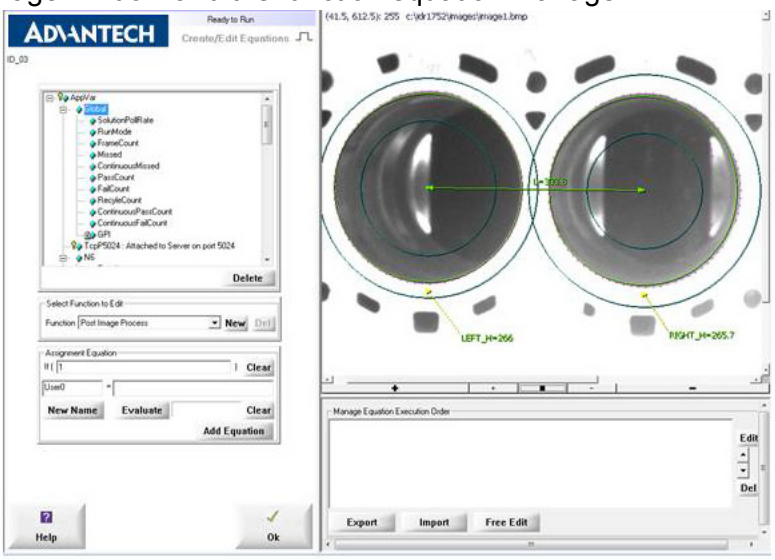
The Script Panel

1.1 Accessing the Script Tool

The Script Tool is mostly used for control and communication, which is why it is accessed through the control panel after the tools have been defined. Simply follow the clicks as shown below:



Inside the script panel there are 3 sections. The control panel on the left side contains a variable tree list, a function list and a basic script editor. On the right side are the image window and the function equation manager.



1.2 The variable Tree List

The variable tree at the top of the control panel lists all of the variables available to the script tool, namely:

- Tool outputs from user-defined measurements (i.e. MS1.Result)
- Predefined keywords (control words such as SOLUTION)
- Discrete I/O (physical inputs and outputs available to the device)
- PLC or communication registers. (user defined connections)
- User variables

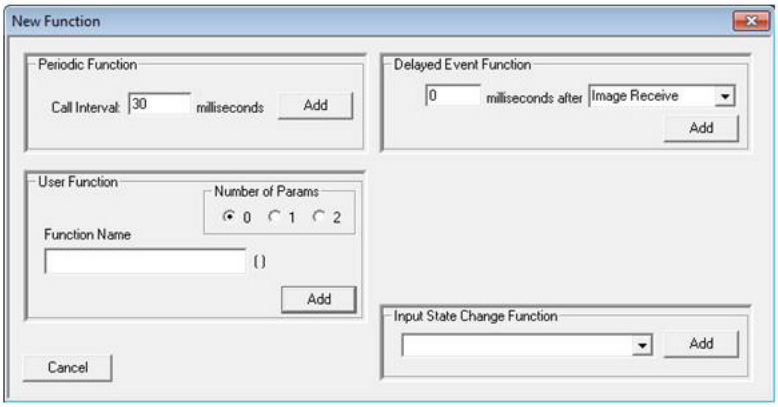
Variables can have Boolean meaning, such as TRUE/FALSE or ON/OFF, or they can store numbers or strings of user defined information.

1.3 The Function List

The Function list is available in the middle of the control panel in the form of a drop-down list. By default it contains predefined functions for solution initialization, as well as pre and post processing. These functions are empty when starting a new solution, meaning they do nothing until you add something to them.

To edit a function, select it in the function drop down list and add instructions using the basic or full script editors available.

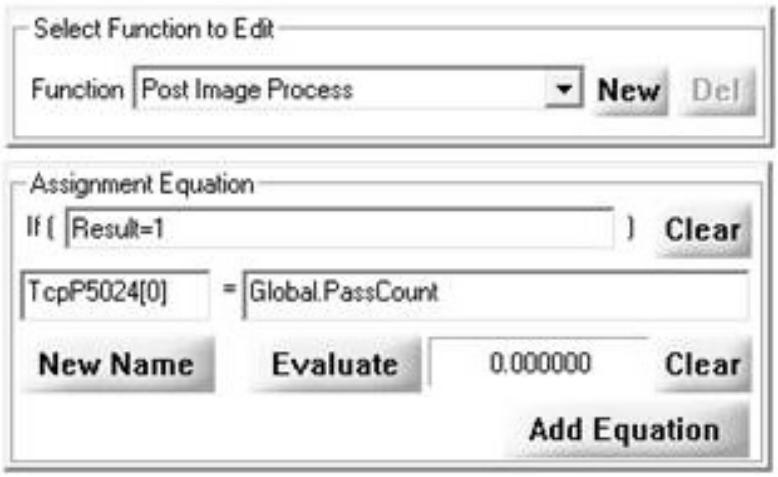
To add functions, click the “New” button and follow instructions in the popup menu. Added functions will appear in the drop down list above. Similarly, remove unwanted functions by selecting them in the drop down list and clicking the “Del” button.



The “New Function” interface shown above will be described in a later section.

1.4 The Basic Script Editor

Users new to programming can start with the simple script editor provided at the bottom of the control panel. It provides the basic means to add instructions to a function. Simply select the function to edit from the function drop down list, enter the condition, variable name and value, then click “Add Equation”. The new instruction will be added to the selected function and displayed in the Equation Manager on the right side of the panel (shown at the bottom of this page).



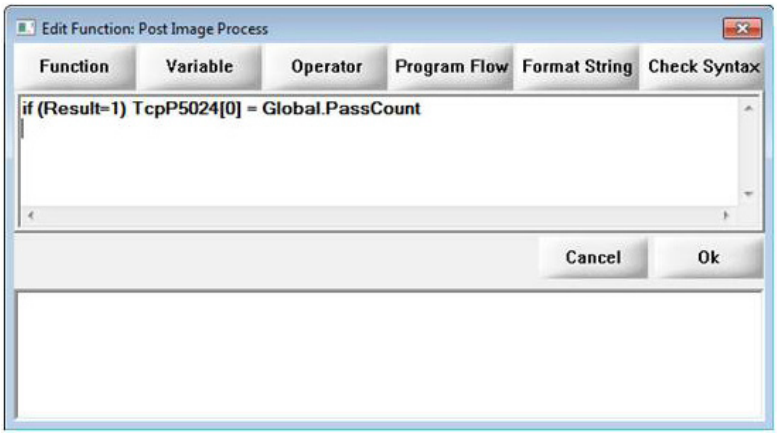
The above example shows how to add an instruction to the “Post Image Process” function. This instruction will send the cumulative pass count to a predefined TCP/IP connection if the current inspection result = pass.

When using the basic script editor, variables may be selected from the variable tree and dragged into the active edit boxes. Clicking the “New Name” button will add a user defined variable (i.e. User0) that is not currently defined in the function. Clicking the “Evaluate” button will process the equation and show you the result.

1.5 The Full Script Editor

The full script editor is accessed by clicking the “Free Edit” button at the bottom of the Equation Manager. When clicked, the free editor panel will pop-up as shown at the bottom of this page.

Equations associated with a function can be exported to a user defined text file by clicking the “Export” button. Similarly, equations stored in a local text file can be imported into a selected function by clicking the “Import” button. This is a convenient method of sharing scripts or script snippets (application specific code) across solutions.



The full script editor offers much more flexibility for designing and editing scripts. Most users quickly migrate to this method over the basic editor described previously.

The free editor provides drop-down menus for quick access to functions, variables, equation operators and program flow commands. A string editor is also included to simplify the formatting of communication strings to external devices. Clicking on a drop-down feature from the list will add it to the script. Manually add parameters and associated control statements to each script as required.

The Check Syntax button checks the “programmatic grammar” of the individual strings, but cannot parse the statements for runtime context errors that may occur outside this window.

1.6 Scripting Basics

1. Comment your code. This will make it much easier for you to debug should you need to come back to the solution at a later time. You can add comment lines using the prefix "//" (two forward slashes) for example: // Initialize the x array
2. All of your expressions, equations, and variables are saved in the Solution file, and persist when the Solution file is reloaded.
3. Statements are formed in a plain algebraic format, for example: **a = b+c**
4. Functions are called simply in the form: **z = aFunctionName(param1, param2)**
5. Variable names have a limit of 60 characters
6. A line of script has a limit of 256 characters. Numbers of lines are only limited by available memory.
7. Max time for the Pre-image Process function is **100ms**
8. Max time for Post Image Process function is **300ms** on EagleEye (1000ms on other platforms)
9. Max time all other functions are **1000ms**.
10. The time a Script takes can be reduced significantly by sending an array of data verses multiple read/writes (time to send an array is the same as a single piece of data). Bit functions can also be used.
11. It is good practice to always close “If” statements with “Endif”, even though this is not required for single command statements. This makes code easier to read and debug.
12. Once created, script variables will persist even if you delete them inside a script. A variable can be selected and deleted from the variable tree list or by starting a new solution.

Chapter 2

About Variables

There are 3 classes or types of variables that can be used in a script:

1. Global device variables – variables associated with external hardware such as I/O
2. Global control variables – variables that provide system wide status or control
3. Local user variables – variables associated with tools or functions defined by the user

2.1 Special Global Variables

Some special variables are not visible in the variable tree or the free editor. These variables are as follows:

- **Solution** - Global command variable that loads a new solution when written (i.e. Solution = 5 would load solution05.bin)
- **RelearnIndex** - Specifies an external input that is sampled when a trigger occurs to initiate a runtime tool relearn.
- **ShowPreprocessed** - Enables (=1) or disables (=0) display of preprocessing in ROI at runtime
- **RelearnIndex** - uses one of the General Purpose Inputs to trigger retraining or relearning specific tools (Match, Barcode, 2D-Code, Locator). **RelearnIndex=1** will execute a relearn (in the next acquired image) when GPI (1) is asserted. This statement should be added in the *Solution Initialize* function. To assign a tool to relearn, create a new **Relearn** variable. *For example*, to assign Barcode1 to be retrained, **Bar1.Relearn=1**. The Barcode tool will be retrained (in the next image) when the input assigned by RelearnIndex GPI(1) is asserted.

Note! The inputs on the EagleEye read 1 when nothing is connected, or when the input is lower than the logic threshold. You must be careful to connect the input so that GPI(1) will only read 1 when relearn is desired, or use the **RelearnOnZero** variable.

- **RelearnOnZero** - **RelearnOnZero =1** forces relearn to occur when the defined Relearn input (RelearnIndex) is 0, not 1. This statement should be added in the *Solution Initialize* function. ***NOTE:*** GPI(0) is the EagleEye trigger input. GPI(1) is the EagleEye IN0.

The **Result** variable returns the result from each defined camera, as well as the overall inspection result. **Result.0** variables return the result before an action is decided (i.e. it is made available so that user-defined functions can use it to define actions).

Result=1 (PASS)
Result=2 (RECYCLE)
Result=3 (FAIL)

You can override (*not typical for most applications*) the composite result, by adding variables “PASS”, “RECYCLE”, “FAIL”, (must be all capitals) and setting their values to 1 (for TRUE) or 0 (for FALSE).

Note! In formation of (creating or composing) the final Result, FAIL supersedes RECYCLE, and RECYCLE supersedes PASS. Put your tests for PASS, FAIL, RECYCLE in the "Post Image Processing" function.

2.2 Persistent Variables

When a solution is saved, any associated variables are saved also. When a solution is loaded, only its associated variables are loaded. This means that variables associated with the previous solution will be deleted unless they are also saved in the new solution. The exception to this rule is **Persistent Variables**. These special variables will persist even if the solution that created them is replaced by a different solution. Persistent variables are therefore used more for system (EagleEye) variables as opposed to solution specific variables. Persistent variables will persist until the EagleEye camera is power cycled. Persistent variables are defined with a **Prog** prefix and can also be saved in a solution file (i.e. Prog.myvariable).

2.3 Variable Basics

1. Use square brackets for variable names, especially names with spaces in them. Notice that variable names inserted or dragged into a field, are enclosed in brackets.
2. All of your expressions, equations, and variables are saved in the Solution file, and persist when the Solution file is reloaded.
3. User added variables belong to the current Solution. Loading a different Solution will cause your user variable set to be replaced with the set belonging to that new Solution.
4. There are many pre-defined variables with special meaning for use in scripts. You can also create your own variables. Referencing a variable automatically creates or instantiates that variable. A separate step for creating or declaring, is not necessary.
5. A complete listing of predefined variables follows on the next page.

2.4 Variable Listing

- **Result.0** - the value of Result, before it is output. This allows equations to evaluate the Result, before the decision is sent to the monitor, decision I/O and other mechanisms. **Result.0** returns 3 values: 1=Pass, 2=Recycle, 3=Reject.
- **Result** - the result of all measurements (the "composite result"). This result is sent to the Monitor, decision I/O and other communication mechanisms (such as PLC, Ethernet, serial port, etc.). **Result** returns 3 values: 1=Pass, 2=Recycle, 3=Reject. Each measurement also has a Result (i.e. L1.Result).
- **Global.GPI[#]** - a general purpose input. The Camera treats and evaluates all inputs as a steady state logic input. ***NOTE:*** GPI(0) is the EagleEye Trigger input. GPI(1) is the EagleEye IN0 input. When using the PL-200, GPI(8) is the Trigger input.
- **Global.GPO[#]** - a general purpose output. Normally, the outputs are held high or low, until the next result is available. (The Pass/Recycle/Fail decision outputs are pulsed.) You can use the pulse function or use the Delayed Event Function to create a pulse output.
- **Global.RunMode** - the current run-state or running mode. 0=running, 1=stopped.
- **Global.FrameCount** - number of frames or images acquired since a Solution was loaded, or since the statistics was reset ("Reset Statistics" button on the Monitor panel).
- **Global.Missed** - number of missed parts or frames.
- **Global.ContinuousMissed** - number of parts or frames missed in a row, or one after another.
- **Global.PassCount** - the value of the Pass counter, or the number of Passed parts.
- **Global.FailCount** - the value of the Fail counter, or the number of Failed parts.
- **Global.RecycleCount** - the value of the Recycle counter, or the number of Recycled parts.
- **Global.ContinuousPassCount** - the number of parts or frames passed in a row, or one after another.
- **Global.ContinuousFailCount** - the number of parts failed in a row, or one after another.
- **SolutionPollRate** – used with the PL-200. Polls the solution index inputs at the specified frequency to determine if a solution switch is requested.

Chapter 3

About Functions

Variables are manipulated using functions. Functions are made up of equations or instructions that affect an outcome or result. Most functions can be shared or called by other functions (like subroutines). Some functions are executed in order (pre-ordered functions), while others are based on a user defined event, such as a time interval or transition on a Global Input. Pre-ordered functions are a special class of functions that execute in a pre-defined order. They can call other functions, but can not be called by other functions.

Inspector Express includes a library of pre-defined functions for analysis, system control and communication. A complete listing of predefined functions starts on page 20.

3.1 Pre-Ordered Functions

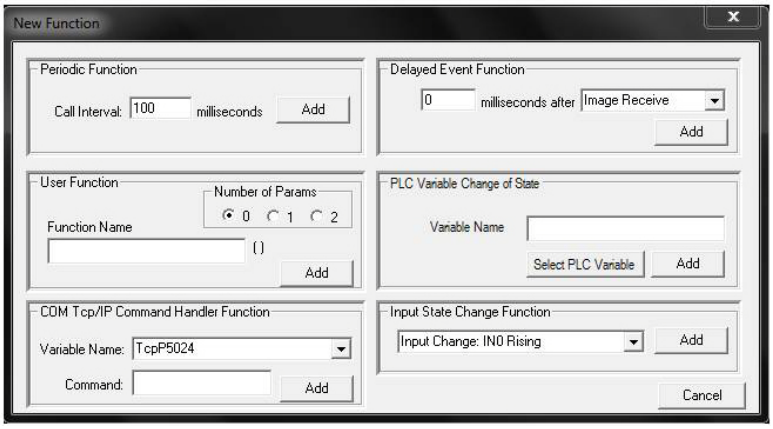
Every solution has 3 pre-ordered functions which execute in the order below:

1. **Solution Initialize** – called immediately after a solution is loaded. Typically used to initialize variables to a known state.
2. **Pre-Image Processing** – called immediately after a new image is received, but before processing begins. Can be used to handshake with other devices or control external I/O.
3. **Post-Image Processing** – Called immediately after processing. Typically used to formulate results and communicate with external devices.

3.2 Event Functions

Event functions offer design flexibility and control for applications that are based on asynchronous events, such as state changes in PLC registers or transition changes on input pins. Event functions do not execute in a predefined order (except for delay functions), but get called when the defining event occurs. When using event functions, care should be taken to avoid unintended conditions associated with asynchronous events.

Click on the New button next to the function list to open the "New Function" setup screen.



- The scripting interface supports 6 types of event functions. Simply define the event that best suits your application need and click the "Add" button. The new empty function will then appear in the function list ready for editing.

3.3 Periodic Function

This is a function that is called at a fixed user defined time interval. The Periodic function is often used to sample external Inputs or PLC registers for the following:

- Checking for a solution switch action
- Providing a system online heartbeat
- Reset variables or restart inspection in the event of a STOP condition

Note! *Accessing PLC registers can be slow (20ms per register is not unrealistic). As a consequence, we recommend defining a periodic interval greater than 100ms for most applications.*

3.4 User Function

The user function is called on demand from other functions, such as the image pre and post processing functions. User functions perform the following purpose:

They define application specific processing functions with parameter passing They provide a means to partition code into smaller, more manageable sub-routines.

Note! *Every user function must include a "return()" statement*

3.5 Delayed Event Function

These functions are called after a special image event occurs:

- New image is received into memory
- Processing of the current image is complete
- On a software trigger

Delayed event functions are typically used to complete I/O events that were initiated during the image pre and post process functions. These functions are time synchronized with pre and post image processing functions.

3.6 PLC Change of State Function

These functions are called when a change of PLC register value or state is detected. A PLC has to be defined for this function to work. This applies only to Ethernet/IP and Modbus Slave PLCs (not a Modbus Master). These functions can be used for the following:

- Handshaking and control of 3rd party equipment
- Triggering EagleEye to take a picture under PLC control when the object to be inspected is in place.

3.7 The Input State Change Function

These functions are called when a selected Input line changes state i.e. from low to high or high to low. These functions can be used for the following:

- I/O control
- Solution switching
- Triggering using an input line to generate a software trigger

3.8 The COM TCP/IP Command Handler Function

Defines a group of functions or statements that are called to handle a specific command received on a specific COM Port or TCP/IP Connection.

When the Command Handler Functions are used, the command format used on the connection must follow the following rules:

All commands are ASCII text (binary data is not allowed).

The Command format is:

```
      command [optional parameters separated by spaces]
(carriage return and /or line feed)
```

Note! *"\r" = carriage Return character.*

"\n" = Line Feed character.

The Command Handler function is added in the "New Function" menu, and is associated with the specified connection and command. Where "Variable Name" specifies the connection, and "Command" specifies the command.

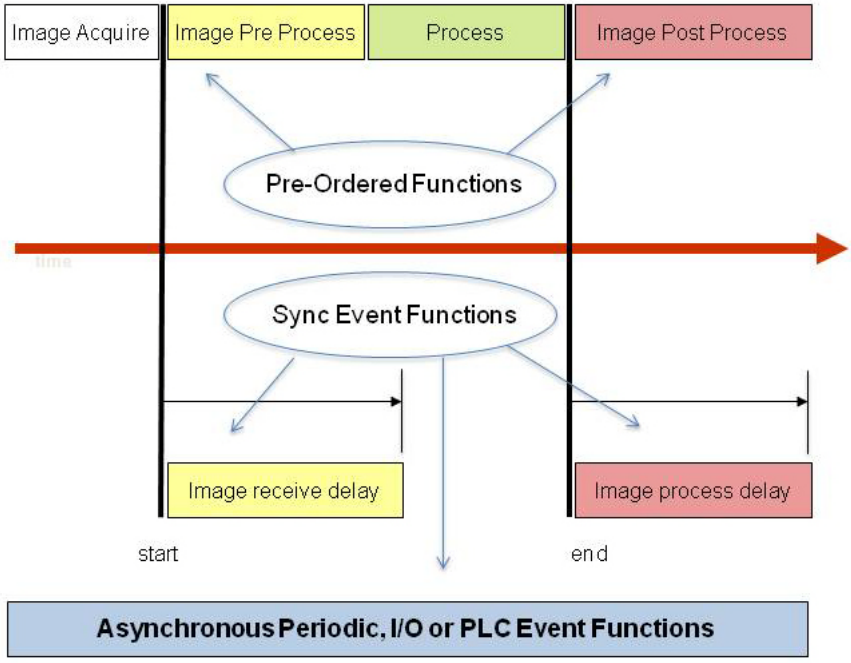
There are 3 special variables which can be used in the command handler function:

`argc` - the number of command parameters received.

`argv` - an array of strings which contain the parameters received.

`comvar` - a special variable which can be passed to the function WriteString to specify a destination which is the same as the source of the command received.

3.9 Function Timing



The above diagram depicts the timing associated with functions. Pre-ordered functions are synchronized to the start and of end of each inspection. The special event functions **image receive delay** and **image process delay** are also synchronized. Other functions shown in the blue box may be used to control asynchronous events.

3.10 Function Listing

Note! *Please consult the product user manual for the most up-to-date listing.*

3.10.1 Math Functions

- **sin**(radians) - result is the sine of the argument *radians*. The argument must be in radians.
- **cos**(radians) - result is the cosine of the argument *radians*. The argument must be in radians.
- **tan**(radians) - result is the tangent of the argument *radians*. The argument must be in radians.
- **asin**(x) - result is the arcsine of *x* in the range -p/2 to p/2 radians, where: -1 <= x <= 1.
- **acos**(x) - result is the arccosine of *x* in the range -p/2 to p/2 radians, where: -1 <= x <= 1.
- **atan**(x) - result is the arctangent of *x* in the range -p/2 to p/2 radians.
- **atan2**(y, x) - result is the arctangent of *y/x* in the range -p to p radians.
- **exp**(x) - result is the exponential value of *x*.
- **logn**(x) - result is the natural logarithm of *x*.
- **sqrt**(x) - result is the square root of *x*.
- **pow**(x, y) - result is *x* raised to the power of *y*.

3.10.2 String and Character Functions

The following String functions were added to find numeric characters in a string (the result of a OCR, barcode or 2-D code read), and convert to numbers (for passing to other equations or peripherals). Other uses are possible.

- **Find** (substring, inString) - finds the first occurrence of substring in the input inString, and returns the zero-based index location of the first matching character. Returns -1 if no match was found. Spaces are counted.

  Example: **idx = find(“00”, “SM WRA 0057 4321")** returns 7, or sets idx = 7.
- **Substring** (string, startIndex, length) - forms a sub-string from the input string, beginning at startIndex (zero-based) of length characters. If length = 0 all characters to the end of the string are included in the sub-string.

  Example: **s2 = substring(“SM WRA 0057 4321", 9, 0)** returns string “57 4321" in s2.
- **StrLen** (string) - returns the number of characters in a string.
- **GetChar** (string, index) - returns the character located at index (zero-based) in the string.
- **SetChar** (string, index, char) - sets the character in string, located at index (zero-based), to char.
- **int**(string) - converts the input *string* (of numbers) to an integer value.
- *Example:* **x = int("33")** sets x = 33
- **float**(string) - converts the input *string* (of numbers) to a floating point value.
- *Example:* **x = float("57.499")** sets x = 57.499
- **char**(int) - converts the input integer *int* to a character.
- **string**(int) - converts the input number *int* (base 10) to a string.

3.10.3 Tool Statistics and Attribute Functions

- **GetMean**(measurementVar) - returns the arithmetic mean for the specified measurement.
- *Example*: **L1Mean = GetMean (L1)**
- **GetStdDev**(measurementVar) - returns the standard deviation for the specified measurement.

  Example: **L1StdDev = GetStdDev (L1)**
- **GetMin**(measurementVar) - returns the minimum value which has occurred for the specified measurement.

  Example: **L1Min = GetMin (L1)**
- **GetMax**(measurementVar) - returns the maximum value which has occurred for the specified measurement.

  Example: **L1Max = GetMax (L1)**
- **ResetVarStats**(measurementVar) - resets the measurement statistics (min, max, mean, std dev) for the specified measurement. All prior data samples for the measurement are cleared out for the statistical calculations.

  Example: **ResetVarStats (L1)**
- **GetToolType**(measurementVar) - returns a number indicating the type of measurement tool, for the specified variable.

  Example: **L1type = GetToolType(L1)** variable L1type returns the value 6.

- **GetNthToolType**(varIndex, CamID) - returns a number indicating the type of measurement tool, for the specified variable index and camera ID. Returns zero if no tool exists for *varIndex* and *camID* (0 for EagleEye)
- *varIndex* – 0 to (number of tools minus 1) which exist, for the specified camera.
- **GetToolName**(varIndex, camID) - returns the simple name (no camera prefix) of the measurement tool, for the specified variable index and camera ID (0 for EagleEye)

  varIndex – 0 to (number of tools minus 1) which exist, for the specified camera.
- **RequestRelearn**(measurementVar) - causes *measurementVar* to be relearned on the next image.

  Example: **RequestRelearn(L1)**
- **SetTolerances**(measurementVar, toleranceArrayIn) - sets the 5 tolerance "pivot points" for the specified measurement variable.

  measurementVar – variable name.

  toleranceArrayIn – 5 element array of tolerance values. (as shown below, under GetTolerances).
- **GetTolerances**(measurementVar, toleranceArrayOut) - gets the 5 tolerance "pivot points" for the specified measurement variable.

  measurementVar – variable name.

  toleranceArrayOut – 5 element array of tolerance values:

  Index Content

  0 Minimum recycle value

  1 Minimum pass value

  2 Perfect value

  3 Maximum pass value

  4 Maximum recycle value

- **SetNthTolerances**(varIndex, camID, toleranceArrayIn) - sets the 5 tolerance "pivot points" for the specified variable index and camera ID (0 for EagleEye)

  varIndex – 0 to (number of tools minus 1) which exist, for the specified camera.

  toleranceArrayIn – 5 element array of tolerance values. (as shown above, under GetTolerances).
- **GetNthTolerances**(varIndex, camID, toleranceArrayIn) - gets the 5 tolerance "pivot points" for the specified variable index and camera ID (0 for EagleEye)

  varIndex – 0 to (number of tools minus 1) that exist

  toleranceArrayOut – 5 element array of tolerance values. (as shown above, under GetTolerances).
- **GetToolValue**(toolName) - returns the measurement value for the tool. The simple tool name ("L1") is passed.

  Example: **value = GetToolValue ("L1")**
- **GetToolResult**(toolName) - returns the Result value for the tool. The simple tool name ("L1) is passed.

  Example: **value = GetToolResult ("L1")**
- **GetVarDimension**(varName) - returns the number of children variables of the variable varName.
- **WriteVar**(narName, value) - write a value to a script variable.

  varName - the name of the variable to write to.

  value - the value to write to the variable. *Example:* **WriteVar("InputThreshold", 4.5)**
- **ReadVar**(varName) - Read a script variable's value.

  varName - the name of a script variable.

3.10.4 Digital IO/Acquisition Control Function

- **pulse**(activeVal, offsetMillisec, durationMillisec) - generates a pulse output.

  activeVal – 1=active-high pulse, 0=active low-pulse.

  offsetMillisec – offset or delay from the moment this statement executes, in milliseconds.

  durationMillisec – duration of the pulse, in milliseconds.

  Example: **Global.GPO[1] = pulse(1,5,10)**

  outputs on GPO1 an active-high pulse of 10 ms duration and offset 5 ms after the statement executes.
- **trigger**( ) - generate an image trigger signal. The Sensor Trigger must be set to "Inspection Trigger" when using this function.
- **ReTrigger**(camID) - causes re-processing the last image on the indicated camID.

  camID – always 0 for the EagleEye.
- **TriggerSource**(source) - set the trigger source or trigger mode.

  source – 0=freerun, 1=internal timer, 2=external trigger, 3=software
- **SetExposure**(exposureTimeMilliseconds) - sets the image exposure time in milliseconds.

  Example: **SetExposure(9.6)** sets the exposure time to 9.6 milliseconds.
- **SetBrightness**(percentX100) - sets the image brightness. The value is a percent, times 100: a value of 60 is 60%.
- **SetContrast**(percentX100) - sets the image contrast. The value is a percent, times 100: a value of 60 is 60%.

- **GetExposure**() - returns the current value of exposure, in milliseconds.
- **GetBrightness**() - returns the current value of brightness.
- **GetContrast**() - returns the current value of contrast.
- **SetImageSource**(imageSource) - sets the source of images to be processed.

  imageSource = 0 - image source is the acquisition device.

  1 - image source is the image file store, the subdirectory \Images in EagleEye.

3.10.5 Logging Functions

- **LogStart**(fileName, onClient) - Start logging the processed frame data to the specified CSV (comma separated values) file.

  fileName – full path and file name to save CSV data to. *For Example*, C:\Logs\iHistlog119200614.csv

  onClient – if **0**, save to file on server; if **1**, save to file on client.
- **LogStop**( ) - Stop logging data that was started by a **logstart** call.
- **LogImage**(fileName) - saves the image to a file. Only works if image logging is enabled. Allows you to substitute a different name for one image. The image log reverts to the name defined in the Communication panel after one image.

  fileName – full path and file name to save the image to.
- **WriteImageFile**(fileName, camID) - To be called only from the "Post Image Process" function, will write the current image from the camera specified by *camID* to the *fileName* specified (0 for EagleEye)

  fileName – full path of file to save. Can use UNC format: \\server Name\shareName\path\fileName to save to a server's shared drive. Use function **DriveConnect**() to connect to a remote server's share drive.
- **WriteImageTools**(fileName, camID) - writes an image file including the tool graphics.

  fileName – full path of file to save. Can use UNC format to save to a servers shared drive: \\serverName\shareName\path\fileName. Use function **DriveConnect**() to connect to a remote server's share drive. *camID* – always 0 for the EagleEye..
- **DriveConnect**(Password, UserName, ServerPath) - connect to a remote server's share. Use in conjunction with **WriteImageFile** or **WriteImageTools** to connect to a remote drive that requires a user name and password login to access.

Note! **DriveConnect** should be called before each **WriteImageFile** or **WriteImageTools** call that writes to a remote drive, to ensure the remote drive stays connected.

Note! **The Inspector Express software also supports ftp file logging, where Inspector Express is the ftp Client logging to an ftp Server. The ftp File Name syntax is:**

ftp://userName:password@host/path

The userName and password are optional in Inspector Express. These may be required by your ftp server.

- **GetFtpFileStatus**() - returns the status of a file on the FTP device:

  0 = idle.
  1 = busy transferring file.
  2 = error in ftp connection.

3.10.6 TCP IO Function

- **WriteFormatString**(commVar, formatString) - Writes a formatted string to the TCP/IO connection specified by *comVar*. See also string formatting reference

  Example: **WriteFormatString(TcpP5025 ,"\n\rLC1 = [LC1]")**

  Example: **WriteFormatString(TcpP5025 , "\n\rLC1 = [LC1%0.3f], L1 = [L1%d]")**
- **WriteString**(comVar, String) - Writes a string to the TCP/IO connection specified by *comVar*. Unlike WriteFormatString, WriteString does not perform embedded variable evaluations. See string formatting reference for special characters supported.

  Example: **WriteString(TcpP5025 ,"\n\rThe measurement is correct")**
- **WriteBytes**(comVar, byteArray, numBytes) - Writes a byte array to the TCP/IP connection specified by *comVar*.
- **ReadByte**(comVar) - Reads the next byte from the TCP/IP connection specified by *comVar* if one is available, otherwise returns 0 immediately.
- **ReadString**(comVar, endingChar) - Reads a string from the TCP/IP connection specified by *comVar* if one is available, otherwise returns an empty string immediately.

  endingChar – Specifies the char which must have been received to signal the end of a received string.
- **IsConnected**(comVar) - Determines the connection state of the TCP/IP connection specified by *comvar*. Returns **1** if connected. Returns **0** if disconnected.

3.10.7 Bit Functions

- **SetBit**(value, bitPosition) - Sets the bit at *bitPosition* in *value*. Returns the new *value*.

  value – a valid number (or variable) to be manipulated on a bit level.

  bitPosition – location of the bit to be set.
- **ClearBit**(value, bitPosition) - Clears the bit at *bitPosition* in *value*. Returns the new *value*.

  value – a valid number (or variable) to be manipulated on a bit level.

  bitPosition – location of the bit to be cleared.
- **GetBit**(value, bitPosition) - tests and returns the state of the bit at *bitPosition* in *value*. Returns the bit state.

  value – a valid number (or variable) to be manipulated on a bit level.

  bitPosition – location of the bit to be tested.

3.10.8 System / Misc. Functions

- **Copy**(source, dest, numElements) - Copy *numElements* from *source* (an array of elements) to *dest* (an array of elements. The copy function can be used to cause multiple PLC registers to be updated in a single transaction.
- **ResetHistory**( ) - clears the history log of stored images and data.
- **ResetStatistics**( ) - clears the pass/recycle/reject counters.
- **SetDisplayStatus**(statusMsg, color) - Sets the message to be displayed in the Inspection Status box (in the Configuration and Status panel associated with the Monitor panel). This overrides the display of “Pass” or “Fail”.

  statusMsg – the string that will be displayed. For multiple lines, add the character \n to indicate a new line. Message text is automatically sized to be the largest possible yet be contained by the Inspection Status box. String formatting information for variables of the form [Var%FormatData] is also supported. **msg1="[L1%0.2f]"** means display the value of L1 with 2 digits to the right of the decimal point. See also string formatting reference.

  color – The string name of the color the message text is to appear in. Possible values are: "black”, “red”, “green”, “yellow”, “blue”, “magenta”, “cyan”, “white”, “darkred”, “darkgreen”, “darkyellow”, “darkblue”, “darkmagenta”, “darkcyan”, “lightgray1”, “moneygreen”, “skyblue”, “cream”, “lightgray2”, “mediumgray”.
- **TimeMillisec**( ) - returns the current time in milliseconds.
- **GetTime**( ) - returns a value representing the current date and time. The value returned equals the number of milliseconds since January 1, 1601 (in the local time zone). See function FormatTime().
- **GetTimeString**( ) - returns a string value representing the current date and time (local time zone).

  Example: **now = GetTimeString()** sets "now" to a string value "7/9/2009 16:25:28:429"
- **FormatTime**(timeVal) - converts a time value in milliseconds since January 1, 1601 to a string representing the current date and time.

  Example: **time1 = GetTime()**
- **dateString = FormatTime(time1)** sets "dateString" to a string value "7/9/2009 15:25:28:429"
- **GetVersion**() - returns the firmware version of the EagleEye (same as the software version of Inspector Express.
- **StartInspect**( ) - start image inspection.
- **StopInspect**( ) - stop image inspection.
- **SwitchingIsEnabled**( ) - Returns 1 if the Solution switching is enabled. Returns 0 if Solution switching is not enabled.
- **GetPixel**(camID, x, y) - Returns the value for the pixel specified by camID, x, and y. If the pixel is color, pass the returned value to GetColor to get a specific primary color value.

  camID - always 0 for the EagleEye.
  x - the x coordinate. 0 = left most column.
  y - the y coordinate. 0 = top row.
  Example: **centerPix = GetPixel(0, 320, 240)**
- **GetColor**(colorID, pixelValue) - Returns the specified primary color value.

  colorID - Specifies primary color desired.
  0 – Blue
  1 – Green

2 – Red

pixelValue - Value for a color pixel, for example a value returned by GetPixel.

Example: **centerPix = GetPixel(0, 320, 240)**

centerRed = GetColor(2, centerPix)

- **return**(FunctionReturnValue) - Returns the specified value *FunctionReturn-Value* from a User defined function.

Example: **Return((p1 + p2) /2)**

- **SetImageEncode**(encodeMethod) - Change the image encoding used to compress images sent to a connected client.

encodeMethod = 0 – No compression.

1 – JPEG compression (default for color).

2 – Proprietary high speed compression (default for mono).

- **AutoSaveEnable**(enable) - Turn solution auto-save on or off. If auto-save is on, when a user exits the top level (main) setup panel, the solution is automatically re-saved, and solution switching is automatically re-enabled.

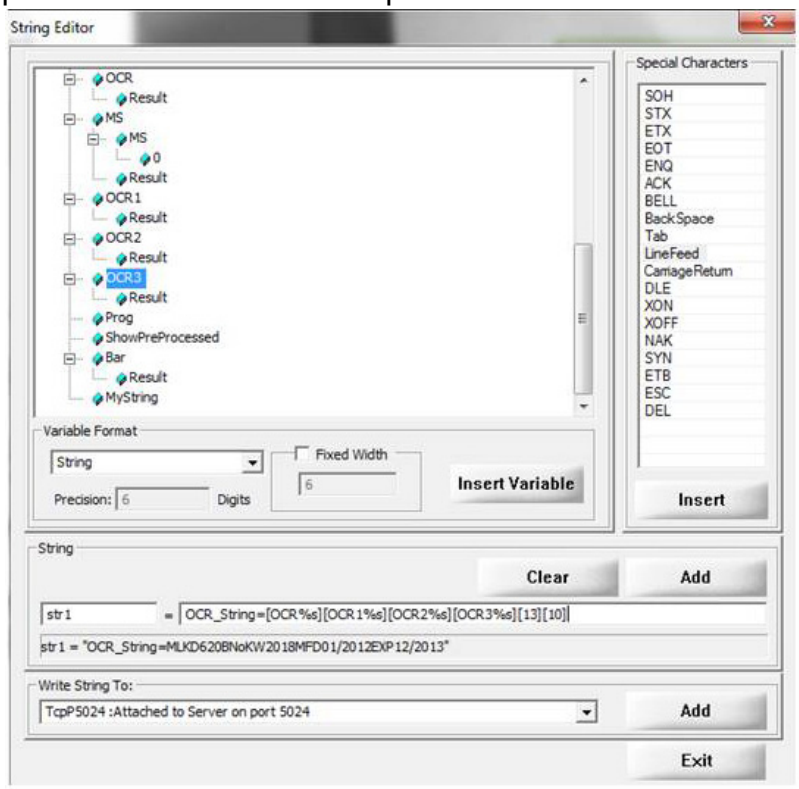
enable = 0 – turn off auto-save.

1 – turn on auto-save.

Chapter 4

The String Editor

The string editor simplifies construction of output strings. The GUI is accessed through the “Free Edit” script tool and allows users to define strings without writing code. Each string is composed of user text, formatted program variables and delimiting characters. Strings can be written to the function being edited and attached directly to a predefined communication port:



Strings are constructed by entering text or dragging variables into the string command line. The above example generates the following lines of code in the function being edited (typically the Post Image Process function):

str1 = "OCR_String=[OCR%s][OCR1%s][OCR2%s][OCR3%s][13][10]"
WriteFormatString(TcpP5024, str1)

4.1 String Formatting Reference

- Special Characters used in strings

\n - Line feed
\r - Carriage return
\t - Horizontal tab
\f - Form feed
\v - Vertical tab
\xhh - specify a hex byte, for example \xa6 Use the byte value a6.
Other control or non-printing characters can be formed using the \x with the hex value for the ascii character. For example \x04 for EOT, \x07 for Bell, \x00 for Null.

- Formatting Variables used in the WriteFormatString function

The general form follows this pattern. Please note items inside braces { }

are optional.
[VariableName %{width} {.precision} type] Example: **[LC1 %4.3f]**

The optional fields, which appear before the type character, control other aspects of the formatting, as described below.

- **Type**

  A required character that determines whether the associated argument is interpreted as a character, a string, or a number. Supported character types:

  c specifies a single-byte character.
  d Signed decimal integer.
  i Signed decimal integer
  u Unsigned decimal integer.
  x Unsigned hexadecimal integer, using "abcdef".
  X unsigned hexadecimal integer, using "ABCDEF".
  e Signed value having the form [–]d.dddd e [*sign*]dd[d[where d is a single decimal digit, dddd is one or more decimal digits, dd[d] is two or three decimal digits depending on the output format and size of the exponent, and *sign* is + or -.
  f Signed value having the form [–]dddd.dddd, where dddd is one or more decimal digits. The number of digits before the decimal point depends on the magnitude of the number, and the number of digits after the decimal point depends on the precision.
  g Signed value printed in **f** or **e** format, whichever is more compact for the given value and precision. The **e** format is used only when the exponent of the value is less than -4 or greater than or equal to the precision argument. Trailing zeros are truncated, and the decimal point appears only if one or more digits follow it.
  G Identical to the **g** format, except that **E**, rather than **e** introduces the exponent (where appropriate).
  s specifies a single-byte character string. Characters are printed up to the first null character or until the *precision* value is reached.

- **width**

  Optional number that specifies the minimum number of characters output. The width argument is a nonnegative decimal integer controlling the minimum number of characters printed. If the number of characters in the output value is less than the specified width, blanks are added to the left until the minimum width is reached. If width is prefixed with 0, zeros are added until the minimum width is reached.

- **precision**

  Optional number that specifies the maximum number of characters printed for all or part of the output field, or the minimum number of digits printed for integer values.

For types: **d, i, u, o, x, X**

The precision specifies the minimum number of digits to be printed. If the number of digits in the argument is less than *precision*, the output value is padded on the left with zeros. The value is not truncated when the number of digits exceeds *precision*.

The default precision is 1.

- ***For types:*** **e, E**

The precision specifies the number of digits to be printed after the decimal point. The last printed digit is rounded.

The default precision is 6. If *precision* is 0 or the period (.) appears without a number following it, no decimal point is printed.

- ***For types:*** **f**

The precision value specifies the number of digits after the decimal point. If a decimal point appears, at least one digit appears before it. The value is rounded to the appropriate number of digits.

Appendix A

Scripting Examples

The following examples demonstrate usages of the script tool. Each example has a brief description of the application and associated code snippets from the relevant functions. These examples cover basic scripting concepts only that apply to typical applications.

A.1 Manipulating Outposts

Many applications require some sort of output manipulation. By default EagleEye outputs are set to the “soft pulse” setting which automatically generates output pulses on GPO[0] and GPO[1] for pass and fail results. Below we’ll show at how to generate pulses or levels using the script tool.

Predefined Function used:

pulse(activeVal, offsetMillisec, durationMillisec) - generates a pulse output.

activeVal – 1=active-high pulse, 0=active low-pulse.

offsetMillisec – offset or delay from the moment this statement executes, in milliseconds.

durationMillisec – duration of the pulse, in milliseconds.

In the **Post Image Processing** function:

```
// Set the pass/fail condition

If (Result=1)                    // If composite Pass condition
        Global.GPO[0] = pulse(1,0,20) // generate a 20 ms pulse on output 0
Else                             // Else fail
        Global.GPO[1] = pulse(1,0,20)// generate a 20 ms pulse on output 1
Endif()                          // Close condition statement
```

Similar equation statements can be used to indicate that a specific measurement caused a failure. There are many different ways to formulate a statement. You can use the first field as part of the statement, enter a 1 (always true) or leave the field blank.

If (MS1.Result = 3) Global.GPO(1) = pulse(1,0,400)

The above equation outputs a 400 ms active high pulse on GPO(1) (no delay) if the MS1 match tool fails (Result=3). The equation below generates the same pulse if the MS1 match tool is not a pass (Result = 2 or 3)

```
If (MS1.Result != 1) Global.GPO(1) = (pulse(1,0,400)
```

MS1 is the measured value of the match. You can use the measured value in statements, in place of the result of a measurement as shown below. Output a 50 ms active high pulse on GPO(1) (5 ms delay) if the MS1 match score is less than 90

```
If (MS1 < 90) Global.GPO(1) = pulse(1,5,50)
```

Similarly, if the Distance measurement L1 is less than 400, output a 50 ms active high pulse on GPO(1) (5 ms delay).

```
If (L1 < 400) Global.GPO(1) = pulse(1,5,50)
```

Below is an example of using the “always true” condition. Set the GPO(1) to logic 1 if MS1 match tool does not pass; set GPO1 to logic 0 if MS1 match tool passes

```
If(1) Global.GPO(1) = (MS1.Result != 1)
```

Replacing “If(1)” with "If ( )" produces the same result.

Note! If you set an output to a level, it will stay that way until you change it. For example you could set an output to “1” as above and then reset it to “0” before processing the next image using the “Image Pre Process Function”.

A.2 Solution Switching

There are different scripting methods available for solution switching. We’ll describe two popular scenarios that use external inputs and Variables. In the case of inputs the scripts can be expanded with the number of inputs (i.e. using the PL-200 will give you 8 inputs to support up to 256 solutions). Both cases use a periodic function:

Predefined variable used:

Solution - Global command variable that loads a new solution when written

Add a **periodic function** (typically 100ms) to solution 1 with the following:

```
// GPI[1] - defines solution 0 (LOW) or solution 1 (HIGH)

If (GPI[1]=0)           // Check for solution #
      Solution = 0      // switch to solution 0
Endif                       // Close condition statement
```

Add a **periodic function** (typically 100ms) to solution 0 with the following:

// GPI[1] - defines solution 0 (LOW) or solution 1 (HIGH)

```
If (GPI[1]=1)           // Check for solution #
      Solution = 1      // switch to solution 1
Endif                       // Close condition statement
```

If you use a variable to define the solution # instead of an input, then the script must be modified to reflect the variable value. EagleEye supports variable access through integrated network commands (inside a 3rd party program) or through PLC control. In the latter case a PLC connection has to be established and a register or tag is assigned as the solution ID variable.

As an example, let's assume EagleEye is connected to a Rockwell ControLogix PLC with a user defined tag called "EagleEye_SOLUTION". We need to setup a periodic script to monitor the tag for a solution change.

Note! *Since this is a variable, it is always recommended to initialize it at solution load time. This will prevent immediately switching to an unexpected solution upon entry if the variable is in an unknown state.*

Init variable in the **solution initialize** function:

```
CIx10.EagleEye_SOLUTION = solution  //Add for each solution
```

Set up **periodic** function to monitor tag:

```
Solution = CIx10.EagleEye_SOLUTION// switch to solution if ID
is different
```

A.3 Trigger Control

Inspector Express supports 3 modes of triggering, internal timer, external input and software controlled. Most applications use external input or software controlled. Internal or external triggering selected using the Inspector Express GUI in the sensor setup panel. Software trigger is controlled via scripting or via network commands.

This simple example shows how to software trigger inspections using scripting.

Predefined functions:

trigger( ) - generate an image trigger signal. The Sensor Trigger must be set to "Inspection Trigger" when using this function.

TriggerSource(source) - set the trigger source or trigger mode.

source – 0=freerun, 1=internal timer, 2=external trigger, 3=software

In **Solution Initialize** function:

```
TriggerSource = 3// Set to software trigger mode on
solution load
Trigger_Enable = 1// Trigger qualifier for handshake (arm ini-
tially)
```

In user defined **Periodic Function**:

```
Trigger_Start = MBSlaveHrs16[2]// Read Modbus defined trigger
register
If(Trigger_Start=1 AND Trigger_Enable = 1)//   Check   trigger
condition
Trigger()       // Software trigger
Trigger_Enable = 0// Turn off Trigger gate
Endif
If(Trigger_Start=0)// Wait until PLC changes Trigger state
Trigger_Enable = 1// Re-arm trigger
Endif
```

Note! In general, software triggers are held off during an active acquisition cycle.

A.4 ReTrigger Example

A special case of triggering uses the “retrigger” function. This function re-triggers the last acquired image for processing, rather than triggering a new image. The function is useful in an application where you first need to identify which part is passing the camera before processing it. In this case, the external trigger captures the image for identification and the retrigger function reloads the same image after the relevant inspection solution is loaded.

Predefined functions:

ReTrigger(camID) - causes re-processing the last image on the indicated camID.
camID – always 0 for the EagleEye.

In **Solution Initialize** function:

```
ReTrigger(0)// Retrigger previous image for inspection
```

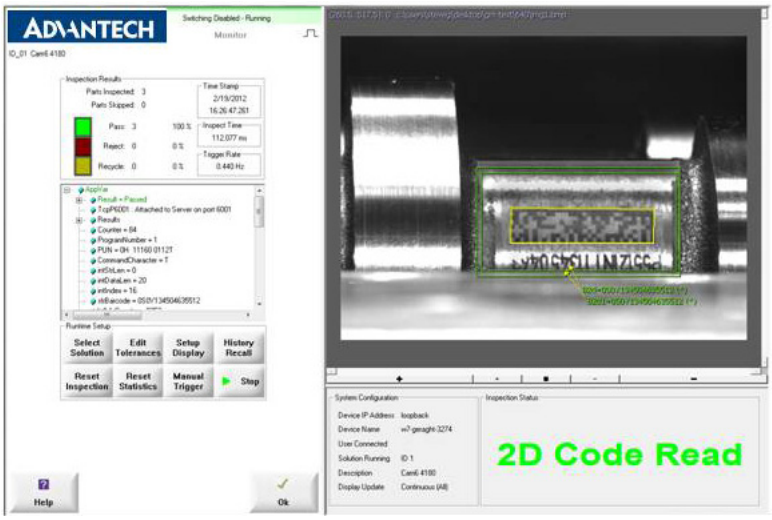
Note! If a client is connected, the GUI may not update correctly following a retrigger after load. In such cases it is good to add a small delay inside a periodic function (using the count variable defined below)

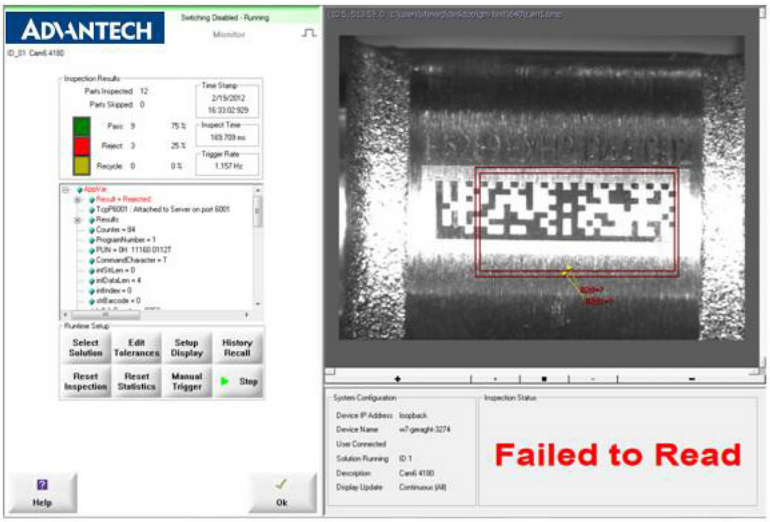
In **Solution Initialize** function:

```
NewSolution = 2// Set counter to 2
```

In 40ms user defined **Periodic Function**:

```
If(NewSolution=0)// When count = 0
ReTrigger(0)          // Force retrigger
Endif
NewSolution = NewSolution -1// Decrement count
```





The font size in the text window is not user selectable, but rather scales according to how much text is being displayed. The font type is also not selectable.

If you wish to display multiple lines of text, you need to construct strings accordingly inside a single function call. If you define multiple SetDisplayStatus() functions in your script, only the text from the last function will be displayed i.e. each will overwrite the previous.

An example of how to define multiple lines of text in the “Post Image Process” function is as follows:

```
Str1 = "Count Result = "// define count string
Str2 = "\n Inspection Passed"// define pass message
Str3 = "\n Pass Count ="// define pass count string
Str4 = "\n Inspection Failed" // define fail message
Str5 = "\n Fail Count ="// define fail count string

if(Result=1)
    SetDisplayStatus(Str1+N.Result+Str2+Str3+Global.PassCount,
"darkgreen")
Else
    SetDisplayStatus(Str1+N.Result+Str4+Str5+Global.FailCount,
"darkred")
Endif
```

A pass result would produce the following text on the monitor screen:

A.5 Customizing Text is the Monitor Pane

Inspector Express offers little in the way of runtime customization, but through scripting it is possible to display results or messages in the “Display Status” window on the Monitor panel.

Predefined functions:

SetDisplayStatus (statusMsg, color) - Sets the message to be displayed in the Inspection Status box (in the Configuration and Status panel associated with the Monitor panel). This overrides the display of "Pass" or "Fail".

statusMsg – the string that will be displayed. For multiple lines, add the character \n to indicate a new line. Message text is automatically sized to be the largest possible that can be contained in the Inspection Status box. String formatting information for variables of the form [Var%FormatData] is also supported: **msg1=”[L1%0.2f]**” means display the value of L1 with 2 digits to the right of the decimal point.

color – The string name of the color the message text is to appear in. Possible values are: "black”, “red”, “green”, “yellow”, “blue”, “magenta”, “cyan”, “white”, “darkred”, “darkgreen”, “darkyellow”, “darkblue”, “darkmagenta”, “darkcyan”, “lightgray1”, “moneygreen”, “skyblue”, “cream”, “lightgray2”, “mediumgray”.

The following script commands will display the messages ‘2D Code read” for pass inspections and “Failed to Read” for fail inspections (screen shots shown on the following page):

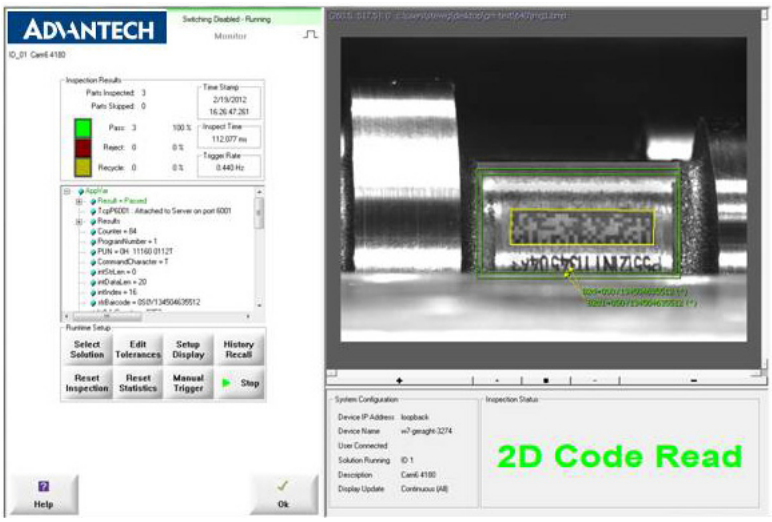
```
If(Result=1)
```

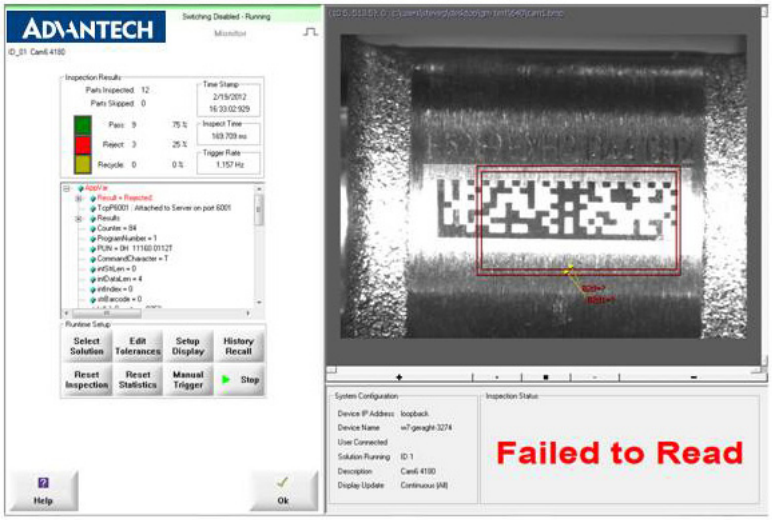
SetDisplayStatus(“2D Code Read”, “green”) // post pass message

```
Else
```

SetDisplayStatus(“Failed to Read”, ”red”) // post fail message

```
Endif
```





The font size in the text window is not user selectable, but rather scales according to how much text is being displayed. The font type is also not selectable.

If you wish to display multiple lines of text, you need to construct strings accordingly inside a single function call. If you define multiple SetDisplayStatus() functions in your script, only the text from the last function will be displayed i.e. each will overwrite the previous.

An example of how to define multiple lines of text in the “Post Image Process” function is as follows:

Str1 = “Count Result = “// define count string
Str2 = “\n Inspection Passed”// define pass message
Str3 = “\n Pass Count =“// define pass count string
Str4 = “\n Inspection Failed” // define fail message
Str5 = “\n Fail Count =“// define fail count string

```
if(Result=1)

SetDisplayStatus(Str1+N.Result+Str2+Str3+Global.PassCount
, "darkgreen")
Else

SetDisplayStatus(Str1+N.Result+Str4+Str5+Global.FailCount
, "darkred")
Endif
```

A pass result would produce the following text on the monitor screen:

A.6 Sensor Control

Scripts can be used to dynamically monitor and adjust sensor parameters to compensate for lighting variation. This is not a common concern since most applications use controlled lighting, but it can be useful in situations where process changes affect part intensity.

Predefined functions:

SetExposure(exposureTimeMilliseconds) - sets the image exposure time in milliseconds.

Example: **SetExposure(9.6)** sets the exposure time to 9.6 milliseconds.

GetExposure() - returns the current value of exposure, in milliseconds.

SetDisplayStatus(statusMsg, color)

This inspection adjusts the camera exposure (shutter) to a lighting brightness range and writes a message below the image. In this example we want to keep the area average between 198 and 202.

Script in the **Post Image Process** function:

```
If(InspectAreaAve<198 OR InspectAreaAve>202)// Check condition
    AdjustShutter()// Adjust Shutter to compensate
Else
    SetDisplayStatus("Inspecting","green")// Write to display
Endif
```

Script in the **User Function** "AdjustShutter":

```
// This script gets called when the inspection intensity
checks go above or below
// user defined limits

If(InspectAreaAve>210)    // Check upper limit
     ShutterAdjust = -0.5      // Lower shutter value
Endif
If(InspectAreaAve<=210 AND InspectAreaAve>202)// Check
mid upper limit
     ShutterAdjust= -0.1       // Lower shutter value
Endif
```

```
If(InspectAreaAve<190)    // Check low limit
     ShutterAdjust= 0.5        // increase shutter value
Endif
If(InspectAreaAve>=190 AND InspectAreaAve<198)// Check
mid lower limit
     ShutterAdjust= 0.1        // increase shutter value
Endif
CurrentShutter=GetExposure()// Get current exposure value
NewShutter=CurrentShutter+ShutterAdjust// Compensate
If(NewShutter>=0.1 AND NewShutter<=100)// Verify within
exposure limits
     SetExposure(NewShutter)// Set new exposure
     SetDisplayStatus("Adjusting Shutter","yellow")//
Inform operator
Else
     SetDisplayStatus("Shutter Out of Range, Check
Light","red") // Inform operator
Endif
Return()
```

A.7 Image Logging

There are different ways to write images. This example shows how to log images to a remote drive.
Predefined function used:

DriveConnect(Password, UserName, ServerPath) - connect to a remote server's share. Use in conjunction with **WriteImageFile** or **WriteImageTools** to connect to a remote drive that requires a user name and password login to access. NOTE:

DriveConnect should be called before each **WriteImageFile** or **WriteImageTools** call that writes to a remote drive, to ensure the remote drive stays connected.

Note! 

The Inspector Express software also supports ftp file logging, where Inspector Express is the ftp Client logging to an ftp Server.

The ftp File Name syntax is:

ftp://userName:password@host/path

The userName and password are optional in Inspector Express. These may be required by your ftp server.

```
// log pass images to network drive
If(Result = 1)
     // Connect to drive
     DriveConnect("password", "administrator",
```

```
"\\MyDrive\D")
     // Construct file name - you can construct a string
name composed of different
     // variables using the "+" operator
     fn = "\\MyDrive\D\PassImages\" + "Job_" + "123_" +
"Image_"+"Num"+".bmp"
     // Write image to selected drive with defined file
name
     WriteImageFile(fn, 0)
Endif
```

The above equations produce a file name of: Job_123_Image_Num.bmp

A.8 Result Logging

This example shows how to log results to a CSV file on the client hard drive. The "client" is a PC connected to a EagleEye camera. Note: Client and Server are one of the same for the EagleEye emulator

To demonstrate both the "logstart" and "logstop" functions, we'll design a small program that stops logging after 20 failures. To do this we'll need a count variable to keep track of failures and we'll need to initialize this variable when the solution first loads.

:

Predefined variable used:

LogStart(fileName, onClient) - Start logging the processed frame data to the specified CSV (comma separated values) file.

fileName – full path and file name to save CSV data to. *For Example*, C:\Logs\iHistlog119200614.csv

onClient – if **0**, save to file on server; if **1**, save to file on client.

LogStop( ) - Stop logging data that was started by a **logstart** call.

In the "Solution Initialize" function:

```
// Init fail_count variable
Fail_count = 0
// Start the CSV file logging function
LogStart("d:\DALSA\failog.csv,0")
```

In the "Post Image Process" function:

```
// Check fail_count value
If (fail_count = 20)
        LogStop()//stop logging
Else
        If (Result.0=3)//check for fail
              fail_count=fail_count+1  //
increment
        Endif
Endif
```

Screenshot of the CSV log file that this example generates. Note that all results are logged until the 21st failure is detected.

Frame Number	TimeStamp	Result	Avg1	failcount
1	20:12:24	Pass	148.32	0
2	20:12:24	Reject	121.66	1
3	20:12:24	Reject	164.09	2
4	20:12:24	Recycle	139.41	2
5	20:12:24	Pass	146.85	2
6	20:12:24	Pass	148.32	2
7	20:12:24	Reject	121.66	3
8	20:12:24	Reject	164.09	4
9	20:12:24	Recycle	139.41	4
10	20:12:25	Pass	146.85	4
11	20:12:25	Pass	148.32	4
12	20:12:25	Reject	121.66	5
13	20:12:25	Reject	164.09	6
14	20:12:25	Recycle	139.41	6
15	20:12:25	Pass	146.85	6
16	20:12:25	Pass	148.32	6
17	20:12:25	Reject	121.66	7
18	20:12:25	Reject	164.09	8
19	20:12:25	Recycle	139.41	8
20	20:12:26	Pass	146.85	8
21	20:12:26	Pass	148.32	8
22	20:12:26	Reject	121.66	9
23	20:12:26	Reject	164.09	10
24	20:12:26	Recycle	139.41	10
25	20:12:26	Pass	146.85	10
26	20:12:26	Pass	148.32	10
27	20:12:26	Reject	121.66	11
28	20:12:26	Reject	164.09	12
29	20:12:26	Recycle	139.41	12
30	20:12:27	Pass	146.85	12
31	20:12:27	Pass	148.32	12
32	20:12:27	Reject	121.66	13
33	20:12:27	Reject	164.09	14
34	20:12:27	Recycle	139.41	14
35	20:12:27	Pass	146.85	14
36	20:12:27	Pass	148.32	14
37	20:12:27	Reject	121.66	15
38	20:12:27	Reject	164.09	16
39	20:12:27	Recycle	139.41	16
40	20:12:28	Pass	146.85	16
41	20:12:28	Pass	148.32	16
42	20:12:28	Reject	121.66	17
43	20:12:28	Reject	164.09	18
44	20:12:28	Recycle	139.41	18
45	20:12:28	Pass	146.85	18
46	20:12:28	Pass	148.32	18
47	20:12:28	Reject	121.66	19
48	20:12:28	Reject	164.09	20
49	20:12:28	Recycle	139.41	20
50	20:12:29	Pass	146.85	20
51	20:12:29	Pass	148.32	20

Below shows how to change the “Post Image Process” script to log ONLY the failed results:

```
// Check for Fail_count limit
If (Fail_count < 20)
        If (Result.0=3)
            LogStart(“d:\DALSA\failog.csv,0”)
            fail_count=fail_count+1
        Endif
Else
        LogStop()//stop logging
Endif
```

A.9 Using Start and Stop Functions

These functions can be used to control the running state of EagleEye. Whilst in the STOP state, EagleEye is effectively in bypass mode, meaning it will not respond to triggers and it will not execute the preordered functions. However, it will continue to execute event functions if defined.

This example shows how to stop Inspector Express (and restart) if a failed condition is detected *n* number of times. In this case, the failed condition is an incorrect number of characters read by the OCR tool. The “Post Image Process” function will be used to test the OCR string and stop inspect when the count reaches 50. We’ll also check the code against a preloaded value (stored in a variable) and set an output state to indicate “good” or “bad” code. We’ll setup a periodic function to detect if Inspector Express has stopped and if so restart it. This same function will be used to reset the failure count variable.

Note! *You must save the solution before the Stopinspect () function will work*

Predefined function used:

StartInspect( ) - start image inspection.
StopInspect( ) - stop image inspection.

In the “post Image Process” function:

```
// Store the number of characters from the OCR string in
variable “slength”
// Count number of times the expected length does not match and
STOP if 50
slength=StrLen(OCR1)
If (slength<8)   //check for wrong number of characters
      fcount=fcount+1//increment bad counter
      If (fcount=50)//check for stop condition
            StopInspect()
      Endif
Endif
```

Also in the “post Image Process” function, we’ll manipulate an output based on the read result being correct or not:

```
// Check value of OCR1 against predefined code and action
output
If (codecheck=OCR1)//codecheck is a preloaded user variable
      Global.GPO[0]=1//output code good
Else
      Global.GPO[1]=1//output code bad
Endif
```

The outputs can also be pulsed as shown earlier or terminated in length using the “Image Process Delay function”. Another option would be to hold the current output state until the next inspection starts (i.e. reset them in the “Image Receive Delay

Function”).
Now we’ll define a simple periodic function to restart Inspector Express based on notification from a PLC:

```
// If Inspector Express has stopped, reset fail feature count
and restart when directed

PLC_Runmode=EIPint[0]//Ethernet/IP register 0
If (PLC_Runmode = 1)
      StartInspect()
      fcount=0
Endif
```

A.10 Communication Using Strings

This script demonstrates how to read a string from an attached Ethernet port. In this case, the string contains command characters for solution control and triggering. The script parses the string to extract the control characters for action. We’ll define the string to be 10 characters (+ delimiter character) as follows:

Char 0 = command character – “T” = Trigger, “S” = Stop, “R” = Restart
Char 1 = Solution # - 0 through 9
Char 2-9 = Job code

This script should be added to a periodic function for the following reasons:
Communication is asynchronous to processing
Periodic functions continue to run when the system is in a “stopped” state, enabling the user to issue a “restart” command.

Predefined functions used in this example:
ReadString(comVar, endingChar) - Reads a string from the TCP/IP connection specified by *comVar* if one is available, otherwise returns an empty string immediately.
Substring(string, startIndex, length) - forms a sub-string from the input *string*, beginning at *startIndex* (zero-based) of *length* characters. If *length* = 0 all characters to the end of the string are included in the sub-string.
trigger( ) - generate an image software trigger signal. The Sensor Trigger must be set to "Inspection Trigger" when using this function.
StartInspect( ) - start image inspection.
StopInspect( ) - stop image inspection. Stops the camera from processing and outputting. For example, something has gone wrong on the line and controlling equipment needs to be stopped.

The Periodic function:

```
ReadBuffer = ReadString( TcpP6001 , 13)// load string until
"CR" line delimiter detected
if(ReadBuffer != "")  // if buffer is not empty
CommandString = ReadBuffer// store string in string variable
CommandCharacter = Substring(CommandString, 0, 1) //extract
command character
SolutionNumber = Substring(CommandString, 1, 1) // extract
solution number
    JobCode = Substring(CommandString, 2, 9) // extract job
code
if(INT(SolutionNumber) >0 AND INT(SolutionNumber)<9) //
validate solution #
        SolutionNumber = INT(SolutionNumber)// convert from
string to INT
     SOLUTION = SolutionNumber// change to specified solution
    endif

//Note: if SolutionNumber is different than current solution
running, change will start //immediately and the following code
will be ignored. If they are the same, the following code //
will be executed
    if(CommandCharacter = "T") // look for trigger command
        trigger( )    // call trigger function
    endif
 if(CommandCharacter = "S") // look for trigger command
        stopinspect( )// call stop function
    endif
 if(CommandCharacter = "R") // look for trigger command
        startinspect( )// call restart function
    endif
Endif                    // close main IF statement
```

A.11 Using Arrays

The script tool supports the use of arrays (consecutive data registers). Arrays can be used to segment or organize data into a user defined structure. Some complex tools output results into arrays for script manipulation, but most users define arrays for more efficient communication.

Array Example 1:

This simple example shows how to parse an array of blob areas (Area.[x]) from the count tool to find the biggest blob (Note: in practice this can easily be done using the "max" feature of the count tool without writing any script):

In Post Image Process Function:

```
array_count=0// reset array_count variable
max_value=0    // reset max_value variable
while(array_count<20)// Do while array_count <20
 array_value = Area.[array_count]// Read entry in array
    if(array_value>max_value)// Check for high value
        max_value = array_value// Make new highest value
    endif
    array_count=array_count+1// Increment counter
endwhile
```

Array Example 2:

This similar example shows how to parse an array of numbers to find two consecutive entries that differ by more than 20%:

Input Array (entries 3 and 4 differ by 25%): ds[0:5]=90, 90, 90, 100, 80, 90

Post Image Process Function:

```
//
numHi = 0
i = 0
while( i < 6 )
    z = i+1
    while( z < 6 )
        if( ds[i] < ds[z] )
            ratio = ds[z] /  ds[i]
        else
            ratio = ds[i] /  ds[z]
        endif
        if(ratio > 1.20)
            highRatio[numHi] = ratio// store high ratio
            highRatio[numHi][0] = I// store first number
            highRatio[numHi][1] = z// store second number
            numHi = numHi+1
        endif
        z = z+1
    endwhile
    i = i+1
endwhile
```

Array Example 3:
This example shows how to match a barcode string to a predefined string stored in an array. The 32 string array is indexed based on a user code supplied through the Global Inputs (PL-200) i.e. if input = 0x4, compare the barcode value with string #4.

Place the 32 strings in array called “BarCodeList”

BarCodeList[0] = “M54321”
BarCodeList[1] = “JJ H4321”
…
BarCodeList[31] = “KLM6721”

Post Image Process Function:

```
idx = 0
i = 0
while( i < 5 )      // Find array entry to compare
    idx = idx | (Global.GPI[i] << i)// Based on 5 GP inputs
    i = i+1
endwhile
BarMatch = BarCodeList[idx]// Compare string
```

Using arrays for communication:

Arrays offer convenience and efficiency when communicating between EagleEye and 3rd party equipment. You can define a common data structure that can be moved quickly between devices using a single command.

WriteBytes(comVar, byteArray, numBytes) - Writes a byte array to the TCP/IP connection specified by *comVar*

copy(source, dest, numElements) - Copy *numElements* from *source* (an array of elements) to *dest* (an array of elements). The copy function can be used to cause multiple PLC registers to be updated in a single transaction

Example:

This example shows how to send the X, Y and Z coordinates to a robot in one Modbus multiple register transaction. With variable MB92HRs16 attached to a Modbus holding register, the following will update the Modbus device coordinates in a single transaction.

```
cmd[0] = x          // X coordinate
cmd[1] = y          // Y coordinate
cmd[2] = z          // Z coordinate
copy (cmd, MB92HRs16, 3)// Send coordinates
```

Similarly, you can store results from multiple tools in a single array and transfer them efficiently using the WriteBytes function:

```
meas[0]= L1         // Array element 0
meas[1]= L2         // Array element 1
Meas[2]= N1         // Array element 2
…
Meas[9]= IntAverag1// Array element 9
WriteBytes( TcpP5025 , meas, 10)// Send 10 elements
```

A.12 Manipulating Bits of Data

This example shows how to use the Bit Function and TCP/IP communication. A byte number is sent to EagleEye which has to be converted to a binary number. The binary number defines which inspection results are to be sent over TCP/IP for display.

Predefined functions Used:

GetBit(value, bitPosition) - tests and returns the state of the bit at *bitPosition* in *value*. Returns the bit state.

value – a valid number (or variable) to be manipulated on a bit level.

bitPosition – location of the bit to be tested.

Script in the **Pre-Image Process Function:**

```
// Initialize data bit variable
Bit0 = 0
Bit1 = 0
Bit2 = 0
Bit3 = 0
```

Script in **Post Image Process Function**:

```
MyData = ReadByte(TcpP5025)// Read byte from TCP port
If(MyData != “”)
Bit0= GetBit(MyData,0)// get bits from MyData byte
Bit1= GetBit(MyData,1)
Bit2= GetBit(MyData,2)
Bit3= GetBit(MyData,3)
SetDisplayStatus(“Data Received”, “green”)// send message to monitor
SendCountData()    // call User function
Else
SetDisplayStatus(“No Data”, “red”)// send message to monitor
Endif
```

Script in **User Function SendCountData**// User defined function

```
WriteFormatString(TcpP5025,”\n\r New Count Data”)// Send start of string
If(Bit0 = 1)
WriteFormatString(TcpP5025,”\n\r    Left    Side    Found    = [Left_Side]”) // send left count
Endif
If(Bit1 = 1)
WriteFormatString(TcpP5025,”\n\r    Right    Side    Found    = [Right_Side]”) // send right count
Endif
If(Bit2 = 1)
WriteFormatString(TcpP5025,”\n\r Top Side Found = [Top_Side]”) // send top count
Endif
If(Bit3 = 1)
WriteFormatString(TcpP5025,”\n\r    Bottom    Side    Found    = [Bot_Side]”) // send bot count
Endif
Return()// Return to main program
```

Example of String produced when Bit0=1:

New Count Data
Left Side Found = 38

ADVANTECH

Enabling an Intelligent Planet

www.advantech.com