DVT Script Reference Manual

Part Ordering Number: DOC-SCR

Documentation Control Number: MAN-102 REV A

Software Version: FrameWork 2.6

Welcome to the large family of DVT product users! We are very pleased that you purchased our products and look forward to helping you build a Smart Factory. We at DVT have made a few improvements and changes to the content of this manual to better assist you in your endeavors. We hope that you find this version of the manual both helpful and accommodating. If you have any questions, suggestions, or comments in regards to any of our user manuals, please help us to better help you by contacting our Technical Editor at docs@dvtsensors.com.

Table of Contents

Chapter 1 – Introduction to DVT Scripts

Even though SmartImage Sensors contain a number of SoftSensors to perform specific tasks, some applications require more customization. Sometimes users need to make intelligent decisions based on bits and pieces of data from different SoftSensors. Sometimes a very specific handshake must be implemented to successfully communicate with an external device. Sometimes users prefer to write their own code and use the standard SoftSensors as data gathering tools. For all these cases and for even more specific cases DVT has created the Script tools, a set of programmable tools. This chapter explains the basic ideas behind DVT Scripts. The most common uses are explained. No reference to syntax is made, but some practical examples about when to use scripts are given. The types of scripts available in SmartImage Sensors are introduced as well. The chapter also demonstrates how to take the first steps to create both Background and Foreground Scripts. The reader is shown how to bring up both editors and how to interpret the different options available.

Why Scripts?

As a starting point to explain the functionality of SmartImage Sensors, the hierarchical organization within SmartImage Sensor must be explained. Figure 1 illustrates the organization inside SmartImage Sensors. There are three well defined levels within SmartImage Sensors: system level, product level and SoftSensor level.

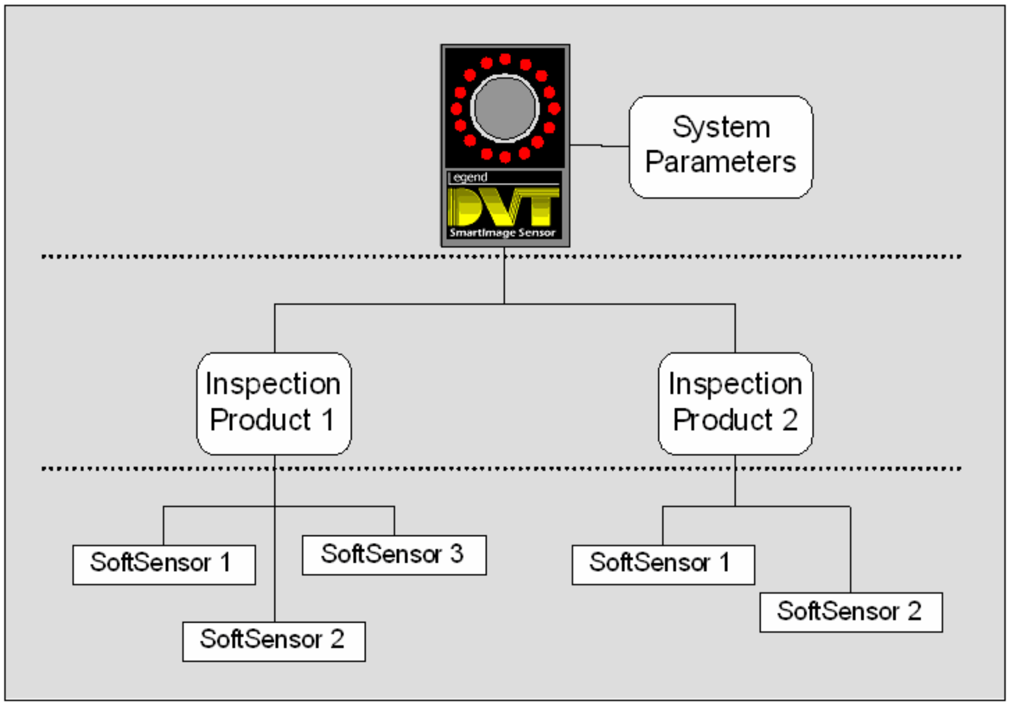


Figure 1: Hierarchical organization within SmartImage Sensors.

These three levels are very independent. The System level contains parameters that affect the functionality of the SmartImage Sensor itself. Parameters such as communication settings, and trigger mode (internal/external) are examples of system level parameters. These parameters do not change between inspections, they define global settings. At the second level, we have the inspection products which are specific for every inspection. Usually SoftSensors perform a variety of inspections from the same or from different parts. An inspection product is needed for every inspection performed. Finally, at the lower level, we have the SoftSensors. SoftSensors perform inspection tasks. Each inspection, as defined by the inspection product, needs to perform a number of tasks in order to fully execute. These tasks are assigned to the SoftSensors.

In order to illustrate this functionality let us work with an example. The part in Figure 2 needs to be inspected. First, we need to verify that the drillings on the left end are vertically centered, a certain distance apart, and of a certain radius. After that, we need to verify that the label on the right end is centered, aligned and that it contains the correct code. The existence of products inside SmartImage Sensors simplifies this task. The inspections can be separated into two groups, those that take place on the left end of the part and those that take place on the right end of the part. This will allow the user to trigger the SmartImage Sensor twice as the part moves in a conveyor from left to right. The first trigger would occur when the left end is in front of the SmartImage Sensor and the second one when the right end is in front of the SmartImage Sensor.

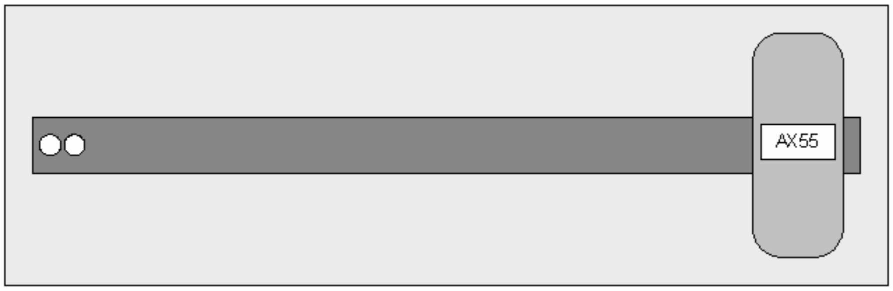


Figure 2: Sample part to be analyzed in two different inspections.

This would not only add modularity to the process but also would allow for a more precise inspection because now we can zoom in on the part. There is no need to have the entire part in a single image. The inspections are then separated into products. A product would inspect the left end of the part and another product would inspect the right end. In order to determine if there is a problem with the part, those products would use a number of SoftSensors. For the first product, both Math and Measurement SoftSensors are needed to perform the inspection. For the second product the same SoftSensors are needed and also a reader SoftSensor (OCR SoftSensor) is required.

This is a very effective internal structure that enables users to encapsulate tasks into specific entities that exist inside SmartImage Sensors and design/troubleshoot each one independently from the others. The only downside of this structure is the independence that every entity has with respect to the others. Inspection products can only control their own SoftSensors. SoftSensors can only reference other SoftSensors for a position reference or other minor tasks. Different inspections cannot communicate with each other, that is, a certain inspection is performed and when it ends all the data used by that inspection disappears from the system. For many applications, this strict independence between entities limits the power of SmartImage Sensors. For those cases, DVT has created a very powerful tool: DVT Scripts. Scripts can be viewed as the exception to the rule. Scripts can access things that other SoftSensor cannot and they can even make communications between entities possible by sharing the same system memory.

Scripts come in two flavors: background scripts and foreground scripts. The basic difference between them is that the first one is independent of inspections and runs at the system level whereas the latter runs on every inspection and exists at the product level. More details about both types of scripts will be given later in this chapter. Scripts have the ability to write values to memory and read values from memory, and since the memory is shared between all scripts in the system, they create the connection between different entities inside the system. Figure 3 shows the functionality added to the system when Scripts are used. Each type of script has a limited scope, but they can share data based on the use of the system's memory (the DVT registers, which will be explained later).

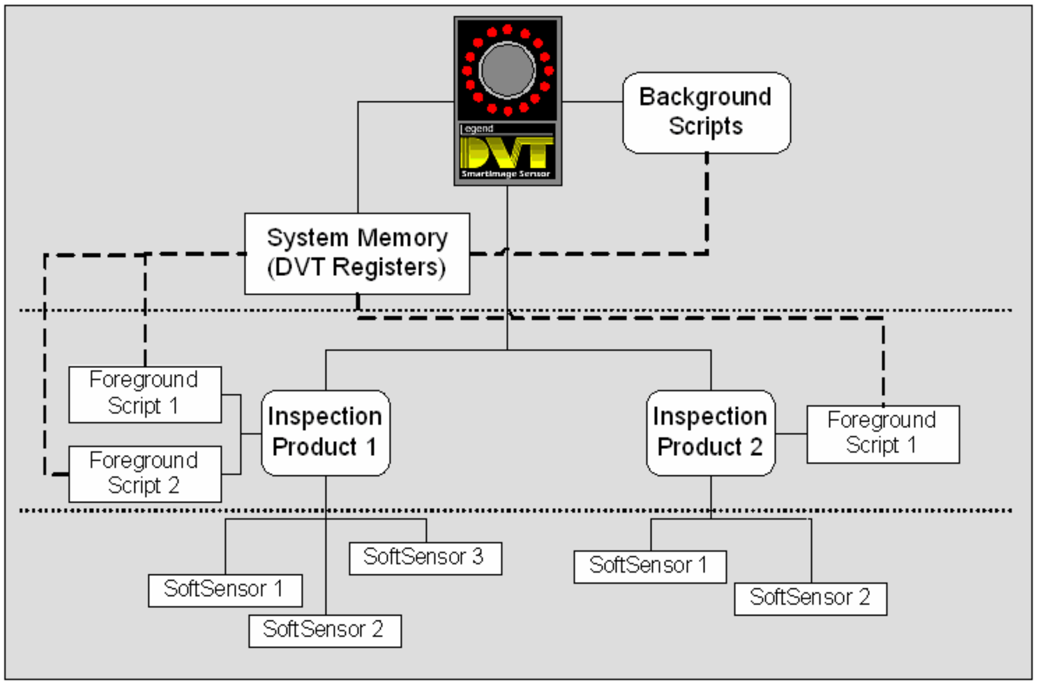


Figure 3: Hierarchical tree with Scripts added to the system.

A closer look at Scripts

So now we know the power that can be obtained from Script tools. Even though many application could be effectively configured with no scripts at all, in many cases there is going to be a need for extra functionality. For those cases the answer is Scripts.

What are scripts?

Scripts are basically a programmable tool used in SmartImage Sensors. Each script is designated as a class that can contain a number of static user-defined functions, with one required method that will be the first to execute when the script is initialized (the name of this method is different between a Foreground Script and a Background Script and will be detailed below). They are designed to be fully customizable to the application's needs. Unlike other parameters within FrameWork, Scripts have no predefined purpose. They are created as an empty tool that is shaped to perform the required tasks according to the user needs.

What exactly can Scripts do? Scripts can perform a number of tasks including but not limited to: accessing data gathered by SoftSensors, accessing and modifying product and SoftSensor parameters, establishing communications with external devices, preprocessing images before SoftSensors analyze them, performing mathematical calculations, and more. There is a predefined set of functions to perform each one of the tasks mentioned above and more, the user only needs to call the specific functions every time using syntax that will be explained in Chapter 3 - Basic Script Syntax.

Types of Scripts

As Figure 3 shows, there are two different types of scripts to fulfill the user needs at different levels within the hierarchical tree inside SmartImage Sensors: Background Scripts and Foreground Scripts.

Background Scripts

Background Scripts are created at the system level. When a Background Script first initializes, a method within the class called `main()` will be the first to execute. Within this method is where most of your script logic will reside. User-defined functions can be added to the class, however, they must precede the `main()` function. Background Scripts are not associated with any Inspection Product, so they are independent of inspections. Also, background Scripts are not automatically executed every time an inspection takes place. There are three ways to start a Background Script:

Run on power-up: this will start the Background Script when the system is powered up. This option is usually selected when a Background Script needs to configure memory registers, restore data to RAM memory, or continuously run on the background.

Start manually: not as common, it allows users to connect to the SmartImage Sensor and start a Background Script manually. This option is used mostly for testing purposes.

Start on a signal: this is an advanced option of Background Scripts. Since they can be set to wait for a specific input to change, they could be started on power-up and set to wait for a specific input from an external device or from another script. In this case the Script is actually started on power-up or manually, but it pauses execution until a signal is received.

Since Background Scripts are created at the system level, they have access to system and product parameters. Common tasks for Background Scripts are to alter product parameters, trigger inspections, establish communications with external devices, preprocess images, etc.

An important consideration regarding the nature of Background Scripts is the duration of their process: once started there are two options for Background Scripts: single shot scripts and continuous execution scripts. Single shot scripts are executed once, and need to be restarted to execute again. This mode is mostly used for testing or for cases where a certain operation needs to be performed on power-up. The second mode, continuous execution, loops through the code executing it continuously. It can be compared to the periodic scan of the rungs by PLC in a ladder logic program.

Foreground Scripts

Foreground Scripts are created at the product level, and they are directly related to specific inspections. When a Foreground Script is executed, a method called `inspect()` will be called. Within this method is where most of your script logic will reside. User-defined functions can be added to the class, however, they must precede the `inspect()` function. Foreground Scripts run every time the Inspection Product that contains them is called to inspect an image. Foreground Scripts are created just like any other SoftSensor, which is why they are also called Script SoftSensors. The main difference that they have with other SoftSensors is that they can access data from the SoftSensors in the same Inspection Product. Common tasks for Foreground Scripts include gathering data from other SoftSensors, performing mathematical or logical computations, sharing information with other areas via system registers, and formatting strings to send out of the system via DataLink. Like any other SoftSensor, Foreground Scripts can be set to PASS, WARN, or FAIL. They can also be used to output data to the result table, monitor I/O lines, and write to and read from the system memory.

Note: Background and Foreground Scripts are classes which can only contain static methods, so you cannot call methods of other classes.

Creating Scripts

Now that we know what Scripts are and what they are used for, we are ready to access the Script Editor. Both types of script use the same syntax, and to make it even simpler, they use the same editor. The difference in the creation of a Script resides on how the editor was opened. To create a background script, the user has to select “Background Scripts” from the main “Edit” menu. This will bring up a dialog box as shown in Figure 4.

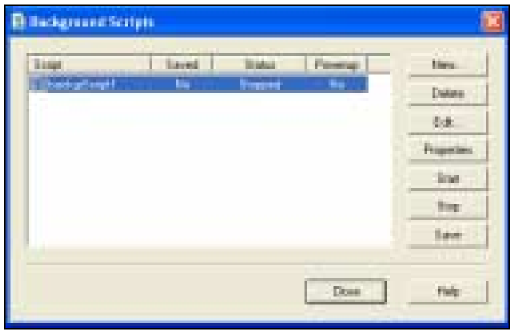


Figure 4: Background Script menu highlighting a script.

This dialog box shows all the Background Scripts present in the system. Furthermore, it indicates if they are saved to flash memory, if they are currently running, and if they start automatically on power up. When a background script is created, it resides in RAM memory just like products and SoftSensors. When the user selects to save them, they move to permanent flash memory where they reside even after cycling the power in the SmartImage Sensor. To add a Background Script to the system the user only needs to click on the “New” button. This will bring up a new dialog box as shown in Figure 5.

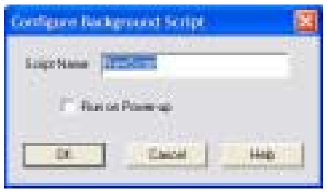
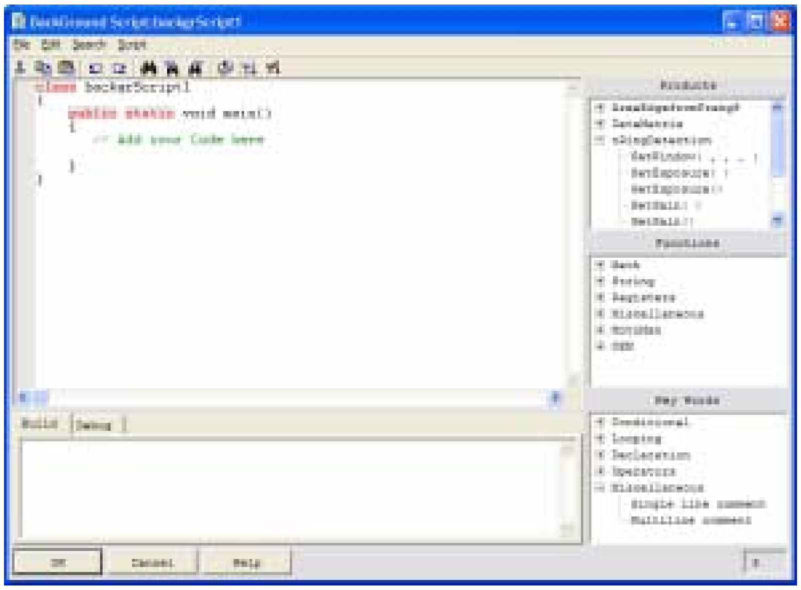


Figure 5: Dialog box to name the newly created Background Script.

This dialog box prompts the user for the name of the Background Script and whether it will be a script that runs automatically on power-up. When naming scripts, be sure to follow the same rules when naming identifiers in most programming languages (i.e. they can contain letters, numbers, and the underscore character (_); they cannot start with a number, no white spaces or other special characters or punctuation marks, and reserved keywords cannot be used). Once the Background Script is successfully created, it can be edited. By simply clicking on the “Edit” button, the user will obtain access to the Script Editor. The Script editor is where both Background and Foreground Scripts are created and edited.

The editor simplifies the way code is written by having all the functions and variables available for point-and-click selection. A screen capture of the editor for Background Scripts is shown in Figure 6. It contains five different panes. The largest pane is used for writing the script. Here is where the Background Script will be. The three panes on the right hand side contain data, declarations, and structures that Background Scripts have access to.

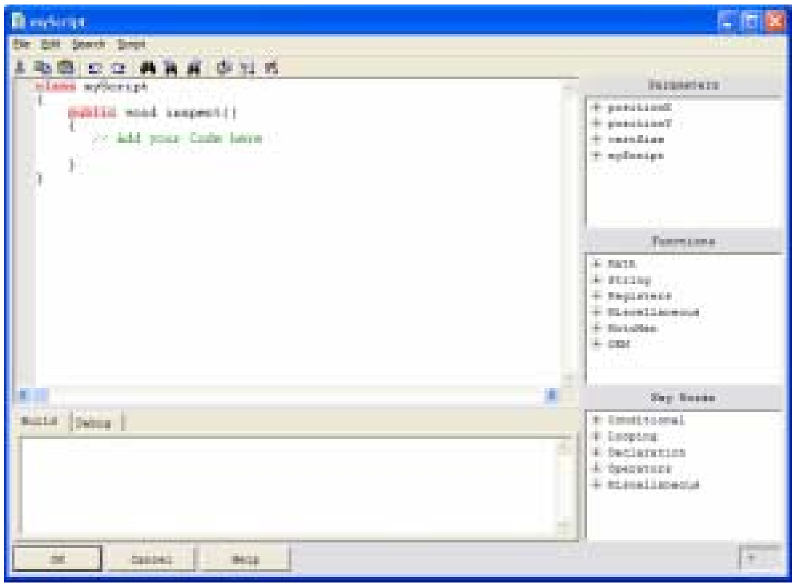


```
class backgrScript1
{
    public static void main()
    {
        // Add your Code here

    }
}
```

Figure 6: Screen capture of the Background Script editor.

The top pane on the right hand side contains the products created in the SmartImage Sensor. Since Background Scripts have access to product parameters, this pane makes available all the products in the system. The figure shows only two products and all the accessible parameters from the second product (called "oRingDetection"). If the user needs to access any of these parameters, a simple double click on the parameter will transfer it to the editing pane. The second pane on the right hand side shows the functions available for Background Scripts. They are divided in categories that can be expanded by clicking on the plus sign. The lower pane on the right hand side contains more general tools such as variable declarations, syntax for conditional and looping structures, operators, and even comments. Notice that neither of the panes contains any SoftSensor data. Background Scripts cannot "see" SoftSensors. The lower pane on the left hand side contains two tabs. The "Build" tab is where the messages from the compiler are shown. The "Debug" tab, used in conjunction with the `DebugPrint()` command is reserved for outputting special strings designed to help with the editing of a script. Finally, the number on the lower right corner of the editor indicates the line in which the cursor is located. This number is used to track problems in the code. When a script is written it needs to be compiled before being executed. This is an automated process of verification that there are no errors in the code. It is performed by clicking in the "compile" button available from the top menu ( ). If the compiler finds any errors it will report the error and the line where the error was found. If there are no errors the compiler indicates so.

The creation of a Foreground Script is slightly different. Foreground Scripts are available from the main SoftSensor toolbar, just like any other SoftSensor. When the user selects to create a Foreground Script, the parameters for it become available, the user only needs to give it a name, apply the changes to the name and select "Edit" to access the Script Editor. In this case, the editor will look slightly different. It will contain functions that are only available to Foreground Scripts and unlike Background Scripts, it will contain all the SoftSensors in the upper right pane. Figure 7 shows a screen capture of such editor. In this case, the script belongs to a product that contains three other SoftSensors (positionX, positionY, and vertSize). These SoftSensors are available from the Parameters pane.



```
class myScript
{
    public void inspect()
    {
        // Add your Code here

    }
}
```

Figure 7: **Screen capture of the Foreground Script editor.**

The fourth SoftSensor shown in that pane is the Script itself. This is why it is very important to select Apply immediately after naming the script and before opening the editor. If the user does not apply the change to the name, the script will have a default name in the editor and the reference to it will not be successfully established once the change in the name takes place.

Timing

An important consideration regarding Foreground Scripts and Background Scripts is the timing for both. Foreground Scripts are executed during every inspection, so if they need to make an output available, change a certain value, etc. the user knows that those changes will have taken place by the time that the inspection finishes. This ensures that every time there is a new value being computed, it will be made available. This is not the case with Background Scripts. Background Scripts run independent of inspection, so for example, in cases where they need to transfer data from inspections, they will not be very reliable unless careful consideration is given. They will likely execute continuously transferring the data that is stored in memory, whether it is fresh data or data from the previous inspection. In those cases, the device reading the outputs should test whether the data that arrived is new or old.

Another important consideration regarding timing is the priority of tasks within SmartImage Sensors. SmartImage Sensors have inspections as their main task. This ensures that when the SmartImage Sensor is triggered, there is no delay, the image acquisition starts immediately and right after that the inspection occurs. In most cases, other processes will be running at the same time. When the inspection begins, the other processes (Background Scripts, communications, etc.) are given a very small slice of processing time compared to what inspections get.

The user should be familiar with these concepts in order to avoid timing issues that usually become difficult to troubleshoot.

Designing a Script

As mentioned before, scripts are fully configurable tools: tools that are empty upon creation and are modeled by the user to better suit the application's needs. This process uses a programming language to describe the functionality that Scripts must have. That programming language is called DVT Scripting and is unique to DVT SmartImage Sensors, but to make it easier to understand, it follows the syntax of very common programming languages. The method to use the programming language consists of designing an algorithm and implementing it using the features of the DVT Scripting language.

Designing an algorithm

An algorithm can be defined in general terms as a precise set of instructions that describes a certain behavior. Every time we explain someone how to do something, we are describing the steps of an algorithm. The algorithm can be viewed as the crucial component in a process that produces a certain output based on certain input. Figure 8 shows how the algorithm converts an input (or a set of inputs) into an output (or a set of outputs).

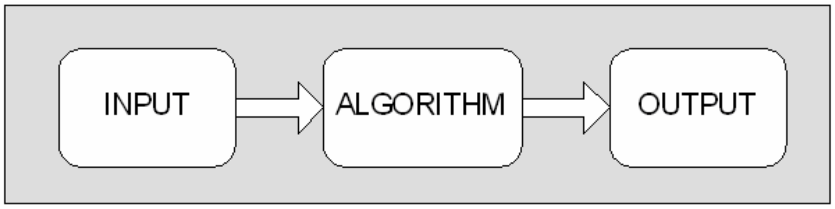


Figure 8: Algorithm as a process.

The goal is to create whatever is inside that box labeled Algorithm. Let us work with a real example to better illustrate the process. The task is to have the SmartImage Sensor triggering three times (with a certain period) when it receives the trigger signal. After all three inspections are taken, compare the position and size of three parts, if the average of those values varies more than 10% with respect to a certain specification, discard the part and increment a counter, otherwise accept the part. The first attempt to design the proper algorithm is shown below:

```
1) Wait for trigger
2) Perform 3 inspections
3) Determine if it is a good part or not (fail if necessary)
```

As a first attempt to design the algorithm, the steps are logically correct, but more specific steps could be added. For example, we could add the data gathering process and the data manipulation to the second step. We know that when each inspection finishes the variables associated with it disappear, so we must gather the data and save it somewhere safe so the last inspection can access it. The only place where we can save data to and access it later is the DVT Registers.

```
1) Wait for trigger
2) Perform 3 times:
      2.0) Inspect Part
      2.1) Save results to memory (registers)
3) Determine if it is a good part or not (fail if necessary)
```

We could also be more specific as to how we plan to determine if it is a good part or not:

```
1) Wait for trigger
2) Perform 3 times:
      2.0) Inspect Part
      2.1) Save results to memory (registers)
3) Access the data saved in memory (3 sets of data)
      3.0) Perform the mathematical computations
      3.1) Compare results with optimal values
      3.2) Determine if this consists of a good part or not
4) Check the results
      4.0) For a good part do nothing
      4.1) For a bad part fail the inspection and update a
counter
```

So the algorithm keeps taking shape. The more specific steps we add to it at design time, the less trouble we are going to have when we actually write the code. We could go a bit further on the design of the algorithm by adding some more specific steps and rewriting part of it to simplify the process of writing it. We could even use flags to avoid the execution of unnecessary steps.

```
1) Wait for trigger and set success flag to true
2) Perform 3 times if success flag is set to true:
      2.0) Attempt to inspect part
         2.1.0) If the part is not present reset the success flag
to false
         2.1.1) If the part is present, inspect and save results
      2.1) Wait for a fixed number of milliseconds
3) Check the success flag and proceed only if it is set to true
      3.0) Access the data saved in memory (3 sets of data)
      3.1) Calculate averages of size and position
      3.2) Compare results with optimal values
      3.3) Determine if this consists of a good part or not
      3.4) For a good part do nothing
      3.5) For a bad part set the success flag to false to fail
the inspection and update a counter
      3.6) Reset the registers used for storage of data to 0
4) If the success flag is set to false indicate failure and
reject part
```

So we have created an algorithm that we believe will perform the task we need when it is translated into a DVT script. Users should always attempt to attack the problems in this manner. Pencil and paper is not an obsolete method when working with algorithm design. It is much easier to make changes to a “recipe” written in English (or the language of your choice) than to attempt to change a block of code.

Note: when the script is written, the algorithm steps should be left as comments in the code to help others understand the process.

Algorithm Components

So far we have discussed how to design the algorithm (or create the recipe) for our script. We have written something that we can easily understand and edit if necessary. The next step is to translate that recipe to terms that SmartImage Sensors can understand. First we have to associate the common components that we used in our algorithm to what the Script language has to offer. The common algorithm components are: data structures, data manipulation structures, conditional expressions, user-defined functions, and other control structures.

Data Structures

Data structures are representations of information used by an algorithm. This includes the input set, the output set, and the internal set which is created by the algorithm to perform the necessary computations. Data structures can be interpreted as containers where the data values are stored. These containers are identified by a name and are variable, that is, the value stored in them can be changed. There are different types of containers for different types of data. We would not want to use the same type of container to store both the string “This SoftSensor failed” and a certain distance. We are most likely to set two different types of containers, one with a higher capacity perhaps named “stringResult” to store the string. The second one would be a smaller container perhaps named “totalDistance” to store our numerical value. It is important to have a good naming convention that describes the contents of the container.

Data Manipulation Instructions

Data manipulation instructions help the algorithm perform the following tasks:

Access the input data

Manipulate the data

Save data to memory and read it back from it

Output the results

For all four steps of the process the algorithm must have the functionality to manipulate data as needed. This is given by the data manipulation instructions. Scripts have many types of data manipulation instructions, users can select the ones that better fit the application needs.

Conditional Expressions

A very important characteristic of DVT Scripts (and computers in general) is the ability to determine a course of action based on a particular test. This is called a conditional expression. Our Script should be able to compare two different containers to check whether the data in those containers follow a certain rule. Based on the outcome of that comparison, the Script should take appropriate action. This comparison represented a conditional expression. In the algorithm we created we have a number of conditional expressions. For example, the way we changed the course of action based on the value of a flag. Every time we used a conditional expression to determine the course of action, we set up a decision point in the program execution as shown in Figure 9.

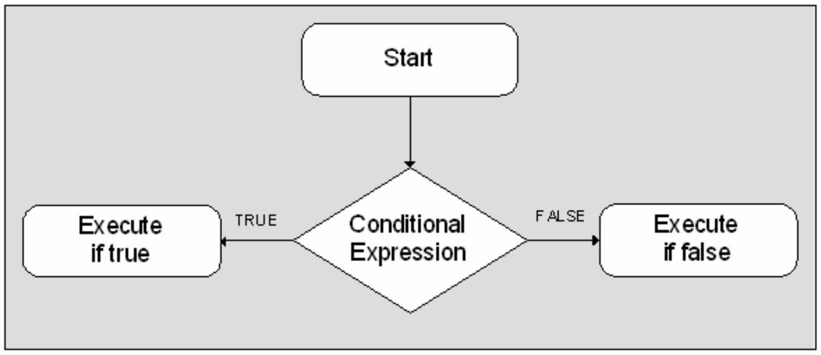


Figure 9: Basic conditional statement altering the execution of the program.

User-defined Functions

In most cases, you can think of your algorithm as being one large task that can be broken down into many smaller tasks. Sometimes you may actually have to repeat some of those tasks multiple times in different areas of your Script. Instead of repeating code in your Script, you can create a user-defined function to accomplish the task and then simply call it whenever you need it. Writing Scripts with user-defined functions in mind will result in code that is cleaner, easier to follow, and more efficient. Figure 10 gives an overview of how you can use user-defined functions in your Script.

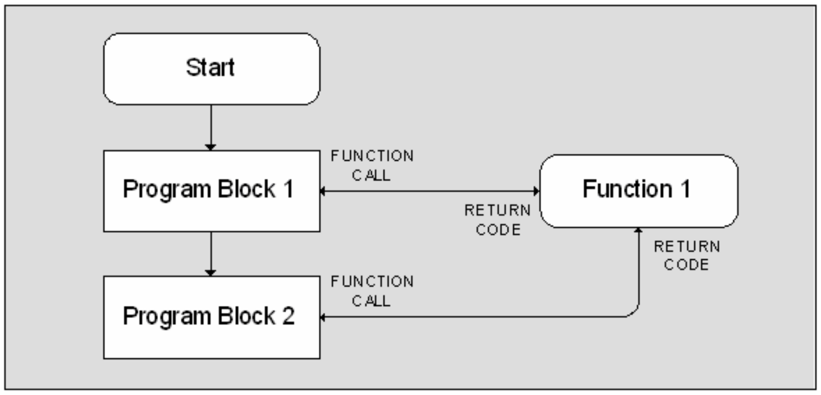


Figure 10: Example of using user-defined functions to simplify execution.

In the above example, Program Block 1 executes a function call to Function 1. In this scenario, Function 1 could be set up to accept parameters and return information back to the calling Program Block 1. User-defined functions are discussed in further detail in Chapter 3.

Control Structures

Control structures are a slightly more complicated version of the conditional statements. Using conditional statements, we checked for a condition once and based on that we executed the appropriate part of the algorithm. A control structure lets us execute part of the algorithm a number of times. In the algorithm we developed we said that part of the code needs to be executed three times. In order to implement that part of the algorithm we need a control structure checking the number of times that the block is executed. This type of control structure is shown in Figure 11. In this example we set up a counter to indicate how many times we inspect and we check the counter every time we execute the code.

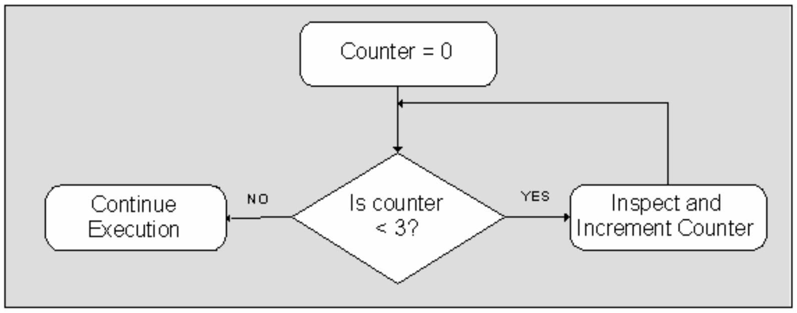


Figure 11: Control Structure for repetition.

Observe that every time the inspection is performed, the counter is incremented. When the counter reaches a value of 3, the test “Is counter < 3?” will evaluate to false and the inspection will not be performed again.

This set of algorithm components would be necessary to write our algorithm as implemented. The challenge now becomes the translation of the algorithm written in a language that humans can read to a language that a machine can read using the control structures mentioned above.

Chapter 2 - Data Types and Manipulation

So far we have discussed what we need to implement an algorithm. Now, we will discuss what is available from the DVT Script language to implement it. This chapter introduces a number of basic commands and formats of scripts that allow users to write the algorithm in a way that the processor aboard the SmartImage Sensor can understand.

Basic Data and Manipulation

The very first set of tools that we need to understand is the types of data we can use in Scripts. The basic data types allow the user to store, access numbers, letters, and Boolean expressions. The basic data types are summarized in Table 1.

Basic data types for Scripts			
Type	**Name**	**Description**	**Approximate Range**
Integer	byte	8-bit unsigned	0 to 255
Integer	short	16-bit signed	-32E03 to 32E03
Integer	int	32-bit signed	-2E09 to 2E09
Integer	long	64-bit signed	-9E18 to 9E18
Floating point	float	32-bit signed	1E-45 to 3E38 *
Floating point	double	64-bit signed	5E-324 to 2E308 *
Character	char	8-bit signed	-128 to 127
Boolean	boolean	1-bit	true - false

Table 1: Basic Data types available for Scripts. Note: the ranges marked with an asterisk (*) indicate that the range is given in absolute value (it is valid for both positive and negative values).

There are four basic groups of data: integers, floating point numbers, characters, and booleans.

Integers are used to represent quantities with no decimal places. Counters, number of blobs found, and number of milliseconds to wait, are examples where the use of integers would be appropriate. The user should also select the type of integer that better suits the application's needs and requires less memory. For a simple counter or to represent the number of blobs found, the `byte` data type might be appropriate unless the value exceeds 255. To express a number of milliseconds the user might need to use the `short` or even `int` data types depending on the application because the `byte` data type would allow for a maximum of 255 milliseconds (about 1/4 of a second). The use of the `long` data type to represent quantities is very rare. It can go up to really large values. It can store a number equal to the number of microseconds in 250 thousand years. Remember that there are one million microseconds in one second. A common use for this data type is bit manipulation of 64-bit words.

Floating point data types are used to store data that requires a number of decimal places of precision. A distance, a radius, and the constant Pi, are all examples of quantities that would require the use of floating point data types. In most cases the `float` data type will be enough given its range of values.

The `char` data type is used to store any type of character that can be represented by an integer between -128 and 127. This data type is used for tasks such as communications where the algorithm needs to verify that the other device responds with the correct set of characters. For a table with the standard ASCII characters refer to Appendix A – ASCII Table of Characters Any character in those lists can be stored in a `char` variable by assigning the correct decimal value to it.

Finally, Booleans are used for indicators. The algorithm that we designed before used some flags. There are two ways to work with flags. Assign the flags to a certain bit so it can take values of zero or one, or use `boolean` variables to determine states (true or false). The choice depends on the user, it is usually easier to work with `boolean` flags because the user has no direct access to the bit level of the memory registers.

Now that we know what the basic data types are, we are ready to create some “containers” and populate them with data. This process consists of two steps: the declaration of the variables and their initialization. The declaration of a variable consists of the indication of the data type and the name we assign to it. The user has no need to type the data type, the script editor can set that up for the user. In order to take advantage of this feature, the user must select declarations (from the Key Words pane of the editor) and double click on the variable type needed. This will bring the format of the declaration to the main pane. For instance, if we wanted to declare a variable (create a container) for a counter that needs to go from zero to 1000, we need to declare a `short` data type of variable. By double clicking on the appropriate choice we would get the following line inserted in the main editing pane of the editor:

```
short !!variable!!;
```

Notice that the declaration includes the data type (`short`) and indicates that we must name the variable. All we need to do is replace the word variable and the marks around it with the actual name. We must keep the semicolon at the end. That is a requirement of the syntax. It tells the compiler that the end of the statement has been reached. Using this procedure we can declare different variables as follows:

```
byte counter, blobCount;
int area1, area2, totalArea;
float distance1, distance2;
char terminationCharacter;
boolean success;
```

Notice how the names describe what the variables are used for. Also notice that the names contain only letters and numbers, and they do not start with a number. Those are syntax rules to declare basic data types. Also, notice how by convention we start the names of basic variables with lowercase letters. Finally, notice that more than one variable can be declared in the same line as long as it refers to variables of the same data types.

The second step of the process is the initialization of the variable. So far we have created the container but it is empty. We need to put some data in it. That process is called initialization and is where the container gets filled. To populate the containers we created with data, we will use a special operator: the assignment operator (=). This operator takes the value on the right and assigns it to the variable on the left. To initialize the variables we created before, we will need the following lines of code:

```
counter = 0;
blobCount = 0;
area1 = 3528;
area2 = 7582;
totalArea = area1 + area2;
distance1 = 125.365;
distance2 = 2 * distance1;
terminationCharacter = 122;
success = true;
```

Notice how an expression can be used on the right hand side of the assignment operator. For the variable that contains the total area, we are simply adding the variables `area1` and `area2.` To calculate `distance2` we simply multiply the variable `distance1` by 2 using the multiplication operator (an asterisk). Finally, the initialization of the `char` variable included a numerical value. If we look at a table of ASCII characters, the number 122 corresponds to the letter z. If we wanted to initialize the character directly to the letter value, we need to include single quotation marks. The following two lines of code perform the same task:

```
terminationCharacter = 122;
terminationCharacter = ‘z’;
```

It is also possible to declare and initialize a variable in the same line as shown below:

```
//declare and initialize variables in the same line
//of code to minimize the number of statements.
int maxTime = 255;
float dist = 0;
```

Arrays

Arrays consist of collections of data of the same type. These collections have a fixed number of cells and the user can reference any cell in particular at anytime. They are used in cases where we need to cycle through an unknown number of items. Let us say that we need to measure the area of all the blobs that a blob generator reports. It could be ten blobs in the current image but only 5 in the next one, and so on. In these cases we set the size of the array at the beginning of the code and we make it equal to the number of blobs found. This way we change the size of the array from inspection to inspection to perform the desired task. Sample code for this situation is provided next. At this time we should assume that we have a variable called `numberOfBlobs` that contains the data. Later we will see how to get that number.

```
int IntMyArray[];
IntMyArray = new int[numberOfBlobs];
```

Notice the presence of the brackets to indicate that it is an array and not a basic data type. The declaration includes the key word `new` to indicate that it is an array. The variable `numberOfBlobs` has to be an integer or the declaration will not be valid. This number defines the number of cells that the array is to have. After the execution of these lines of code, the array will contain a number of empty cells. The user can refer to those cells by using an index, with the starting index for arrays being zero (0). For example if `numberOfBlobs` was equal to 5, the array would end up having five cells which could be individually addressed as follows: `IntMyArray[0]`, `IntMyArray[1]`, `IntMyArray[2], IntMyArray[3]`, and `IntMyArray[4]`. The initialization of this type of data structure requires more than a single line, in this case it would require 5 lines of code as follows:

```
IntMyArray[0] = 25;
IntMyArray[1] = 35;
IntMyArray[2] = 45;
IntMyArray[3] = 55;
IntMyArray[4] = 65;
```

Usually, this process is done inside a loop. By writing a looping structure we can achieve the same results without writing many lines of code. Imagine that the array had 250 cells. It would be very redundant to use 250 statements to initialize it. So the preferred option is to loop. Loops will be explained later, but the code below illustrates their use without using specific syntax:

```
Repeat 250 times
{
      myArray[x] = x;
}
```

This will initialize the value in every cell of the array to the value of the cell index.

Another feature of arrays is the length field. When users need to cycle through the cells of an array but they are not sure about the size of the array, they can access it with a single call to this field using the syntax below:

```
int myArray[] = new int[55]; //declare an array of 55 cells
int arraySize = myArray.length; //get the size of the array
                              //into a variable
```

Note: Arrays could be set up to start at base 1 by including the following line before any arrays are declared:

```
ArrayBaseIndexOne = true;
```

Strings

Strings are sequences of characters. A string can be viewed as a collection of characters that does not need to have a predefined size. Unlike arrays, which do have a predefined size, strings require a termination character, which is given by the null character. Strings are a very special type of data for which the declaration of a `String` has two minor differences with respect to that of basic data types. The data type (String) starts with uppercase letters. To initialize the variable we can simply assign a sequence of characters inside double quotation marks as follows:

```
String errorMessage;
errorMessage = “Invalid input. Try Again.”;
```

A major difference between strings and arrays is that arrays start at cell number 1, and strings start at cell number 0. That is, the first element of an array uses the cell number 1, there is no cell zero for arrays. Strings start at cell number zero, so a string of 10 characters has cell numbers 0 through 9. FrameWork contains a set of predefined functions which we will discuss next. Strings are considered objects in Scripts. A simple definition of an object (in the way Scripts use them) would be a complex data type capable of executing a number of predefined functions. That is, FrameWork includes a number of complex data types that can be used in the code by manipulating references to instances of them. The fact that they are complex does not mean that they require extra work. In fact, it means that their complexity has been already reduced to a set of functions. Strings are the first object we will discuss. Strings have been already introduced as a data type. They can store a sequence of characters. Once they are declared we can use the following functions to manipulate them:

charAt()	Length()
compareTo()	substring()
indexOf()	toFloat()
String()	toInteger()
toByteArray()	DoubleToString()

charAt()

Syntax

```
MyString.charAt (int n);
```

Arguments

An integer value specifying the character number in the string

Return values

Returns the ASCII code for the nth character in the String called MyString.

Example

```
String MyString;
MyString = "Inspection Failed";
char thirdChar;
thirdChar = MyString.charAt(3);
```

Notes

Strings are indexed starting at zero, so the char variable should contain a 112 after the execution of the code above. The number 112 is the decimal ASCII representation of the letter “p”. Refer to Appendix A – ASCII Table of Characters, for a list of codes.

compareTo()

Syntax

```
Stringname.compareTo (String AnotherString);
```

Arguments

A second string to be compared with the original one

Return values

Integer value Result. Result = 0 if Stringname is the same as AnotherString. If Stringname is greater than Anotherstring (using ASCII values for the comparison) the result value is positive. If AnotherString is greater than Stringname, the result value is negative.

Example

```
String String1, String2;
int result;
String1 = "Code1";
String2 = "Code2";
result = String1.compareTo(String2);//result in this case
//is negative because “Code2” is greater than “Code1”
```

indexOf()

Syntax

```
Stringname.indexOf (String Searchstring);
```

Arguments

A second string containing a sequence of characters to be located in the original one

Return values

Integer value Position. It represents the position of the first character of the first occurrence of the string Searchstring in stringname. The value is -1 if the string is not found.

Example:

```
String MyString;
MyString = "string test";
result = MyString.indexOf("ing");//result should contain
//a 3 after executing this code
```

Notes

Strings are indexed starting at zero.

length()

Syntax

```
Stringname.length();
```

Return values

Integer value length. It represents the number of characters in the string Stringname

Example:

```
String MyString;
```

```
int result;
MyString = "number of characters";
result = MyString.length();//result should contain a 20
```

Notes

Even though there are 20 characters in the string, remember that they are indexed from 0 to 19.

substring()

Syntax

```
Stringname.substring (int beginIndex, int endIndex);
```

Arguments

Integer values indicating the index of the first and last characters to be extracted

Return values

String containing the characters from beginIndex to endIndex positions of the original string.

Example

```
String Mystring, SubString;
int startIndex, endIndex;
MyString = "number of characters";
startIndex = 4;
endIndex = 12;
SubString = MyString.substring(startIndex,endIndex);
//the string Substring should contain the string “er of cha”
//after execution
```

Notes

Strings are indexed starting at zero.

toFloat()

Syntax

```
Stringname.toFloat()
```

Return values:

String Stringname as double (float) data type. The value is 0.0 if the string cannot be converted.

Example

```
double dist;
String Data;
Data = "0.5";
dist = Data.toFloat();
```

Notes

The toFloat() command can convert a `String` to a floating point data type. This is useful when a floating point value is transferred as a `String` and needs to be converted back to floating point value.

toInteger()

Syntax

```
Stringname.toInteger()
```

Return values

The string Stringname as int (integer) data type. The value is 0 if the string cannot be converted.

Example

```
int msec;
String MilliSecs;
MilliSecs = "18356";
msec = MilliSecs.toInteger();
```

Notes

The toInteger() command can convert a `String` to an `int` data type. This is useful when an integer value is transferred as a `String` and needs to be converted back to integer.

String()

Syntax

```
String(byte[] b);
```

Arguments

A byte array

Return values

A string with characters that correspond to the ASCII codes in the bytes of the original array.

Example:

```
byte b[] = new byte[3];//declare an array of bytes
String DVTStr;
b[0] = 68; // ASCII code for 'D'
b[0] = 86; // ASCII code for 'V'
b[0] = 84; // ASCII code for 'T'
DVTStr = String(b); // DVTStr now contains "DVT"
```

toByteArray()

Syntax

```
MyString.toByteArray();
```

This virtual method converts a string into a byte array.

Return values

An array of byte variables containing the ASCII values of the characters in the string.

Example:

```
byte buff[] = new byte[3];
String LogoString = "DVT";
buff = LogoString.toByteArray(); // now buff[0]=68, buff[1]=86,
//and buff[2]=84
```

DoubleToString()

Syntax

```
DoubleToString(double val, int num_places);
```

Arguments

Double value to be converted to String and Integer value indicating the number of decimal places to be used for the conversion.

Return values

String containing the digits that represent the double value.

Example:

```
double piSq;
String myStr="";
piSq = 3.141592654 * 3.141592654;//evaluates to
//9.869604403666763716
myStr = DoubleToString(piSq,5);//myStr now contains 9.86960
```

Special Characters

\'

Syntax

```
Stringname = "stringchars\'morechars";
Stringname = "stringchars" + "\'" + "morechars";
```

Return values

A string with the ’ character included (in this case: stringchars'morechars)

\b

Syntax

```
Stringname = "stringchars\bmorechars";
Stringname = "stringchars" +"\b"+ "morechars";
```

Return values

String with a backspace character included

Notes

When shown on the Result Table, the \b is printed as a square symbol.

\"

Syntax

```
Stringname = "stringchars\"morechars";
Stringname = "stringchars" + "\"" + "morechars";
```

Return values

A string with " character included (in this case: stringchars"morechars)

\

Syntax

```
Stringname = "stringchars\\morechars";
Stringname = "stringchars"+"\\"+morechars";
```

Return values

A string with a backslash character included

\n

Syntax

```
Stringname = "stringchars\nmorechars";
Stringname = "stringchars" + "\n" + "morechars";
```

Return values

A string with new line character included

Notes

When shown on the Result Table, the \n is printed as a square symbol.

\r

Syntax

```
Stringname = "stringchars\rmorechars";
Stringname = "stringchars" + "\r" + "morechars";
```

Return values

A string with a return character included

Notes

When shown on the Result Table, the \r is printed as a square symbol.

\t

Syntax

```
Stringname = "stringchars\tmorechars";
Stringname = "stringchars" + "\t" + "morechars";
```

Return values

A string with the tab character included

Notes

When shown on the Result Table, the \t is printed as a square symbol.

\x

The characters \x can be used in a string to insert control characters according to their hexadecimal ASCII number. The \x is typically used for less common control characters. For the more common control characters like Carriage Return (\xOD) or Line Feed (\x0A) the specially control characters \r and \n can be used.

Syntax

```
Stringname = "stringchars\xODmorechars";
```

Notes

The value 0D added to the string is the hexadecimal value of the character in ASCII standards.

Chapter 3 - Basic Script Syntax

Now that the different data types have been explained, it is time to discuss some basic syntax about DVT scripts that will let us move on to more advanced topics.

Comments

It is good programming practice to include explanations in our code that describe the purpose of the program itself and of every block of code (if not every statement). These are comments; the more comments a certain script has, the more readable and easier to troubleshoot it becomes. Since we are writing code, we must include comments in a way that the SmartImage Sensor does not attempt to execute them. They have to be part of the code, but isolated from it because it is intended to be read by the author or by another person. DVT Scripts offer two different types of comments: single line comments and blocks of comments. Single line comments are used for short explanations (less than one line) about a minor step in the code that might not be easy to visualize. To create single line comments the user needs to start them with a sequence of two forward slash characters (`//`). The code illustrates the use of single line comments:

```
float distance1;//used to store distance from A to B
float distance2;//used to store distance from B to C
float totalDistance;//will hold the total distance from A to C
distance1 = 3215.56;//just a value for this example
distance2 = 2563.58;//just a value for this example
//now the total distance from A to C will be computed
totalDistance = distance1 + distance2;
```

This brief piece of code illustrates the use of comments. When the SmartImage Sensor attempts to run the Script, it will omit the words that come after the double forward slash. The user must keep in mind that these are single line comments. If the comment is too long and jumps onto the next line it will cause an error unless that line is started with the sequence of two forward slash characters, too.

Multi line comments are used to include a number of lines of explanations about a certain part of the script or the entire script. Multi line comments start with the sequence of a forward slash character followed by an asterisk (`/*`), and end with the opposite sequence (`*/`). The code below is another way to comment the previous block of code using multi line comments:

```
/*the following variables will be used to extract the distances
from A to B and from C to D and to compute the total distance
from A to C*/
float distance1;
float distance2;
float totalDistance;
distance1 = 3215.56;
distance2 = 2563.58;
totalDistance = distance1 + distance2;
```

Notice that the comments jump to the next line, but since it was declared as a multi line comment, it will be accepted by the compiler.

Key Words and Reserved Words

Just like in any programming language, there is a set of words that cannot be used for variable names. FrameWork includes a new feature to easily tell when we are using a reserved word or a key word. When we use those, they will be automatically highlighted in different colors. This is intended to simplify the interpretation of the code by the user and to avoid using reserved word as variable names. If your script editor does not highlight the keywords in colors, you should verify that you have a copy of the file “DVTScript.ini” in your FrameWork directory. Table 2 includes a list of words that are not used in the code but are still not supported by the compiler.

package	transient	default
import	volatile	finally
public	class	continue
protected	extends	break
private	implements	try
static	throws	throw
abstract	super	catch
final	interface	
native	switch	
synchronized	case	

Table 2: List of reserved words that are not currently used in DVT Scripts.

Even though these words are not currently used in Scripts, they are highlighted as reserved by the compiler.

Conditional Expressions and Control Structures

Now that we have discussed the basics of data types and syntax, we can explain the different ways that we have to change the course of a program.

If Statements

The `if` and `if/else` statements are a major component of the language. In general, users employ scripts to perform a task that is not performed anywhere else in the software. A common situation arises when the application requires that a decision be made based on the numerical output of many SoftSensors. In those cases, we need tools that can read those numerical outputs and analyze them. Those tools are Scripts, and in those cases is where the `if` and `if/else` statements become very useful. This type of statement is used to take alternate roads in the code. Figure 12 shows a graphical interpretation of how the program changes direction based on the condition we specify.

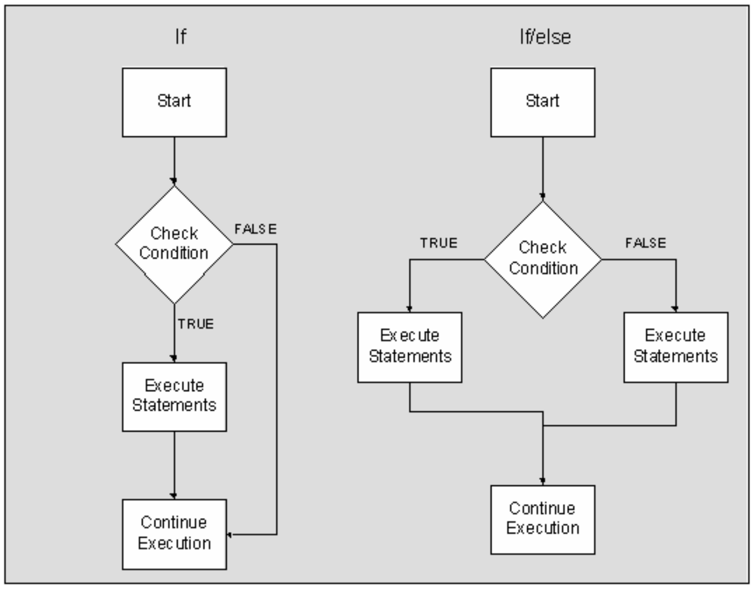


Figure 12: Graphical interpretation of the `if` and `if/else` statements.

The syntax for the basic `if` statement is shown below:

```
if (condition)
      statement;
```

Notice how the line containing the `if` statement does not have a semicolon at the end, only the actual statement to be executed has it. The code shown here indicates that if the condition evaluates to `true`, the statement will be executed, otherwise, it will be skipped. This syntax is good for single statements like in this case, but if the condition required more than one statement to be executed, we would need the following syntax:

```
if (condition)
{
      statement1;
      statement2;
      statement3;
}
```

In this case three statements are executed if the condition evaluates to `true`. Another option for `if` statements is the case where a different set of statements has to be executed if the condition evaluates to false. These situations use the `else` statement as follows:

```
if (condition)
{
      statement1;
      statement2;
      statement3;
}
else
{
      statement4;
      statement5;
      statement6;
}
```

In this case, the upper three statements are executed if and only if the condition evaluates to true, otherwise, the last three statements are executed. Finally, and to increment the flexibility of this function even more, we can nest and concatenate these statements. We could evaluate a number of conditions and based on them execute different sets of statements. The code below illustrates this:

```
if (condition1)
{
      if (condition2)
      {
            statement1; //condition1 = true and condition2 = true
      }
      else
      {
            statement2; //condition1 = true and
                        //condition2 = false
      }
}
else if (condition3)
{
      if (condition4)
      {
            statement3; //condition1 = false, condition3 = true
```

```
                        //and condition 4 = true
      }
      else
      {
            statement4; //condition1 = false, condition3 = true
                        //and condition 4 = false
      }

}
else
{
      if (condition5)
      {
            statement5; //condition1 = false, condition3 = false
                        //condition5 = true
      }
      else
      {
            statement6; //condition1 = false, condition3 = false
                        //condition5 = false
      }
}
```

In this case we have five different conditions to be tested and 6 different statements to be executed. This code should illustrate how to make good use of comments and indentation. Every statement to be executed is followed by a comment explaining why it is executed. Also, the curly brackets that separate blocks of statements are indented to make the code easier to read. As a programming practice, the reader should trace the code and read the comments to fully understand the program.

Loops

The next control structure consists of the repetition of a set of statements in a Script. In many cases some statements need to be executed a number of times. Looping structures are designed for that purpose. There are three different looping structures: the `while` loop, the `do/while` loop, and the `for` loop. The difference between the different types of loop is given by how the condition for repetition is evaluated and by how many times they execute. The main characteristics are summarized here:

`for` loops are used when the number of repetitions is known. When the user knows exactly the number of repetitions to perform beforehand, a `for` loop should be used. This type of loop requires the use of a counter that gets updated after every iteration.

`while` loops are used when the exact number of times that the statements are evaluated varies. In those cases a condition is set and the loop checks that condition before every iteration. In this case, if the condition is `false,` the statements are not executed even once

`do/while` loops are very similar to `while` loops except that in this case the condition is evaluated after every iteration, so if the condition is false, the statements still get evaluated once.

Figure 13 shows a graphical representation of all three types of loop.

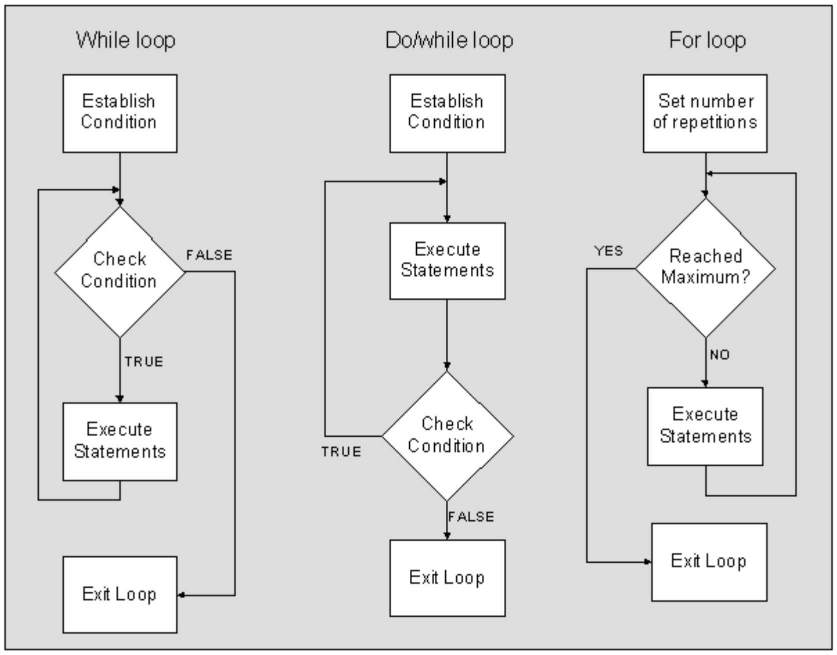


Figure 13: Representation of the functionality of the different types of loop.

The syntax for the `while` loop is shown below. It requires a condition and curly brackets to enclose the statements that need to be executed on every iteration.

```
while(condition)
{
      statement1;
      statement2;
}
```

In this case the code checks the condition, if it is true it executes the statements and it keeps doing that until the condition becomes false.

The syntax for the do/while loop is very similar, it is shown below:

```
do
{
      statement1;
      statement2;
}
while(condition);
```

Notice that this type of loop requires a semicolon after the `while(condition)` statement to indicate that there is no more code that belongs to that block.

The syntax for the `for` loop is slightly different, it requires a counter that will keep rack of the number of iterations. Three different ways to use the `for` loop are shown below:

```
//this method declares and initializes the variable
//inside the for statement
for(int count=0; count <= 5; count = count+1)
{
      statement1;
      statement2;
}

//this method declares and initializes the variable
```

```
//outside the for statement
int count2 = 0;
for(; count2 <= 5; count2 = count2+1)
{
      statement1;
      statement2;
}

//this method declares the variable outside the for statement and
//initializes it inside the for statement
int count3;
for(count3 = 0 ; count3 <= 5; count3 = count3+1)
{
      statement1;
      statement2;
}
```

In all cases the variable is automatically updated, there is no need to update the counter. When the `for` statement is used, the user specifies by how much to increment the counter after every iteration.

User-defined Functions

In the early stage of the process of making DVT Scripts fully Java compatible, users can already benefit by being able to create their own functions. This functionality was appropriate for the type of functions script performed, but as the power of scripts increased, a more effective way to write code was needed. A script today could potentially perform many complex tasks, and since all those tasks had to be included in a single block of code, scripts became more difficult to write and even more difficult to troubleshoot. By creating functions, users can make the code more modular, and even reusable.

A function is basically a set of statements designed for a certain purpose that could take in certain parameters and return other parameters. One can think of functions as workers where the script is the manager. The script does not have to perform specific tasks as long as there is a worker available to do it. Following this reasoning, the script would become simply a sequence of function calls. A simple example of a function would be a function that finds the highest value in a collection of data. That function needs to take in the collection of data (which could be a set of numbers) and return the highest number, so whenever the maximum of a group of numbers is needed, users could use the function. In this case the script that calls the function needs to provide it with the data to process and accept the value that the function returns. This is only valid within the script in which the function was created in (however, duplicates of the function can exist in other classes). If a problem arises with the script, every function can be individually tested which simplifies the process. It is significantly easier to find coding errors where few operations are performed than to find them in a very long sequence of statements.

In order to create user functions users need to follow some basic rules:

- Ensure that a function is called only after it was defined
- Follow standard syntax for the creation of the function
- Indicate the arguments they take in (the values it expects to see when it is called), if any
- Indicate the return type, which determines the type of variable to be returned by the function

Let us use an example to illustrate this, let us create an empty foreground script called “myScript” and create a function in it to find what the maximum of three numbers is; these numbers are the numbers of blobs found by three different Blob Selectors. Initially, the code looks like this:

```
class myScript
{
    public void inspect()
    {
        //Add your code here

    }
}
```

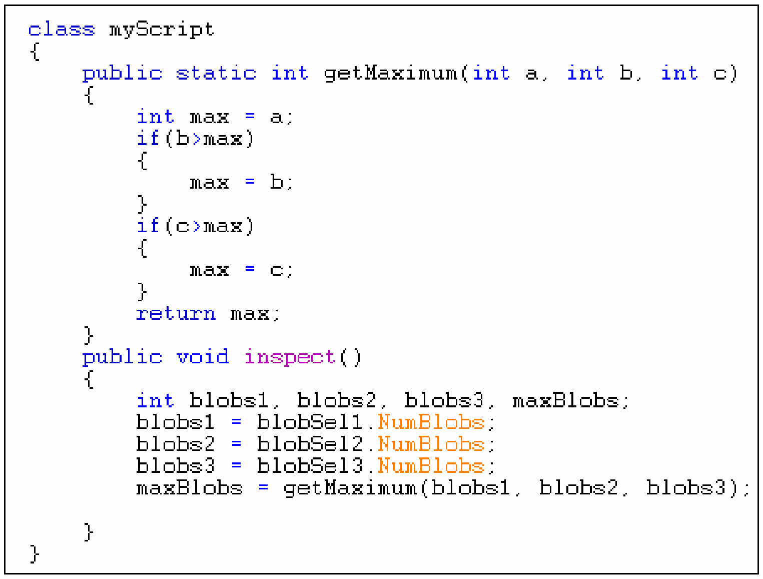
Next, we need a few variables to contain the actual number of blobs, so the next step would leave our script looking like this:

```
class myScript
{
    public void inspect()
    {
        int blobs1, blobs2, blobs3;
        blobs1 = blobSel1.NumBlobs;
        blobs2 = blobSel2.NumBlobs;
        blobs3 = blobSel3.NumBlobs;


    }
}
```

Now we need to start defining the function, so we are going to follow the steps mentioned before. The function needs to be declared before it is called, so our function will certainly be included above the inspect function. As of syntax, we have to mimic what the inspect function does and add the keyword “static” to it. We have to use the keyword “public” then the keyword “static” then the data type that is returned (void means nothing is returned) and then the name of the function. The last thing to consider is the list of arguments, the values that the function will take in. In our example, the function will take in 3 integer variables and return 1 integer value, so the script with the function added looks like this:

```
class myScript
{
    public static int getMaximum(int a, int b, int c)
    {


    }
    public void inspect()
    {
        int blobs1, blobs2, blobs3;
        blobs1 = blobSel1.NumBlobs;
        blobs2 = blobSel2.NumBlobs;
        blobs3 = blobSel3.NumBlobs;


    }
}
```

As the figure shows, the variables that the function takes in have arbitrary names. In fact, we can use any name for these parameters for their scope is limited to the scope of the function. That is, integer variables a, b, c, or any variable defined inside the function will not be available from anywhere else in the script but our function. This is a very important concept and users should always keep it in mind when writing scripts. All the variables must be defined inside functions, and that limits their scope. It was illegal in the past to have different variables with the same name. With the current implementation it is legal to do it, but it is never a good idea for it makes the code harder to read. Now our script is missing a few things: the computation for the maximum value, and the function call. After we add these items, the script looks like this:



```
class myScript
{
    public static int getMaximum(int a, int b, int c)
    {
        int max = a;
        if(b>max)
        {
            max = b;
        }
        if(c>max)
        {
            max = c;
        }
        return max;
    }
    public void inspect()
    {
        int blobs1, blobs2, blobs3, maxBlobs;
        blobs1 = blobSel1.NumBlobs;
        blobs2 = blobSel2.NumBlobs;
        blobs3 = blobSel3.NumBlobs;
        maxBlobs = getMaximum(blobs1, blobs2, blobs3);

    }
}
```

The question is, if max is not seen by the function inspect, how can we read its value? The answer is by using the return statement. When our function computes the maximum it returns the value and exits, this value can be read by the function that calls it by using as assignment operator like the figure shows.

More functions could be added as needed, and user functions can even call other user functions as long as they follow rule number 1 above: a function can only be called after it is defined. For example if we were to add a function that calls the getMaximum function and it is called by the inspect function, it would have to be defined after getMaximum but before the inspect function. The same happens with background scripts, except that for background scripts there is no inspect function but a main function instead.

Comparison operators

Until now, we have discussed a number of ways to check conditions. We have seen how to change the execution of the code based on certain conditions. Now we will see how to check those conditions. Every time we check a condition, we are evaluating something to see if it is `true` or `false`, so one way of checking conditions is to use a `boolean` variable. These variables, as explained before, can have only two values: `true` and `false`. Another way to do that is to use logical operators. Logical operators are used to compare values and produce a boolean result from the comparison. The most common operators are:

The equal operator (==)

Used for comparisons of basic data types

Syntax

```
argument1 == argument2;
```

Arguments

This operator accepts any basic data type as an argument. It can also be used to verify if an object is null by comparing the object with null.

Return values

Returns the boolean value of true if the arguments are equal, returns false otherwise

Example:

```
if (x==1)
{
      statement1;//statement performed if x is equal to 1
}
```

Notes

Do not attempt to use this operator to compare strings. Use the string function “compareTo().”

The less-than-or-equal-to operator (<=)

Used for comparisons of basic data types

Syntax

```
argument1 <= argument2;
```

Arguments

This operator accepts any basic data type as an argument.

Return values

Returns the boolean value of true if argument1 is less than or equal to argument 2, returns false otherwise

Example:

```
if(x<=1)
{
      statement1;//statement performed if x is less or equal to 1
}
```

The less-than operator(<)

Used for comparisons of basic data types

Syntax

```
argument1 < argument2;
```

Arguments

This operator accepts any basic data type as an argument.

Return values

Returns the boolean value of true if argument1 is less than argument2, returns false otherwise

Example

```
if(x<1)
{
      statement1;//statement performed if x is less than 1
}
```

The greater-than operator(>)

Used for comparisons of basic data types

Syntax

```
argument1 > argument2;
```

Arguments

This operator accepts any basic data type as an argument.

Return values

Returns the boolean value of true if argument1 is greater than argument2, returns false otherwise

Example

```
if(x>1)
{
      statement1;//statement performed if x is greater than 1
}
```

The greater-than-or-equal-to operator (>=)

Used for comparisons of basic data types

Syntax

```
argument1 >= argument2;
```

Arguments

This operator accepts any basic data type as an argument.

Return values

Returns the boolean value of true if argument1 is greater than or equal to argument2, returns false otherwise

Example

```
if(x>=1)
{
      statement1;//statement performed if x is greater or
                  //equal to 1
}
```

The not operator (!)

Syntax

```
!argument1;
```

Arguments

A boolean data type or an expression that evaluates to a boolean value (true or false)

Return values

Returns the boolean value of true if argument1 is false, returns true otherwise

Examples

```
//using a boolean variable
if (!badresult)
{
      statement1; //performed if the variable badresult is false
}

//using an expression
if(!(x>=1))
{
      statement1; //statement executed if x is not greater
                  //than or equal to 1
}
```

The not-equal operator (!=)

Used for comparisons of basic data types

Syntax

```
argument1 != argument2;
```

Arguments

This operator accepts any basic data type as an argument. It can also be used to verify if an object is null by comparing the object with null.

Return values

Returns the boolean value of true if the arguments are not equal, returns false otherwise

Examples

```
if (x!=1)
{
      statement1; //statement performed if x = something
                  //other than 1
}
```

The or operator (||)

Used to concatenate conditional operations

Syntax

```
(boolean argument1) || (boolean argument2);
```

Arguments

Two boolean data types or expressions that evaluate to boolean values (true or false)

Return values

Returns the boolean value of true if either argument1, argument2, or both are true, returns false otherwise

Example:

```
if ((x==1) || (y==2))
{
      statement1;//performed if either x equals 1 or y equals 2
}
```

The and operator (&&)

Used to concatenate conditional operations

Syntax

```
(boolean argument1) && (boolean argument2);
```

Arguments

Two boolean data types or expressions that evaluate to boolean values (true or false)

Return values

Returns the boolean value of true if both arguments are true, returns false otherwise

Example:

```
if ((x==1) && (y==2))
{
      statement1;//performed if x equals 1 and y equals 2
}
```

Bit Manipulation

In many cases it is necessary to perform operations that access or modify single bits. So far, we have not seen any operator capable of doing that. This section discusses why we need those operators and describes the use of them.

Bits and Bytes

We have mentioned so far that the size of DVT registers is 8-bits. That means that we have eight individual binary digits in every register. Those binary digits can take only the values of 0 and 1; FrameWork provides some bit operators to access those digits individually. A byte is a group of 8 bits, so one DVT register is the same as one byte. That is why the `byte` data type in scripts takes only one register; it is represented by eight bits. The `int` data type for example, has been defined as a 32-bit integer value. For that reason, it has been said that `int` variables take four DVT registers when saved to memory. One example of the use of bit manipulators is the use of Inputs and Outputs. DVT SmartImage Sensors have a total of 32 inputs and 64 outputs. In order access/change a single output, we need to get a 64-bit word and use it with the appropriate mask. The example below illustrates this process with a 16-bit word. We have a set of 16 values mapped into a 16-bit word. Every bit in this word represents a data point (for example an output line). Our task is to set bit 11 to 0. The process is very simple and it is shown in Table 3. First, a variable is created with the same number of bits as our data (16 bits). This variable could be a `short` data type. The variable is initialized to 1 to have a nonzero bit. It is the shifted to the left, so the nonzero bit is bit number 11. It is then inverted and anded with the original variable.

	bit 15	**bit 14**	**bit 13**	**bit 12**	**bit 11**	**bit 10**	**bit 9**	**bit 8**	**bit 7**	**bit 6**	**bit 5**	**bit 4**	**bit 3**	**bit 2**	**bit 1**	**bit 0**
Original Data:	**0**	**1**	**1**	**1**	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**0**
STEP 1																
16-bit Variable:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
STEP 2																
Shift the Variable	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
STEP 3																
Invert Variable	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1
STEP 4																
Inverted Variable	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1
Original Data:	0	1	1	1	1	0	1	0	1	0	1	0	1	0	1	0
Apply bitwise AND	**0**	**1**	**1**	**1**	**0**	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**0**

Table 3: Use of the bit manipulation operators.

This process uses three bit manipulation operators: shift, inversion and the AND operator. All the operators available form FrameWork are explained next.

The bitwise OR operator

The “|” is the bit “OR” operator, it reads the data in a bit from two variables and if either one is set to 1 (or both are set to 1), it returns a 1, otherwise it returns a zero.

Syntax

```
variable1 | variable2;
```

Arguments

Two integer data types with the same number of bits

Return values

Returns a variable of the same size as the argument with bit values corresponding to the result of a comparison between the bits in both arguments according to the truth table below.

argument1	argument2	result
0	0	0
0	1	1
1	0	1
1	1	1

Example:

```
byte i, j, mask;
i= 8; //bit values of 00001000
mask = 4; //bit values of 00000100
j = i|mask;  //sets the third bit from the right: 00001100 =
decimal 12
```

The bitwise AND operator

The “&” is the bit “AND” operator, it reads the data in a bit from two variables and if both are set to 1, it returns a 1, otherwise it returns a zero.

Syntax

```
variable1 & variable2;
```

Arguments

Two integer data types with the same number of bits

Return values

Returns a variable of the same size as the argument with bit values corresponding to the result of a comparison between the bits in both arguments according to the truth table below

argument1	argument2	result
0	0	0
0	1	0
1	0	0
1	1	1

Example:

```
byte data, dataMask;
dataMask = 4; //bit values of 00000100
if ( (data & dataMask) != 0){
      //this statement will be executed if the third bit from the
      //right in variable data is a 1.
}
```

The bitwise XOR operator

The “^” is the bit “XOR” operator, it reads the data in a bit from two variables; if exactly one of them is set to 1, it returns a 1, otherwise it returns a zero.

Syntax

```
variable1 ^ variable2;
```

Arguments

Two integer data types with the same number of bits

Return values

Returns a variable of the same size as the argument with bit values corresponding to the result of a comparison between the bits in both arguments according to the truth table below:

argument1	argument2	result
0	0	0
0	1	1
1	0	1
1	1	0

Example:

```
byte i, j, mask;
i=20;            //00010100
mask = 12;       //00001100
j = i^mask;  //j is 00011000 or 24
```

The bitwise NOT operator

The “~” is the bit “NOT” operator, it reads the data in a bit from a variable and inverts the state of every bit

Syntax

```
~variable1;
```

Arguments

An integer data type

Return values

Returns a variable of the same size as the argument with bit values inverted according to the truth table below

Argument	result
0	1

1	0

Example:

```
byte data1;
data1 = 20;   //binary value of: 00010100
data1 = ~data1;  //data1 is now: 11101011
```

The Signed Left Shift operator

The “<<” is the bit “Left Shift” operator, it takes two arguments and shifts the bits in the first argument to the left a number of places specified by the second argument.

Syntax

```
var1 = var1<<3;
```

Arguments

An integer data type and an integer value

Return values

A variable of the same size as the argument with the bits shifted to the left a number of places equal to the value of the integer argument

Example:

```
byte data1;
data1 = 10;   //binary value of: 00001010
data1 = data1<<3;//data1 is now: 01010000 (decimal 80)
```

Notes:

When a binary value is shifted to the left one place its value is duplicated, so in this case we started with a 10 and shifted left three times to get first a 20, then a 40 and finally an 80. Note that new positions are filled with zeros (0), as demonstrated in the above example.

The Signed Right Shift operator

The “>>” is the bit “Right Shift” operator, it takes two arguments and shifts the bits in the first arguments to the right a number of places specified by the second argument.

Syntax

```
var1 = var1>>3;
```

Arguments

An integer data type and an integer value

Return values

A variable of the same size as the argument with the bits shifted to the right a number of places equal to the value of the integer argument

Example:

```
byte data1;
data1 = 80;   //binary value of: 01010000
data1 = data1>>3;//data1 is now: 00001010 (decimal 10)
```

Notes

When a binary value is shifted to the right one place its value is halved, so in this case we started with an 80 and shifted right three times to get first a 40, then a 20 and finally a 10. Note that when bit shifting a negative number with the signed right shift operator, the leading bit position is always filled with a one (1), otherwise it's filled with a zero.

The Unsigned Right Shift operator

The ">>>" is the bit "Right Shift" operator, it takes two arguments and shifts the bits in the first arguments to the right a number of places specified by the second argument.

Syntax

```
var1 = var1>>>3;
```

Arguments

An integer data type and an integer value

Return values

A variable of the same size as the argument with the bits shifted to the right a number of places equal to the value of the integer argument

Example:

```
byte data1;
data1 = 80;   //binary value of: 01010000
data1 = data1>>>3;//data1 is now: 00001010 (decimal 10)
```

Notes

When a binary value is shifted to the right one place its value is halved, so in this case we started with an 80 and shifted right three times to get first a 40, then a 20 and finally a 10. Note that with the unsigned right shift operator, the leading bit position is always filled with a zero (0). This means an unsigned right shift operation always results in a positive number.

Miscellaneous Functions

The clock() function

The `clock()` function returns the number of milliseconds since the last SmartImage Sensor power-up as an integer value.

Syntax

```
clock();
```

Arguments

None

Return values

The number of milliseconds since power-up as an integer value

Example:

```
int totalTime;
totalTime = clock();
      //execute a number of statements here
totalTime = clock() - totalTime;//total time will indicate
                        //the duration in milliseconds of the
                        // execution of the statements
```

Notes

Do not capitalize ‘clock’ or the script will return a syntax error.

The sleep() function

The `sleep()` function takes in an argument (integer data type) and pauses the execution of the code for the number of milliseconds specified as the argument.

Syntax

```
sleep(int totalPause);
```

Arguments

An integer value

Return values

None

Example:

```
byte flags;
while(true)
{
      flags = RegisterReadByte(100);
      if((flags & 00001000) == 0)
      {
            //execute some commands
      }
      sleep(250);//pause execution for 250 msec
}
```

This example shows a loop that reads a value from a register and tests it to determine if certain some operations need to be executed. The loop executes every 250 milliseconds to avoid overloading the processor. The `sleep()` command is used to set this interval.

Note

If the argument is negative, the function uses a zero instead of giving an error.

The DebugPrint() function

This function prints a string to the Script Debug Window located in the main Edit Menu of FrameWork. This function is very valuable for troubleshooting your scripts.

Syntax

```
DebugPrint(String str);
```

Arguments

String to display in the Script Debug Window

Return values

None.

Example

```
if(data1 < data2)
{
      //execute statements
      dataDiff = data2-data1;
      DebugPrint("data1 was less that data2 by: "+dataDiff);
}
else
{
      //execute statements
      dataDiff = data1-data2;
      DebugPrint("data2 was less that data1 by: "+dataDiff);
}
```

Notes

In this case a condition is being checked. The `DebugPrint()` statement is used to show the relevant data in both cases.

Every time a DebugPrint statement is executed, a new line appears in the Script Debug Window (Available from the main Edit menu). This line contains the following parts:

A prompt indicating new line($)

A time stamp in milliseconds that gets reset to zero on power-up

An integer value containing the SoftSensor ID of the Foreground Script that executed the DebugPrint statement if it is from a Foreground Script. In case the statement comes from a Background Script it contains the Background Script ID

An integer value indicating if the string comes from a Background Script (value = 2) or a Foreground Script (value = 0).

The string containing the message

Sample display for a string coming from a Foreground Script with sensor ID 4:

$ 3214562 4 0 Here is the String

SetMatchString()

This function is exclusive of Background Scripts and is the only function in a Background Script that can access and modify a SoftSensor. It is used to change the match string for a reader SoftSensor (OCR, DataMatrix, or Barcode).

Syntax

```
SetMatchString(Product P, String SensorName, String NewString);
```

Arguments

Product P: A product object containing the product where the SoftSensor to be changed is located

String SensorName: A string containing the name of the SoftSensor to be accessed and modified.

String NewString: A string that is to be set as the match string in the specified SoftSensor.

Return values

Int result: result = 1 for a successful operation. Result = -1 for errors (ex. Sensor not found, called from a foreground script, etc.)

Example

```
//change the match string in sensor CodeReader
//of the product Labels to “AC765”
Product P;
P = GetProduct(“Labels”);
SetMatchString(P , “CodeReader”, “AC765”);
```

Note

This is the only command in a background script that allows the user to access a SoftSensor. Users should remember that SoftSensor access is given to Foreground Scripts only.

GetImageID()

This function is exclusive of Foreground Scripts. It is used to obtain the image ID of the image being inspected by the product containing the script.

Syntax

```
int imgID = GetImageID ();
```

Arguments

NONE

Return values

Int imgID: integer number containing the image ID

Example

```
//add the image ID to the script string
//to send out via DataLink
int imgID = GetImageID();
myScript.String = “Current Image ID = ” + imgID;
```

Chapter 4 – Accessing SoftSensor data

This chapter describes how to get data from SoftSensors. In many cases, users need to extract information from the SoftSensors to perform calculations based on those or to send out to an external device. Foreground Scripts (also called Script SoftSensors) can be used to extract the necessary data by using a combination of commands and syntax. As explained before, this type of Script reside at the product level thus gaining access to all the SoftSensors (even to other foreground scripts) in the product. Both the commands needed and the syntax used are explained in this chapter.

Syntax for Basic SoftSensor Data Extraction

In order to gain access to SoftSensor data from scripts, users simply need to create a variable and assign the desired data from the desired SoftSensor to it. To reference the individual SoftSensor parameters, the user needs to use the following syntax:

```
<SoftSensor name>.<SoftSensor parameter>
```

Let us start with a basic example. We have a particular SoftSensor from which we need to extract the X and Y position. That is, the SoftSensor is likely to be detecting the presence of a certain part. We need those coordinates to perform a certain calculation. The SoftSensor name is locator; the simple syntax needed to perform this task is shown below:

```
int posX, posY;
posX = locator.Position.X;
posY = locator.Position.Y;
```

This simple procedure transfers the X and Y coordinates of the position determined by the SoftSensor to the integer variables posX, and posY. This is a very basic example, what if we wanted to make sure that the locator passed the inspection? In many cases, the SoftSensor looking for a certain position can be set to pass or fail when the object has changed the location too much or is not present at all. The new Script would look like this:

```
int posX, posY;
if(locator.Result == PASS)
{
      posX = locator.Position.X;
      posY = locator.Position.Y;
}
```

In this case, we are declaring the variables but only accessing the data from the other SoftSensor if that SoftSensor passed the inspection, otherwise we do not execute any commands. Notice the use of all uppercase letters in the keyword `PASS.` This is required for the compiler to correctly interpret the commands. Another way to refer to the result is the numerical value. The inspection result from a SoftSensor is assigned a numerical value. The value zero means that the SoftSensor passed the inspection. A negative value indicates that the inspection result for that SoftSensor was `FAIL`, whereas a positive value for the SoftSensor result indicates a result of `WARN`. The absolute value of these numerical values indicates the cause for failure (or warning) which is displayed in the result table. It is recommended to use the strings `PASS`, `WARN`, and `FAIL` to make the scripts more readable and simplify troubleshooting.

Our example above referred to a translation SoftSensor, but different types of SoftSensor can provide different sets of data, for instance, an OCR reader can provide a string as the output for a particular image, a measurement SoftSensor can provide a distance, and so on. The next section describes the type of parameters available form the different SoftSensors. Readers should be familiar with the types of SoftSensor to better understand the parameters explained here.

Translation SoftSensor Parameters

Translation SoftSensors offer the following parameters:

Result: overall result of the inspection of the part as an integer value (0 for PASS, positive for WARN, and negative for FAIL)

Position.X: absolute x-coordinate of the point passed as a position reference by this SoftSensor.

Position.Y: absolute y-coordinate of the point passed as a position reference by this SoftSensor.

MinIntensity: lowest intensity value found in the area analyzed by the SoftSensor (in levels from 0 to 255).

MaxIntensity: highest intensity value found in the area analyzed by the SoftSensor (in levels from 0 to 255).

Threshold: threshold level (in levels from 0 to 255).

Contrast: indicates the contrast found in the area analyzed by the SoftSensor (in levels from 0 to 255).

TransformedPoint.X: transformed absolute x-coordinate of the point passed as a position reference by this SoftSensor. Requires the use of a coordinate system SoftSensor.

TransformedPoint.Y: transformed absolute y-coordinate of the point passed as a position reference by this SoftSensor. Requires the use of a coordinate system SoftSensor.

Rotation SoftSensor Parameters

Rotation SoftSensors offer the following parameters:

Result: overall result of the inspection of the part as an integer value (0 for PASS, positive for WARN, and negative for FAIL)

Position.X: for the rotational arc, this represents the absolute X-coordinate of the center of the arc, for the rotational parallelogram this represents the absolute X-coordinate of the point passed as a position reference by this SoftSensor.

Position.Y: for the rotational arc, this represents the absolute Y-coordinate of the center of the arc, for the rotational parallelogram this represents the absolute Y-coordinate of the point passed as a position reference by this SoftSensor.

Point.X: for the arc tool it represents the absolute x-coordinate of the point of intersection of the arc and the desired feature. For the parallelogram tool it has the same value as the Position.X field.

Point.Y: for the arc tool it represents the absolute y-coordinate of the point of intersection of the arc and the desired feature. For the parallelogram tool it has the same value as the Position.Y field.

MinIntensity: lowest intensity value found in the area analyzed by the SoftSensor (in levels from 0 to 255).

MaxIntensity: highest intensity value found in the area analyzed by the SoftSensor (in levels from 0 to 255).

Threshold: threshold level (in levels from 0 to 255).

Contrast: indicates the contrast found in the area analyzed by the SoftSensor (in levels from 0 to 255).

TransformedPoint.X: transformed absolute x-coordinate of the point passed as a position reference by this SoftSensor. Requires the use of a coordinate system SoftSensor.

TransformedPoint.Y: transformed absolute y-coordinate of the point passed as a position reference by this SoftSensor. Requires the use of a coordinate system SoftSensor.

Angle: the angle calculated by the SoftSensor. Absolute angles are reported as negative if they are measured counterclockwise, and as positive if they are measured clockwise follows:

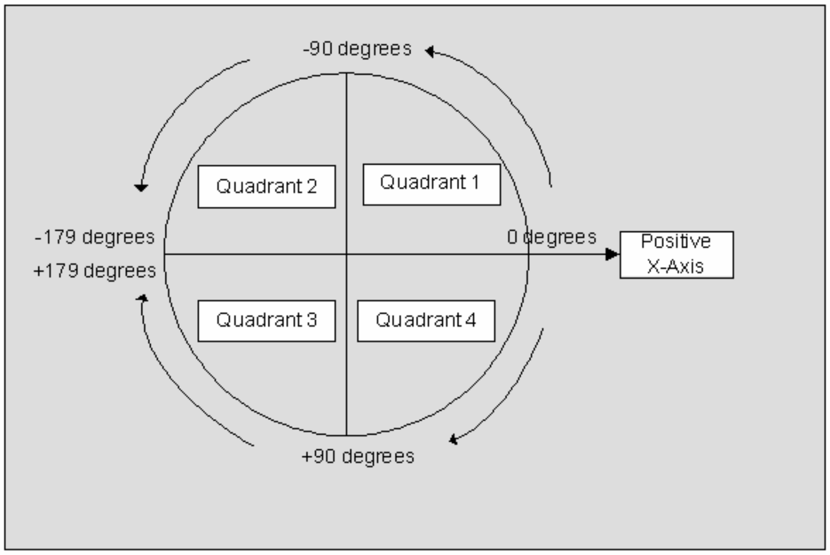
0 to +90 degrees: quadrant 4

90 to 180 degrees: quadrant 3

0 to -90 degrees: quadrant 1

-90 to -180 degrees: quadrant 2

The figure below illustrates the value of the angles relative to quadrants and the positive X-axis



Intensity SoftSensor Parameters

Intensity SoftSensors offer the following parameters:

Result: overall result of the inspection of the part as an integer value (0 for PASS, positive for WARN, and negative for FAIL)

MinIntensity: lowest intensity value found in the area analyzed by the SoftSensor (in levels from 0 to 255).

MaxIntensity: highest intensity value found in the area analyzed by the SoftSensor (in levels from 0 to 255).

Threshold: threshold level (in levels from 0 to 255).

Contrast: indicates the contrast found in the area analyzed by the SoftSensor (in levels from 0 to 255).

MeanIntensity: average intensity value found in the area analyzed by the SoftSensor (in levels from 0 to 255).

MedianIntensity: median intensity value found in the area analyzed by the SoftSensor (in levels from 0 to 255).

Percent: percent of the pixels in the SoftSensor that are below threshold (dark pixels) scaled by 100 (which gives it a range from 0 to 10000). For example, if the SoftSensor reports 97.15% bright area, it contains 2.85% dark area, so this field would return 285.

IVal: array of 255 cells containing integer values that represent the number of pixels in the SoftSensor at every intensity level. Intensity levels range from 0 to 255, this array has cells 1 through 256 to represent every intensity level.

TotalPixelCount: the total number of pixels in the SoftSensor.

EdgeCount SoftSensors

EdgeCount SoftSensors expose the following parameters:

Result: overall result of the inspection of the part as an integer value (0 for PASS, positive for WARN, and negative for FAIL)

MinIntensity: lowest intensity value found in the area analyzed by the SoftSensor (in levels from 0 to 255).

MaxIntensity: highest intensity value found in the area analyzed by the SoftSensor (in levels from 0 to 255).

Threshold: threshold level (in levels from 0 to 255).

Contrast: indicates the contrast found in the area analyzed by the SoftSensor (in levels from 0 to 255).

NumEdges: the number of edges found in the image according to user specifications.

FeatureCount SoftSensors

FeatureCount SoftSensors expose the following parameters:

Result: overall result of the inspection of the part as an integer value (0 for PASS, positive for WARN, and negative for FAIL)

MinIntensity: lowest intensity value found in the area analyzed by the SoftSensor (in levels from 0 to 255).

MaxIntensity: highest intensity value found in the area analyzed by the SoftSensor (in levels from 0 to 255).

Threshold: threshold level (in levels from 0 to 255).

Contrast: indicates the contrast found in the area analyzed by the SoftSensor (in levels from 0 to 255).

NumFeatures: the number of features found in the image according to user specifications.

Measurement SoftSensors

Measurement SoftSensors offer access to many parameters, some of them unique to the type of Measurement SoftSensor being used. The common parameters to all the measurement SoftSensors are:

Result: overall result of the inspection of the part as an integer value (0 for PASS, positive for WARN, and negative for FAIL)

MinIntensity: lowest intensity value found in the area analyzed by the SoftSensor (in levels from 0 to 255).

MaxIntensity: highest intensity value found in the area analyzed by the SoftSensor (in levels from 0 to 255).

Threshold: threshold level (in levels from 0 to 255).

Contrast: indicates the contrast found in the area analyzed by the SoftSensor (in levels from 0 to 255).

The rest of the parameters are unique to every SoftSensor. Even though in many cases the parameters share the name, the functionality is not the same. The parameters for every type of Measurement SoftSensor are:

For the basic Line SoftSensor:

Position.X: absolute x-coordinate of the location of the edge found. Used for positioning.

Position.Y: absolute y-coordinate of the location of the edge found. Used for positioning.

Point.X: absolute x-coordinate of the location of the edge found. In this case the value is reported as a floating point number with sub pixel accuracy if a gradient based threshold is used.

Point.Y: absolute y-coordinate of the location of the edge found. In this case the value is reported as a floating point number with sub pixel accuracy if a gradient based threshold is used.

For the Measure Across Line SoftSensors:

Position.X: absolute x-coordinate of the location of the center of the feature found. Used for positioning.

Position.Y: absolute y-coordinate of the location of the center of the feature found. Used for positioning.

Point.X: absolute x-coordinate of the location of the center of the feature found. In this case the value is reported as a floating point number with sub pixel accuracy if a gradient based threshold is used.

Point.Y: absolute y-coordinate of the location of the center of the feature found. In this case the value is reported as a floating point number with sub pixel accuracy if a gradient based threshold is used.

For Area Edge Line SoftSensor:

Position.X: integer value indicating absolute x-coordinate of the intersection of the edge found and the first scan line (the edge of the SoftSensor). Used for position reference.

Position. Y: integer value indicating absolute y-coordinate of the intersection of the edge found and the first scan line (the edge of the SoftSensor). Used for position reference.

Point.X: floating point value indicating absolute x-coordinate of the intersection of the edge found and the first scan line (the edge of the SoftSensor). If the SoftSensor uses a gradient based threshold, this number will report sub pixel accuracy.

Point.Y: floating point value indicating absolute y-coordinate of the intersection of the edge found and the first scan line (the edge of the SoftSensor). If the SoftSensor uses a gradient based threshold, this number will report sub pixel accuracy.

Angle: angle determined by the edge found and the edge of the SoftSensor where all the scanning lines begin. As viewed from the intersection point between the two, the angle will be positive if the direction as defined is clockwise, negative if it is counterclockwise.

Straightness: reports the distance in pixels, between the two data points that are farthest away from the output line in each direction. If the edge is perfectly straight, this value should be close to zero. As the edge becomes noisier, the value increases.

NumEdgePoints: number of data points collected (the number of intersections between scanning lines and part edges).

EdgePoint.X[]: array containing the absolute x-coordinate of the data points used for the line fitting (resulting from intersection of scan lines and part edges). It contains "NumEdgePoints" cells.

EdgePoint.Y[]:array containing the absolute y-coordinate of the data points used for the line fitting (resulting from intersections of scan lines and part edges). It contains "NumEdgePoints" cells.

For Measure Across Area:

Position.X: integer value indicating absolute x-coordinate of the intersection of the output line (midline between the edges found) and the first scan line (the edge of the SoftSensor). Used for position reference.

Position. Y: integer value indicating absolute y-coordinate of the intersection of the output line (midline between the edges found) and the first scan line (the edge of the SoftSensor). Used for position reference.

Point.X: floating point value indicating absolute x-coordinate of the intersection of the output line (midline between the edges found) and the first scan line (the edge of the SoftSensor). If the SoftSensor uses a gradient based threshold, this number will report sub pixel accuracy.

Point.Y: floating point value indicating absolute y-coordinate of the intersection of the output line (midline between the edges found) and the first scan line (the edge of the SoftSensor). If the SoftSensor uses a gradient based threshold, this number will report sub pixel accuracy.

Angle: angle determined by the lines that the SoftSensor computes as the true edges of the part being measured. The angle is measured from the line closer to the origin of the SoftSensor to the other one. As viewed from the intersection point between the two, the angle will be positive if the direction as defined is clockwise, negative if it is counterclockwise.

Straightness: this is the same concept as in the Area Edge Line SoftSensor but in this case the SoftSensor reports the highest value between the two lines that it fits.

NumEdgePoints: number of data points collected (the number of intersections between scanning lines and part edges). Only one data point per line is included in this value.

EdgePoint.X[]: array containing the absolute x-coordinate of the data points used for the line fitting (resulting from intersection of scan lines and part edges) of the first line (the one closer to the origin of the SoftSensor. It contains “NumEdgePoints” cells.

EdgePoint.Y[]:array containing the absolute y-coordinate of the data points used for the line fitting (resulting from intersections of scan lines and part edges) of the first line (the one closer to the origin of the SoftSensor. It contains “NumEdgePoints” cells.

EdgePoint2.X[]: array containing the absolute x-coordinate of the data points used for the line fitting (resulting from intersection of scan lines and part edges) of the second line (the one farther away from the origin of the SoftSensor. It contains “NumEdgePoints” cells.

EdgePoint2.Y[]:array containing the absolute y-coordinate of the data points used for the line fitting (resulting from intersections of scan lines and part edges) of the second line (the one farther away from the origin of the SoftSensor. It contains “NumEdgePoints” cells.

For Find Circle:

Position.X: integer value indicating absolute x-coordinate of the center of the circle found. Used for position reference.

Position. Y: integer value indicating absolute y-coordinate of the circle found. Used for position reference.

Point.X: floating point value indicating absolute x-coordinate of the center of the circle found. This number reports sub pixel accuracy.

Point.Y: floating point value indicating absolute y-coordinate of the center of the circle found. This number reports sub pixel accuracy.

Roundness: reports the distance in pixels, between the two data points that are farthest away from the output circle in each direction. If the edge is perfectly straight, this value should be close to zero. As the edge becomes noisier, the value increases.

NumEdgePoints: number of data points collected (the number of intersections between scanning lines and part edges).

EdgePoint.X[]: array containing the absolute x-coordinate of the data points used for the circle fitting (resulting from intersection of scan lines and part edges). It contains “NumEdgePoints” cells.

EdgePoint.Y[]:array containing the absolute y-coordinate of the data points used for the circle fitting (resulting from intersections of scan lines and part edges). It contains “NumEdgePoints” cells.

Math Tools

Math tools offer an assortment of SoftSensors that vary a lot in their functions. Thus, the sets of parameters available are very different as well. Except for the Result and intensity levels parameters, common to all of them, they contain specific parameters. The parameters are:

Result: overall result of the inspection of the part as an integer value (0 for PASS, positive for WARN, and negative for FAIL)

Distance: only applicable to the Distance SoftSensor. Reports the distance calculated according to user parameters.

Angle: only applicable to the Angle SoftSensor and SoftSensors that report lines (lines are reported as a point and an angle). Reports the angle calculated according to user parameters (could be the slope of the line). For the Angle SoftSensor it reports the value of the angle between the specified lines. For SoftSensors that report a line as their output (Midline, Line Through two points and Perpendicular) it reports the angle using standard sign from -180 to + 180 degrees: positive angles above the x-axis, and negative angles below the x-axis.

Scale: only applicable to the Scale Factor SoftSensor. Reports the scale factor calculated according to user parameters.

Point.X: only applicable to SoftSensors that have points or lines as their outputs. Represents the sub pixel absolute x-coordinate of the point being reported (lines are reported as a point and an angle)

Point.Y: only applicable to SoftSensors that have points or lines as their outputs. Represents the sub pixel absolute y-coordinate of the point being reported (lines are reported as a point and an angle)

TransformedPoint.X: only applies to SoftSensors that report single points (Midpoint, Intersection, and Coordinate Transformation). Contains the transformed sub pixel absolute X-coordinates of the point (when a coordinate transformation is applied).

TransformedPoint.Y: only applies to SoftSensors that report single points (Midpoint, Intersection, and Coordinate Transformation). Contains the transformed sub pixel absolute Y-coordinates of the point (when a coordinate transformation is applied).

Readers

Reader SoftSensors, which include 1D barcode readers, 2D code readers, and OCR offer the following parameters:

Result: overall result of the inspection of the part as an integer value (0 for PASS, positive for WARN, and negative for FAIL)

Contrast: indicates the contrast found in the area analyzed by the SoftSensor (in levels from 0 to 255). Only available for barcode reader and DataMatrix SoftSensors.

NumErrors: exclusive of 1D and 2D readers. Indicates the number of errors detected in the code being read and requires that the code being read contains some type of error checking method.

String: refers to the data extracted from the image. This filed exposes a string which is populated with the data resulting from decoding the image.

Exclusive of 1D Reader:

MinIntensity: lowest intensity value found in the area analyzed by the SoftSensor (in levels from 0 to 255).

MaxIntensity: highest intensity value found in the area analyzed by the SoftSensor (in levels from 0 to 255).

Threshold: threshold level (in levels from 0 to 255).

Exclusive of DataMatrix:

AxialNonuniformity: indicates how “stretched” the matrix is. This is a standard DataMatrix quality measurement.

Position.X: indicates the absolute X-coordinate of the corner of the DataMatrix where the solid sides meet.

Position.Y: indicates the absolute Y-coordinate of the corner of the DataMatrix where the solid sides meet.

Position.ThetaX: indicates the rotation associated with the DataMatrix found. See formula below for an explanation on how to interpret it.

Position.ThetaY: indicates the rotation associated with the DataMatrix found. See formula below for an explanation on how to interpret it.

$$\theta = ArcCos\left(\frac{Position.ThetaX}{16384}\right) = ArcSin\left(\frac{Position.ThetaY}{16384}\right)$$

PrintGrowth: refers to the variation of the size of the dark cells for the same DataMatrix because of changes in the printing process.

UnusedErrorCorrection: refers to the available “room for error” in the current image.

SymbolContrast: refers to the contrast found in the area where the DataMatrix is present. The difference in intensity between dark and light cells.

Exclusive of OCR:

NumChars: number of characters being read.

CharacterPosition.X[]: array containing the absolute x-coordinate for each character being read. The lower left corner of the character is used.

CharacterPosition.Y[]: array containing the absolute x-coordinate for each character being read. The lower left corner of the character is used.

CharacterThreshold[]: array containing the threshold levels used for each character.

CharacterScore[]: array containing the individual character scores (which refers to how well the characters in the current image match the learned characters).

CharacterDistance[]: array containing the distance from the origin of the SoftSensor to the location of each character. Used to detect presence of spaces in the code.

Blob Tools

Blob tools consist of two SoftSensors: blob generator and blob selector. Each one has its own parameters as shown below, but the important thing to remember is that since they work with a number of blobs, they contain mostly arrays as their parameters. Each entry in the array refers to a different blob, so the size of these arrays is the number of blobs present. Notice that all the array variables contain the brackets at the end of the name. That means that besides using the parameter the user needs to specify an index (blob number).

Result: overall result of the inspection of the part as an integer value (0 for PASS, positive for WARN, and negative for FAIL)

NumBlobs: indicates how many blobs the SoftSensor found. This number sets the size of the arrays containing information about the blobs.

Exclusive of the blob generator:

MinIntensity: lowest intensity value found in the area analyzed by the SoftSensor (in levels from 0 to 255).

MaxIntensity: highest intensity value found in the area analyzed by the SoftSensor (in levels from 0 to 255).

Threshold: threshold level (in levels from 0 to 255).

Contrast: indicates the contrast found in the area analyzed by the SoftSensor (in levels from 0 to 255).

Exclusive of the blob selector:

BlobAngle[]:this value is calculated only when Calculate Blob Angles is checked in the Blob Selector Parameters Tab. There are four options to obtain the angle:

```
//get the angle as formatted using the Blob Parameters tab
//of the blob selector
softsensorname.BlobAngle[n];

// get the angle of the blob’s principal axis using the
//0 to 180 degree format
softsensorname.BlobAngle.PA180[n];

// get the angle of the blob’s principal axis using the
// 0 to 360 degree format. To establish a direction
//here, the algorithm uses mass distribution.
softsensorname.BlobAngle.PA360[n];

// get the angle that the line through the centroid and
//maximum point of the blob makes with the direction
// of the positive x axis
softsensorname.BlobAngle.MaxPoint[n];
```

BlobArea[]: array containing the area of all the blobs found

BlobPosition.X[]:array containing the absolute x-coordinates of the center of the blobs found by the SoftSensor. Used for position reference.

BlobPosition.Y[]:array containing the absolute y-coordinates of the center of the blobs found by the SoftSensor. Used for position reference.

BlobBoundingBox: set of arrays containing information about the bounding box for each one of the blobs found. The user can even access the information for the bounding box as follows:

```
//n is a number between 1 and the maximum number of blobs
SoftSensor.BlobBoundingBox.Width[n]
SoftSensor.BlobBoundingBox.Height[n]
//upper-left corner x point of box
SoftSensor.BlobBoundingBox.X0[n]
//upper-left corner y point of box
SoftSensor.BlobBoundingBox.Y0[n]
//lower-right corner x point of box
SoftSensor.BlobBoundingBox.X1[n]
//lower-right corner y point of box
SoftSensor.BlobBoundingBox.Y1[n]
```

BlobCompactness[]: array containing the compactness of every blob found by the SoftSensor.

BlobEccentricity[]:array containing the compactness of every blob found by the SoftSensor.

BlobIntensity[]:array containing the compactness of every blob found by the SoftSensor. Intensity values are scaled between 0 and 255.

BlobMaxPoint.X[]:array containing the absolute x-coordinate of the point that is farthest away from the centroid of the blob. One cell for each blob.

BlobMaxPoint.Y[]:array containing the absolute y-coordinate of the point that is farthest away from the centroid of the blob. One cell for each blob.

BlobPerimeter[]:array containing the perimeter of every blob found by the SoftSensor.

BlobRadius[]:array containing the radius of every blob found by the SoftSensor.

BlobTransformedPoint.X[]:array containing the absolute x-coordinate of the blobs found by the SoftSensor corrected with a coordinate transformation.

BlobTransformedPoint.Y[]:array containing the absolute y-coordinate of the blobs found by the SoftSensor corrected with a coordinate transformation.

TemplateMatch SoftSensor

Result: overall result of the inspection of the part as an integer value (0 for PASS, positive for WARN, and negative for FAIL)

MinIntensity: Lowest intensity value found in the area analyzed by the SoftSensor (in levels from 0 to 255).

MaxIntensity: Highest intensity value found in the area analyzed by the SoftSensor (in levels from 0 to 255).

Threshold: Threshold level (in levels from 0 to 255).

Contrast: Indicates the contrast found in the area analyzed by the SoftSensor (in levels from 0 to 255).

Position.X: absolute x-coordinate of the location of the center of the template found, point passed as a position reference by this SoftSensor. This parameter is only applicable to the SoftSensor that searches for the template.

Position.Y: absolute y-coordinate of the location of the center of the template found, point passed as a position reference by this SoftSensor. This parameter is only applicable to the SoftSensor that searches for the template.

TransformedPoint.X: contains the transformed absolute x-coordinates of the position (when a coordinate transformation is applied). This parameter is only applicable to the SoftSensor that searches for the template.

TransformedPoint.Y: contains the transformed absolute y-coordinates of the position (when a coordinate transformation is applied). This parameter is only applicable to the SoftSensor that searches for the template.

PixelError: number of pixels in the SoftSensor that report an error in the image.

PercentPixelError: percentage of the pixels in the SoftSensor that report an error in the image.

MaxPixelError: number of pixels (in the segment that contains the most error) that report an error in the image.

MaxPercentError: percentage of pixels (in the segment that contains the most error) that report an error in the image.

ObjectFind SoftSensor

Result: overall result of the inspection of the part as an integer value (0 for PASS, positive for WARN, and negative for FAIL)

NumObjects: number of Objects found by an ObjectFind SoftSensor as an integer value.

Point.X: absolute x-coordinate of the object being found. This option is used for cases where a single object is being located.

Point.Y: absolute y-coordinate of the object being found. This option is used for cases where a single object is being located.

ObjectPosition.X[]:array containing the absolute x-coordinates of the objects being found. This option is used both when single and multiple objects are being located. Every cell in the array corresponds to a different object; for single objects only the first cell is used (index 1).responds to a different object.

ObjectPosition.Y[]:array containing the absolute y-coordinates of the objects being found. This option is used both when single and multiple objects are being located. Every cell in the array corresponds to a different object; for single objects only the first cell is used (index 1).responds to a different object.

ObjectAngle[]:array containing the angle by which the objects are rotated respect to the learned position. This option is used both when single or multiple objects are being located, for single object simply use index 1 of the array. Every cell in the array corresponds to a different object. The angle is based on what the position of the object as learned (that represents the zero degree angle). Angle formats are given by 0 to 180 degrees starting from the positive x-axis. Clockwise direction gives a positive angle, counterclockwise gives a negative one.

ObjectTransformedPoint.X[]:array containing the transformed x-coordinates of the objects being found. This option is used both when single or multiple objects are being located, for single object simply use index 1 of the array. Every cell in the array corresponds to a different object. A coordinate transformation SoftSensor is needed.

ObjectTransformedPoint.Y[]:array containing the transformed y-coordinates of the objects being found. This option is used both when single or multiple objects are being located, for single object simply use index 1 of the array. Every cell in the array corresponds to a different object. A coordinate transformation SoftSensor is needed.

NumEdges: number of edges of the largest object in the Object Find’s region (not necessarily the learned object). Use this in conjunction with EdgePoint.X[] and EdgePoint.Y[].

ObjectEdgePoint.X[]: array containing the sub pixel x-coordinates of edges found by an ObjectFind SoftSensor.

ObjectEdgePoint.Y[]: array containing the sub pixel y-coordinates of edges found by an ObjectFind SoftSensor.

Note: the EdgePoint parameter does not refer to the learned or found object, but to the largest object in the area analyzed by the SoftSensor.

Pixel Counting SoftSensor

Result: overall result of the inspection of the part as an integer value (0 for PASS, positive for WARN, and negative for FAIL)

MatchPixelCount: indicates the number of pixels (of the best color match) present in the image. Equivalent to the DominantColorPixels entry in the result table for this SoftSensor.

MatchIndex: indicates the index of the best matched color. Useful when working with lists of colors.

MatchColorName: indicates the name of the best matched color. Useful when working with lists of colors.

PixelPercent: percentage of the pixels in the SoftSensor that are recognized as pixels of any of the colors in the list of learned colors.

PixelCount: number of the pixels in the SoftSensor that are recognized as pixels of any of the colors in the list of learned colors.

Color Monitoring SoftSensor

Result: overall result of the inspection of the part as an integer value (0 for PASS, positive for WARN, and negative for FAIL)

MatchScore: provides the overall color difference scaled from 0 to 10,000.

MatchIndex: indicates the index of the best matched color. Useful when working with lists of colors.

MatchColorName: indicates the name of the best matched color. Useful when working with lists of colors.

ColorDeltaX: indicates the color difference found in every channel. X could be 1, 2, or 3. For example, when using RGB space channel 1 is for Red, channel 2 is for Green, and channel 3 is for Blue. This value is scaled up by 100, that is, for a color difference in red of -2.56, this field would report -256. When a Monochrome system is being used, this parameter changes to IntensityDelta.

LiveMeanX: indicates the mean value for a certain channel, X could be Red, Green, Blue, _L, _a, _b, _C, and _H depending on the color domain and the channel selected. This value refers to the image being analyzed. This value is scaled up by 100, that is, for a mean of 25.83, this field would report 2583. When a Monochrome system is being used, this parameter changes to LiveMeanIntensity.

LiveStdDevX: where X could be Red, Green, or Blue, indicates the standard deviation of the values in each channel. This value is scaled up by 100, that is, for a standard deviation of 6.55 this field would report 655. When a Monochrome system is being used, this parameter changes to LiveStdDevIntensity.

GoldenMeanX[]: where X could be Red, Green, Blue, _L, _a, _b, _C, or _H, represents an array indicating the mean values of the learned colors. The array contains one cell for every color in the list. For example, to get the mean value of Hue for color number 5, the command should be `sensorName.GoldenMean_H[5];.` This value is scaled up by 100, that is, for a mean of 25.83, this field would report 2583. When a Monochrome system is being used, this parameter changes to GoldenMeanIntensity.

GoldenStdDevX[]:where X could be Red, Green, or Blue, represents an array indicating the values of the standard deviation for the learned colors. The array contains one cell for every color in the list. This value is scaled up by 100, that is, for a standard deviation of 6.55 this field would report 655. When a Monochrome system is being used, this parameter changes to GoldenStdDevIntensity.

Segmentation SoftSensor

Result: overall result of the inspection of the part as an integer value (0 for PASS, positive for WARN, and negative for FAIL)

NumSegments: returns the number of segments found in the image.

LiveSegmentPosition.X[]: array containing the absolute x-coordinates of the center of the segments found in the image.

LiveSegmentPosition.Y[]: array containing the absolute y-coordinates of the center of the segments found in the image.

LiveAveX[]: where X can be Red, Green, or Blue, refers to the average content of Red, Green or Blue present in a certain segment. Each array has one cell for every segment found in the image. The value reported is the actual value is scaled up by 100.

LivePixelCount[]: array containing the pixel count for all the segments found in the image.

Script SoftSensor (Foreground Script)

Script SoftSensors (or Foreground Scripts) have a number of parameters as well. However, these parameters can be altered from the script. That is, the user can access this set of parameters from another script (just like any SoftSensor) but it is possible to change the parameters of the Script being edited. The parameters are:

Result: the user can read (from another script) or set (from current script) the result (PASS, WARN, or FAIL). This is useful when the application requires complex decisions, a script could make the decision and pass or fail to indicate the overall result.

Note: The Script will not fail if it references a SoftSensor that has failed. If you wish to have the Script SoftSensor fail when it references a failed SoftSensor, ensure that the following line is included in the beginning of your script:

```
FailOnFailingReference = true;
```

Intensity: this field is reserved for an integer value. Any integer value could be assigned to this field.

Line: set of four parameters: Line.A, Line.B, Line.X, and Line.Y. The equation for a line in this format is Ax + By = 0. The parameters A and B determine the slope while the parameters X and Y determine a point in the line.

Position : set of parameters that can be referenced by other SoftSensors through the "Enable Position Reference" checkbox. Position consists of four separate parameters: Position.X, Position.Y, Position.ThetaX, and Position.ThetaY. These parameters are used to change the position of a script, and they are used as follows;

```
//Script Positioning: directly assign integer values or variables
//to the x and y parameters
this.Position.X=0;
this.Position.Y=0;
//for the angle (reference of rotation) it requires more work
//first declare a float variable for the angle in radians
float radangle;
int degangle=0;
//convert it to degrees if needed
radangle = degangle*3.14159/180;
//load the parameters as follows
//the output angle is scaled by 16384
this.Position.ThetaX=cos(radangle)*16384;
this.Position.ThetaY=sin(radangle)*16384;
//When all these parameters are set and another SoftSensor
//references the Foreground Script for position, it will
//translate and rotate as indicated by the script parameters
```

Distance : This field is reserved for a floating point value. Any floating point value could be assigned to this field.

Angle : This field is reserved for a floating point value. Any floating point value could be assigned to this field.

Point : a set of floating point parameters. Point consists of two parameters: Point.X and Point.Y representing the coordinates of a point in the image.

String: a string data-type of maximum length 200 characters. Any string of less than 200 characters could be assigned to this output parameter.

DVal[]: array of double variables used only in a script SoftSensor. The DVal array can be used to store double values which can then be used with Data Link. Before using it, the size of the array must be defined. This method was a standard procedure in older systems. In newer systems users are encouraged to use a String instead of this array.

IVal[]:array of integer variables used only in a script SoftSensor. The IVal array can be used to store integer values which can then be used with Data Link. Before using it, the size of the array must be defined. This method was a standard procedure in older systems. In newer systems users are encouraged to use a String instead of this array.

Note: Once the Script SoftSensor (or Foreground Script) outputs have been assigned, the user needs to select which one(s) to display to the result table by adding a checkmark to the corresponding outputs in the Outputs tab of the SoftSensor parameter dialog. Only those selected in that tab will be displayed on the result table.

Spectrograph SoftSensor

MatchScore: is the Spectrum Difference.

MatchIndex: the index number of the closest spectrum match.

MatchSpectrumName: the name of the closest spectrum match.

LivePeakIntensity: highest intensity in the spectrograph.

LivePeakLocation: the location of LivePeakIntensity in nm.

PeakIntensityChange: the intensity difference between the learned peak intensity and LivePeakIntensity.

PeakLocationChange: the change in location between the learned peak intensity and LivePeakLocation in nm.

LiveAverage[]: is used to get the intensity of spectrum at a particular wavelength. Resultant spectrum is computed by averaging. LiveAverage[500] returns the intensity of spectrum at 500nm if available, and an error otherwise.

LiveContrast[]: gets the contrast (Max – Min) at a particular wavelength.

LiveMaximum[]: gets the maximum at a particular wavelength.

LiveMedian[]: gets the median at a particular wavelength.

LiveMinimum[]: gets the minimum at a particular wavelength.

LivePeak[]: gets the peak at a particular wavelength.

LiveStdDev[]: gets the standard deviation at a particular wavelength.

Note: The absolute color values of the resultant live spectrum can be read by: LiveMean_X, LiveMean_Y, LiveMean_Z, LiveMean_L, LiveMean_a, LiveMean_b, LiveMean_C, LiveMean_H.

Note: The absolute color values of learned spectrums can also be read. The index to the list is zero based. Here is the list: GoldenMean_X[], GoldenMean_Y[], GoldenMean_Z[], GoldenMean_L[], GoldenMean_a[], GoldenMean_b[], GoldenMean_C[], GoldenMean_H[].

Advanced Functionality: Sensor Object

All the parameters specified so far in this chapter are readily available from the script editor just one click away. These parameters represent data that the user can access from the SoftSensors to perform more complex decisions. There is another way to access the SoftSensor from Foreground Scripts that allows the user to modify SoftSensor parameters: the use of Sensor Objects. Sensor Objects are entities with a predefined functionality. When one of those entities are created and assigned to an existing SoftSensor, the functionality can be used to manipulate parameters in that SoftSensor. When using a Sensor Object, some advanced functions can be used. Those functions are explained in this section.

GetSensorByName()

The GetSensorByName Function is used to set a Sensor Object using the sensor name. Use of this function is necessary when using any sensor script methods.

Syntax

```
S=GetSensorByName(string Name);
```

Arguments

A string containing the name of the desired SoftSensor.

Returns

A Sensor object associated with the Sensor Name.

Example

```
//create a Sensor Object and assign SoftSensor “OCRReader” to it
Sensor S;
S=GetSensorByName("OCRReader");
```

GetSensorById()

This function is used to initialize a Sensor Object by referencing the internal sensor ID of the SoftSensor. These IDs can be determined by issuing a sensor query command (#SQ) for a product via the system terminal. The GetSensorByName function is often used instead of this function.

Syntax

```
<SensorObjectName>.GetSensorById(int id);
```

Arguments:

An integer value indicating the internal ID of the SoftSensor to be assigned to the Sensor Object

Return values

The function returns a reference to the Sensor Object (the Sensor handle)

Error Codes

-KS_IVALID_PARAM: (-2) invalid bit position specified for the virtual input.
-KS_TIMEOUT (-4) Wait operation timed out.

Examples:

```
//The following script initializes the Sensor Object
Sensor S;
S=GetSensorById(1);
```

SetMatchString()

The SetMatchString() Method is used to set a Sensor match string for a Reader SoftSensor.

Syntax

```
<SensorObjectName>.SetMatchString(String NewString);
```

Arguments:

A String or String variable containing the new string to be matched by the SoftSensor.

Return values

None

Examples:

```
// SetMatchString for an OCR SoftSensor
Sensor S;
S = GetSensorByName("OCRReader");
S.SetMatchString("ABCD1234");
```

SetParams()

The SetParams() method is used to set parameters associated with a SoftSensor. This method can be useful for setting the Shape of a SoftSensor or Importing a dominant color into a SoftSensor.

Syntax

```
<Sensor Object Name>.SetParams(String Command);
```

Arguments:

A string literal or string variable containing the command to set the SoftSensor parameters. These commands are derived from the commands list (this list is included in the FrameWork help files and is available for free download from the DVT website). The string consists of the parameter ID and the data for that parameter.

Return values

An `int` value indicating the result of the operation. Result=0 for success, Result<0 for error.

Examples:

```
//The following script sets the shape of a blob SoftSensor
//to a rectangle with an upper left corner 100,100 and
//lower right corner 200,200
Sensor Blobs;
Blobs = GetSensorByName("BlobGen");
Blobs.SetParams("0 10 100 100 200 200");
```

This example changes the shape of the SoftSensor, so the String contains a zero (parameter ID for sensor shape according to the commands list) and the data. The data consists of the shape of the SoftSensor which is an area rectangle (according to commands list it is represented by a 10) and the specific data for that shape which according to the commands list is the x and y coordinates of the upper left corner of the rectangle followed by the x and y coordinates of the lower right corner of it. All the parameters are separated by a space. The resulting string is then “0 10 100 100 200 200” for a rectangle form pixel (100,100) to pixel (200,200).

```
//this scripts makes a color sensor learn new colors
Sensor CSensor;//declare the sensor object
```

```
//assign the existing “colorArray” softsensor to the object
CSensor = GetSensorByName("colorArray");
//check if the assignment was successful, if it is not,
//the reference to the object will contain a null character
//so we need to verify that the reference is not null
if(CSensor != null)
{
      DebugPrint("Successful assignment");//for debugging
      if(RegisterReadByte(200) == 0)
      {
            //if a certain flag is set to 0 execute this command
            CSensor.SetParams("45");
      }
      else if(RegisterReadByte(200) == 1)
      {
      //if the flag is set to 1 execute this command
            CSensor.SetParams("44");
      }
}
```

This example involves a Pixel Counting SoftSensor called “colorArray.” The idea here is that the pixel counting SoftSensor has to learn a single color or many colors at the same time based on the status of a flag. If the register 200 is set to 0 learn a single color, if it is set to 1 learn many colors, otherwise do nothing. The strings used are “45” to learn a single color and “44” to learn many colors.

Stats()

This command populates an array of integer values indicating the times of the inspection.

Syntax

```
<Sensor Object Name>.Stats(int statsArray[]);
```

Arguments:

statsArray: a four element integer array

Return values

This command populates the array as follows:

element 1 is the minimum inspection time

element 2 is the maximum inspection time

element 3 is the average inspection time

element 4 is the last inspection time

Examples:

```
//declare array to hold values
int stats[]=new int[4];
//declare sensor object
Sensor s;
String str;
//assign sensor
s=GetSensorByName("colorArray");

if (s!=null)
{
      //get stats from sensor if not null
      s.Stats(stats);
      //set string to display in result table
      str = stats[1]+"/"+stats[2]+"/"+stats[3]+"/"+stats[4];
      this.String = str;
}
```
This example makes the script’s string equal to a sequence of four integer values separated by a forward slash character. Those four numbers represent the times from the inspection.

Chapter 5 - Accessing Product Data

This chapter describes how to get data from Products and how to change a limited number of Product Parameters. In many cases, users need to change parameters such as exposure time and product gain, which are product specific. Background Scripts can be used to extract the necessary data by using a combination of commands and syntax. As explained before, this type of Script reside at the system level thus gaining access to all the products in the SmartImage Sensor. Both the commands needed and the syntax used are explained in this chapter.

Syntax for Basic Product Data Extraction

In order to gain access to Product data from scripts, users simply need to create a variable and assign the desired data from the desired Product to it. To reference the individual Product parameters, the user needs to use the following syntax:

```
<Product name>.<Product parameter>
```

Let us illustrate this with an example. We have a particular Product from which we need to change the exposure time. A particular Foreground Script is reading the data from an intensity SoftSensor in that product and saving that value as a `float` data type to a particular DVT register. The Background Script then needs to read the value in that register and alter the exposure time. It will increase the exposure time if the intensity read is too low, it will decrease it if the intensity level is too high, and it will not change it if the intensity read is at an acceptable level. Assuming that the value is written to register 20 and that the product name is Product1, the syntax would be as shown below:

```
float minIntensity = 70, maxIntensity = 90;
float intensityLevel;
while(true)
{
   //first the value is read
   intensityLevel = RegisterReadFloat(20);
   //next, it is compared to the minimum allowed
   if(intensityLevel < minIntensity)
   {
      //change exposure time if necessary
      Product1.SetExposure( Product1.GetExposure() * 1.2);
   }
      //compare it to maximum if it was higher than minimum
   else if(intensityLevel > maxIntensity)
      {
      //change exposure time if necessary
         Product1.SetExposure(Product1.GetExposure() * 0.8);
      }
   sleep(3000);//pauses execution for 3000 msec
}
```

In this case the Background Script is declaring the necessary variables and then entering a loop which gets executed every 3000 milliseconds (note the sleep command). Every three seconds the Background Script reads the intensity level from a memory register (a Foreground Script should write it to memory, remember that Foreground Scripts are the only scripts that can access SoftSensor data). Once the value is read, it is compared to the arbitrary levels of minimum intensity and maximum intensity. Depending on the result of these comparisons, the exposure time is incremented or decremented by 20%. The command used to perform these changes is a combination of two commands: `GetExposure()` and `SetExposure()`. The first one accesses the value and the second one applies changes to it. The available commands are explained next.

SetWindow()

This command is used to change the partial window of acquisition. By using this command the Background Script can change the window for a particular product as needed.

Syntax

```
Product1.SetWindow(int x0, int y0, int x1, int y1);
```

Arguments:

Four integer values specifying the coordinates of the upper left corner of the window (x0, y0) and the lower right corner of the window (x1, y1)

Example:

```
int x0, x1, y0, y1;
x0 = 15;
y0 = 15;
x1 = 625;
y1 = 465;
Product1.SetWindow(x0, y0, x1, y1);
```

Notes

The coordinates are in pixels, so for a 640 by 480 resolution SmartImage Sensor the ranges are 0 to 639 for x and 0 to 479 for y.

GetExposure()

This command is used to obtain the exposure from a product

Syntax

```
Product1.GetExposure();
```

Arguments:

None

Return values

The exposure time as an integer value in microseconds.

Example:

```
int expTime;
expTime = Product1.GetExposure();
```

Notes

This function uses returns the exposure time in microseconds whereas the user interface displays it in milliseconds.

SetExposure()

This command is used to set the exposure from a product

Syntax

```
Product1.SetExposure(int expTime);
```

Arguments:

The new exposure time for the product as an integer value (in microseconds).

Examples:

```
//two ways to increase the exposure time by 20%
//first method: the long way
int expTime;
expTime = Product1.GetExposure();//obtain current value
expTime = expTime * 1.2; //update value
Product1.SetExposure(expTime);//set new value
//second method: shorter but more complex
//this method does everything in one line and avoids the
//use of extra variables.
Product1.SetExposure(Product1.GetExposure() * 1.2);
```

Notes

This function returns the exposure time in microseconds whereas the user interface displays it in milliseconds.

GetGain()

This function is used to access the value set for the gain in a product.

Syntax

```
float f = Product1.GetGain();
```

Arguments:

None

Return values

The product gain as a float.

Example:

```
float f;
f = Product1.GetGain();
```

Notes

The values for the gain range from 1 to 25, the default value is 2

SetGain()

This function is used to modify the value set for the gain in a product.

Syntax

```
Product1.SetGain(float newGain);
```

Arguments:

A `float` data type variable containing the new value for the gain

Examples:

```
//Example 1
float newGain = 3.3;
Product1.SetGain(newGain);
//Example 2: increase current gain by 20%
float newGain;
newGain = Product1.GetGain() * 1.2;
Product1.SetGain(newGain);
```

Notes

The values for the gain range from 1 to 25, the default value is 2

Advanced Functionality: Product Object

All the parameters specified so far in this chapter are readily available from the script editor just one click away. These parameters represent the basic functionality of the products. There is another way to access Products from Background Scripts: the use of Product Objects. Product Objects are entities with a predefined functionality. When one of those entities is created and assigned to an existing Product, more advanced functionality becomes available: the ability to access the inspection product (without knowing which one it is), the ability to loop through all the products in the SmartImage Sensor, and the ability to manipulate references to a product without using that product’s name. That is, all the functions explained so far can be called from this

Product Object, but other functions become available as well. Those advanced functions are explained in this section.

GetProduct()

The GetProduct() function is used to set a Product object using the product name.

Syntax

```
GetProduct(string ProdName);
```

Arguments:

A string containing the product name.

Examples:

```
// increase product exposure time by 10%
Product P;
P=GetProduct("My Product");
P.SetExposure(P.GetExposure()*1.10);
```

Notes

As the example illustrates, the basic functions (such as GetExposure()) can still be used when working with Product Objects

GetProductById()

The GetProductByID() function is used to set a Product variable using the digital ID set in the Product Management dialog in FrameWork.

Syntax

```
GetProductById(int prodID);
```

Arguments:

Integer value containing the product digital ID

Return values

The product with the corresponding digital ID

Examples:

```
// increase product exposure time by 10%
Product P;
P=GetProductById(3);
P.SetExposure(P.GetExposure()*1.10);
```

GetInspectProduct()

The GetInspectProduct() function is used to set a Product based on the current inspection product, which can be externally changed using product selection.

Syntax

```
MyProduct = GetInspectProduct();
```

Arguments:

None

Return values

The current inspection product

Examples:

```
// increase product exposure time by 10%
Product P;
P = GetInspectProduct();
P.SetExposure(P.GetExposure()*1.10);
```

GetFirstProduct()

The GetFirstProduct() function is used in combination with the Next() method to loop through the products in a SmartImage Sensor.

Syntax

```
MyProduct = GetFirstProduct();
```

Arguments:

None

Return values

The first product that was created in or loaded to the SmartImage Sensor.

Examples:

```
//see the Next() function for a complete example
Product P;
P=GetProduct("My Product");
```

Next()

The Product Next method is used to change a Product variable to the next DVT product in the SmartImage sensor. Note: the next product is determined by the order in which the products were created in or loaded to the SmartImage Sensor. This function is often used to change product parameters in multiple products using a while loop.

Syntax

```
MyProduct.Next();
```

Return values

The next product according to the order in which they were added to the system

Example:

```
//increase the gain in all products by 20%
Product P;
P=GetFirstProduct();
while(P!=null)
{
      P.SetGain(P.GetGain() * 1.2);
      P = P.Next();
      sleep(150);
}
```

ID()

The Product ID method returns the digital product ID of the associated product as set in the Product Management Dialog.

Syntax

```
MyProduct.ID();
```

Arguments:

None

Return values

Digital Product ID of the associated product as an Integer.

Examples:

```
// Put the digial ID of a product named Product1
// in a register
Product P;
P = GetFirstProduct();
RegisterWriteInteger(20,P.ID());
```

Select()

This command is used to set a certain product as the inspection product.

Syntax

```
MyProduct.Select();
```

Arguments:

None

Example:

```
//this example reads a boolean variable and sets the inspection
//product accordingly
Product MyProduct;
boolean useProduct1 = (RegisterReadByte(100) == 1);

if(useProduct1 == true)
{
      Product1.Select();
}
else
{
      Product2.Select();
}
```

Inspect()

The Inspect function triggers the SmartImage Sensor to take an inspection. The product Select method can be used to set the inspection product prior to inspection. Typically, the busy output bit should be monitored to make sure the inspection is completed before allowing the script to continue.

Syntax

```
Inspect();
```

Arguments:

None

Example:

```
//This example sets the inspection product, takes an
//inspection, then waits for the inspection to be finished.
Product P;
long Bit = 1;
```

```
//Select the product, in this case the system must contain a
//product called "Product 1"
P = GetProduct("Longo");
P.Select();  //Set inspection product

//record the time before the inspection
int time = clock();

Inspect();  //trigger inspection

//wait for inspection to finish by monitoring the busy output
while ((GetOutputs() & (Bit<<3) )!=0) ;

time= clock() - time;

//Output the inspection time
DebugPrint("InspectionTime = " + time);
```

Stats()

This command populates an array of 4 integer values with statistical data about the inspections.

Syntax

```
<Product Object Name>.Stats(int statsArray[]);
```

Arguments:

statsArray: a four element integer array

Return values

This command populates the array as follows:

Element 1 is the last inspection result (-1 means PASS, 0 means WARN, and 16 means FAIL). If the product contains no SoftSensors and the SmartImage Sensor is not running inspections, a -2 will be assigned to this element.

Element 2 is the non-failed inspection count, or PASS + WARN. It keeps count of the inspections that produced a WARN or a PASS as the overall result.

Element 3 is the WARN count. It keeps count of the inspections that produced a WARN as the overall result.

Element 4 is the FAIL count. It keeps count of the inspections that produced a FAIL as the overall result.

Examples:

```
//declare array to hold values
int stats[]=new int[4];
//declare product object
Product P;
String str;
while(true)
{
      //assign product
      P = GetInspectProduct();
      if (P != null)
      {
            P = GetInspectProduct();
            P.Stats(stats);
            str = "Stats: ”+stats[1]+"/"+stats[2];
            str = str + "/"+stats[3]+"/"+stats[4];
            str = str + “ For Product ID: "+P.ID();
            DebugPrint(str);
      }
      sleep(5000);
}
```

This example of a Background Script prints the inspections statistics every 5 seconds. Notice that the configuration of the string could be done in a single line, it was done in 3 lines instead to make it easier to read from the manual.

Name

This is a field of the Product Object not a method. It returns the name of the product that the Product Object is referencing.

Syntax

```
MyProduct.Name;
```

Arguments:

None

Return values

String containing the name of the product

Examples:

```
Product P;
P = GetInspectProduct();
string ProdName = P.Name;//the string ProdName contains now the
                        //product name of the inspection product
```

Chapter 6 - Mathematical computations

This chapter discusses the functions and operators available for mathematical calculations and how to use them. Every function is explained and a quick example is included to better illustrate the functionality.

Mathematical Operators and Functions

One of the powerful features of DVT Scripts is the ability to perform mathematical computations. In order to simplify those, there are a number of functions and operators available to the user. The table below shows all of them.

OPERATORS	FUNCTIONS
=	pow(,)
+	abs()
-	sqrt()
*	sin()
/	cos()
--	tan()
++	asin()
+=	acos()
-=	atan()
*=	atan2()
/=	
%=	
&=	
|=	
^=	
<<=	
>>=	
>>>=	

The assignment operator (=)

The assign operator is used to assign values to variables; it works with all data types.

Syntax

```
x=y;
```

Arguments

Any data type

Return values

None

Example

```
int i, x, y;
double d;
String MyStr;
i=4;//assign value to integer
d=4.567;//assign value to double
```

```
MyStr=”123”//assign value to string
x=5; //assign value to integer
y=x;//assign value of variable to another variable
```

Notes

Use assign (=) to store one value in another variable. See the equal (==) operator to compare two values for a true condition.

The addition operator (+)

The addition operator works with integer and floating point numbers to produce the sum of the values. When using the addition operator on strings, the result is a concatenated string. When used in a string operation, the addition operator converts integer and floating point values to a string.

Syntax

```
x+y;
```

Arguments:

Two basic data types

Return values

The result of adding the numerical value of the variables a and b in the case where the arguments are basic data types. If one of the arguments is a string, the result is a new string resulting from concatenating the data.

Example

```
//Write DVT Corp using + operator
int j=10,x1,x2;
String myStr;
j=j+1;//j is 11 now
myStr="DVT" + " Corp";//DVT + Corp = DVT Corp
x1 = 8;
x2 = 9;
myStr="x1+x2 = " + (x1+x2);//the string is now “x1+x2 = 17”
```

The subtraction operator (-)

The subtraction operator works with both integers and doubles to produce the difference between two values.

Syntax

```
expression - expression2;
```

Arguments

Two variables of the same basic data type

Return values

The result of subtracting expression2 from expression

Example:

```
int i,j,k,m;
double d;
j=7;
k=4;
m=j-k;
```

```
k=3-4;
d=1.23 – 0.02;
```

The multiplication operator (*)

The multiplication operator works with both numerical types to obtain the product of two values.

Syntax

```
x*y;
```

Arguments

Two variables of the same numerical basic data type

Return values

The value of x times y

Examples:

```
double area, r =4.5, rSquared;
rSquared = r*r; //result is I squared
area=3.1415*rSquared; //Area result is pi times the
                        //radius squared
```

The division operator (/)

The division operator works with both integers and doubles to produce the quotient of two values.

Syntax

```
x/y;
```

Arguments

Two variables of the same numerical basic data type

Return values

The value of x divided by y

Example

```
double d,R;
R=d/2.0;// radius of a circle
```

Notes

Division between two integers results in an integer.

The increment operator (++)

Used to increment variables. There are two ways you can use the increment operator: postfixed or prefixed. If the operator is postfixed, the variable is first used and then incremented; if the operator is prefixed, the variable is first incremented and then used.

Syntax

Postfix increment

```
variable1++;
```

Prefix increment

```
++variable1;
```

Arguments

An integer data type

Return values

Returns an integer value incremented by one (1)

Example:

```
int prefix_cnt = 5;
int postfix_cnt = 5;

this.Point.X = ++prefix_cnt; //point X will equal to 6
DebugPrint("" + prefix_cnt); //DebugPrint displays 6
this.Point.Y = postfix_cnt++; //point Y will equal to 5
DebugPrint("" + postfix_cnt); //DebugPrint displays 6
```

The decrement operator (--)

Used to decrement variables. There are two ways you can use the decrement operator: postfixed or prefixed. If the operator is postfixed, the variable is first used and then decremented; if the operator is prefixed, the variable is first decremented and then used.

Syntax

Postfix decrement

```
variable1--;
```

Prefix decrement

```
--variable1;
```

Arguments

An integer data type

Return values

Returns an integer value decremented by one (1)

Example:

```
int prefix_cnt = 5;
int postfix_cnt = 5;

this.Point.X = --prefix_cnt; //point X will equal to 4
DebugPrint("" + prefix_cnt); //DebugPrint displays 4
this.Point.Y = postfix_cnt--; //point Y will equal to 5
DebugPrint("" + postfix_cnt); //DebugPrint displays 4
```

The addition assignment operator (+=)

This operator uses infixed notation and is equivalent to op1 = op1 + op2.

Syntax

```
x += y;
```

Arguments

Two numerical values.

Return values

The value of x results from adding the numerical value of the variables x and y.

Example:

```
int x = 2;
int y = 3;

x += y;
this.String = x; //string equals 5;
```

The subtraction assignment operator (-=)

This operator uses infixed notation and is equivalent to op1 = op1 - op2.

Syntax

```
x -= y;
```

Arguments

Two numerical values.

Return values

The value of x results from subtracting the numerical variable y from x.

Example:

```
int x = 2;
int y = 3;

x -= y;
this.String = x; //string equals -1;
```

The multiplication assignment operator (*=)

This operator uses infixed notation and is equivalent to op1 = op1 * op2.

Syntax

```
x *= y;
```

Arguments

Two numerical values.

Return values

The value of x results in the product of x multiplied by y.

Example:

```
int x = 2;
int y = 3;

x *= y;
this.String = x; //string equals 10;
```

The division assignment operator (/=)

This operator uses infixed notation and is equivalent to op1 = op1 / op2.

Syntax

```
x /= y;
```

Arguments

Two numerical values.

Return values

The value of x results in the quotient of x divided by y.

Example:

```
float x = 2.0;
float y = 3.0;

x /= y;
this.String = x; //string equals 0.67;
```

The modulus assignment operator (%=)

This operator uses infixed notation and is equivalent to op1 = op1 % op2.

Syntax

```
x %= y;
```

Arguments

Two numerical values.

Return values

The value of x results in the modulus of x and y.

Example:

```
int x = 2;
int y = 3;

x %= y;
this.String = x; //x equals 2;
```

The bitwise AND assignment operator (&=)

This operator uses infixed notation and is equivalent to op1 = op1 & op2.

Syntax

```
x &= y;
```

Arguments

Two integer data types with the same number of bits.

Return values

Assigns x a value the same size as the arguments, with bit values corresponding to the result of a comparison between the bits in both arguments according to the truth table below:

argument1	argument2	result
0	0	0
0	1	0
1	0	0
1	1	1

Example:

```
int x = 2; //bit value of 10
int y = 3; //bit value of 11

x &= y;
this.String = x; //x equals bit value of 10 (decimal value of 2)
```

The bitwise OR assignment operator (|=)

This operator uses infixed notation and is equivalent to op1 = op1 **|** op2.

Syntax

```
x |= y;
```

Arguments

Two integer data types with the same number of bits

Return values

Assigns x a value the same size as the arguments, with bit values corresponding to the result of a comparison between the bits in both arguments according to the truth table below:

argument1	argument2	result
0	0	0
0	1	1
1	0	1
1	1	1

Example:

```
int x = 2; //bit value of 10
int y = 3; //bit value of 11

x |= y;
this.String = x; //x equals bit value of 11 (decimal value of 3)
```

The bitwise XOR assignment operator (^=)

This operator uses infixed notation and is equivalent to op1 = op1 ^ op2.

Syntax

```
x ^= y;
```

Arguments

Two integer data types with the same number of bits

Return values

Assigns x a value the same size as the arguments, with bit values corresponding to the result of a comparison between the bits in both arguments according to the truth table below:

argument1	argument2	result
0	0	0
0	1	1

1	0	1
1	1	0

Example:

```
int x = 2; //bit value of 10
int y = 3; //bit value of 11

x |= y;
this.String = x; //x equals bit value of 01 (decimal value of 1)
```

The signed left shift assignment operator (<<=)

This operator uses infixed notation and is equivalent to op1 = op1 << op2.

Syntax

```
x <<= y;
```

Arguments

Two integer data types

Return values

Assigns x a value of the same size as the argument with the bits shifted to the left a number of places equal to the value of y

Example:

```
byte x = 2; //bit value of 00000010
byte y = 3;

x <<= y;
this.String = x; //x equals bit value of 00010000 (decimal value
                 //of 16)
```

The signed right shift assignment operator (>>=)

This operator uses infixed notation and is equivalent to op1 = op1 >> op2.

Syntax

```
x >>= y;
```

Arguments

Two integer data types

Return values

Assigns x a value of the same size as the argument with the bits shifted to the right a number of places equal to the value of y

Example:

```
byte x = 8; //bit value of 00001000
byte y = 3;

x >>= y;
this.String = x; //x equals bit value of 000000001 (decimal value
                 //of 1)
```

The unsigned right shift assignment operator (>>>=)

This operator uses infixed notation and is equivalent to op1 = op1 >>> op2.

Syntax

```
x >>>= y;
```

Arguments

Two integer data types

Return values

Assigns x a value of the same size as the argument with the bits shifted to the right a number of places equal to the value of y.

Example:

```
byte x = 8; //bit value of 00001000
byte y = 3;

x >>= y;
this.String = x; //x equals bit value of 000000001 (decimal value
                 //of 1)
```

The power function

The pow() function is used to raise a number to a power.

Syntax

```
pow(double i, double j);
```

Arguments

Two numerical variables

Return values

The argument i raised to the j power as a double

Example:

```
Double i,j,k;
 i=4.56;
 j=2.65;
 k=pow(i,j); //result is 55.751535
```

The Absolute Value function

The absolute value function returns the absolute value of the argument.

Syntax

```
abs(double value);
```

Arguments

Any basic signed data type

Return values

The absolute value of the argument

Example:

```
x=abs(-2.6); //result is 2.6
```

```
x=abs(2.6); //result is 2.6 as well
```

The Square Root Function

The square root function works with both integers and doubles and returns a double value representing the square root of the argument.

Syntax

```
sqrt(double (or int) value);
```

Arguments

The argument value can be any number

Return values

The square root as a double

Example:

```
//computing the distance between point A (xa,ya), and B(xb,yb)
double dist;
float dx = xb – xa;
float dy = yb – ya;
dist = sqrt((dx*dx)+(dy*dy));
```

The Sine function

The sine function takes the angle in radians as a double and returns the sine of the angle

Syntax

```
sin(double angle);
```

Arguments

The argument angle is in radians.

Return values

The sine of the angle as a double

Example:

```
double theta, sinTheta;
theta = 45;//angle in degrees
sinTheta=sin(theta*3.1415/180); //convert degrees to radians
                                //result is 0.71
```

The Cosine function

The cosine function takes the angle in radians as a double and returns the cosine of the angle

Syntax:

```
cos(double angle);
```

Arguments

The argument angle in radians as a double

Return values

The cosine of the angle as a double.

Example:

```
double theta, cosTheta;
theta = 45;//angle in degrees
cosTheta=cos(theta*3.1415/180); //convert degrees to radians
                                //result is 0.71
```

The Tangent function

The tangent function takes the angle in radians as a double and returns the tangent of the angle.

Syntax

```
tan(double angle);
```

Arguments

The argument angle is in radians as a double

Return values

The tangent of the angle as a double number between –infinity and infinity

Example:

```
double theta, tanTheta;
theta = 45;//angle in degrees
tanTheta=tan(theta*3.1415/180); //convert degrees to radians
                                //result is 1
```

The ArcSine function

The arcsine function returns the arc sine or inverse sine of the specified value in radians. Multiply the returned value by 57.2974 (or 180/pi) to convert the returned value to degrees.

Syntax

```
asin(double value);
```

Arguments

The argument value is a double number between –1.0 and 1.0

Return values

The angle as a double in radians between –pi/2 and pi/2

Example

```
double theta, sinTheta;
sinTheta = 0.71;//sine of Theta
theta=asin(sinTheta);//theta should be ~ pi/4 or 45 degrees
```

The ArcCosine function

The arccosine function returns the arc cosine or inverse cosine of the specified value in radians. Multiply the returned value by 57.2974 (or 180/pi) to convert the returned value to degrees.

Syntax

```
acos(double value);
```

Arguments

The argument value is a double number between –1.0 and 1.0

Return values

The angle as a double in radians between 0 and pi

Example:

```
double theta, cosTheta;
cosTheta = 0.71;//cosine of Theta
theta=acos(cosTheta);//theta should be ~ pi/4 or 45 degrees
```

The ArcTangent function

The arctangent function returns the arc tangent or inverse tangent of the specified value in radians. Multiply the returned value by 57.2974 (180/pi) to convert the returned value to degrees.

Syntax

```
atan(double value);
```

Arguments

The argument value is the tangent of the angle as a double.

Return values

The angle in radians as a Double between –pi/2 and pi/2

Example:

```
double theta, tanTheta;
tanTheta = 0.71;//tangent of Theta
theta=acos(tanTheta);//theta should be ~ pi/4 or 45 degrees
```

The ArcTangent2 function

The arctangent2 function returns the arc tangent or inverse tangent of the specified value in radians. Multiply the returned value by 57.2974 (180/pi) to convert the returned value to degrees.

Syntax

```
atan2(X double value, Y double value);
```

Arguments

X and Y are any numbers as doubles, where X is the length of the opposite side and Y is the length of the adjacent side of the trigonometric triangle.

Return values

The angle in radians as a double between –pi and pi

Example

```
rad=atan2(2.0,1.0); //result is 1.1071 radians ~ 63 degrees
```

Note

By passing the arguments to this function with the respective signs, the true angle can be obtained (in the appropriate quadrant).

The Line Fit function

The LineFit uses linear regression to fit a line along a set of points.

Syntax

```
boolean result = LineFit(double xCoord[],double yCoord[],int max,
double abcParams[]);
```

Arguments

Double xCoord[ ]: array of x coordinates of the points to be used.

Double yCoord[ ]: array of y coordinates of the points to be used.

Int max: maximum number of points to use for the computation.

Double abcParams[ ]: array that will contain the outputs of the function. These outputs consists of three parameters: a, b, and c. These parameters describe the line by being inserted in the formula a x + b y + c = 0. The slope of the line is m = - b / a, this slope is measured from the x axis and it is positive if it is measured clockwise, negative otherwise.

Return values

Boolean result: this value is set to true if the function was executed with no errors, otherwise it is set to false.

Example:

```
//declare variables to be used as the function arguments
double blobXCoord[] = new double[bSelector.NumBlobs];
double blobYCoord[] = new double[bSelector.NumBlobs];
int maxPoints = bSelector.NumBlobs;
double lineOutput[] = new double[3];
boolean b;
//variable to control the loop
int counter = 1;

//loop to populate arrays with data from the blobs
while(counter <= maxPoints)
{
      blobXCoord[counter] = bSelector.BlobPosition.X[counter];
      blobYCoord[counter] = bSelector.BlobPosition.Y[counter];
      counter = counter + 1;
}

//compute line
b = LineFit(blobXCoord, blobYCoord, maxPoints, lineOutput);

//determine if the operation was successful
if(b)
{
      //output the slope of the line
      myScr.String = "Slope= " + (-lineOutput[1]/lineOutput[2]);
}
else
{
      //output the slope of the line
      myScr.String = "Unsuccessful operation";
}
```

Chapter 7 - Using system memory: DVT Registers

We mentioned before that one of the features that make Scripts very powerful is the ability to save data directly to the system memory. In the examples used until now to illustrate the syntax of scripts, we used variables. Variables are declared, initialized, used, and when the Script stops executing, they disappear and the data in them is lost. So if we wanted to share that data with the next inspection or with another type of Script, it would not be possible. This chapter explains the methods used to share data by using the system registers.

Registers as Global Variables

In order to share data, Scripts should have a concept of global variables. Global variables retain their data as long as the system is powered, and that data can be accessed and modified from any Script at anytime. Register can be accessed even from an external device that can open an Ethernet connection using the TCP/IP or Modbus TCP protocols. DVT Registers behave as global variables. The DVT registers is a set of 16384 8-bit registers that can store any type of data. The first 12 registers (registers 0 through 11) are reserved for a specific use, so we can arbitrarily change the values in registers 12 to 16384. We will discuss the commands used to transfer data from a variable to a register, but first, we will look at some important considerations about using registers. Registers have a size of 8 bits, so depending on the data type written to registers is how many register it takes.

Supported data types for register write/read operation			
Name	**Description**	**DVT Registers**	**Approximate Range**
byte	8-bit unsigned integer	1	0 to 255
short	16-bit signed integer	2	-32E03 to 32E03
int	32-bit signed integer	4	-2E09 to 2E09
long	64-bit signed integer	8	-9E18 to 9E18
float	32-bit signed floating point	4	1E-45 to 3E38 *
double	64-bit signed floating point	8	5E-324 to 2E308 *
String	8 bits/char+1 byte	# chars +1	N/A

Table 4: Number of registers needed for each one of the data types that can be saved to registers.

Notice that only those data types included in the table can be saved to registers. This might not seem to be such an important issue, but if the user is not careful with the size of the variables, data can be overwritten without warning. The example shown below illustrates how the data in memory can be easily altered by accident.

Writing an integer to register 101 will use the registers marked with an X

	0	1	2	3	4	5	6	8	7	8	9
100		X	X	X	X						
110											

Writing another integer to register 102 will use registers marked

with a 0, overwriting some of the X registers

	0	1	2	3	4	5	6	8	7	8	9
100		X	0	0	0	0					
110											

Write the new integer to register 105 instead, it will not alter the data

	0	1	2	3	4	5	6	8	7	8	9
100		X	X	X	X	0	0	0	0		
110											

The user must remember to skip some register in order to avoid altering previously saved data. A good rule of thumb is to skip ten registers every time. By doing this, the user will not overwrite basic data types. If the user needs to write Strings, which may take more than 10 registers, the user could select a different range of registers. For example, use registers starting at register 20 to save basic data types every ten registers. Use registers starting at register 3000 to write Strings skipping the necessary number of registers to avoid the destruction of data. The commands to write to and read from system registers are:

RegisterReadByte()

Reads a byte data type from a specified register

Syntax

```
RegisterReadByte(int regNum);
```

Arguments

An integer value indicating the register number

Return values

A byte value read from the register number specified by the variable regNum.

Example:

```
byte b;
b = RegisterReadByte(16);  //Reads 1 byte of Register 16
```

Notes

The byte data type occupies one DVT register.

RegisterReadShort()

Reads a short data type starting at a specific register

Syntax

```
RegisterReadShort(int regNum);
```

Arguments

An integer value indicating the register number

Return values

A short data type value starting at register location regNum.

Examples:

```
short s;
s = RegisterReadShort(100);//Big Endian Representation
s = RegisterReadShort(100,0);//Big Endian Representation
s = RegisterReadShort(100,1);//Little Endian Representation
                              //(Byte order reversed)
```

Notes

This short data type occupies two DVT registers.

RegisterReadInteger()

Reads an integer data starting at a specific register

Syntax

```
RegisterReadInteger(int regNum);
```

Arguments

An integer value indicating the register number

Return values

An integer data type value starting at register location regNum.

Examples:

```
int i;
i = RegisterReadInteger(100);//Big Endian Representation
i = RegisterReadInteger(100,0);//Big Endian Representation
i = RegisterReadInteger(100,1);//Little Endian Representation
                              //(Byte order reversed)
i = RegisterReadInteger(100,2);//16 bit Word order reserved
```

Notes

The integer data type occupies four DVT registers.

RegisterReadLong()

Reads a long data type starting at a specific register

Syntax

```
RegisterReadLong(int regNum);
```

Arguments

An integer value indicating the register number

Return values

A long data type value starting at register location regNum.

Examples:

```
long l;
l = RegisterReadLong(100);  //Big Endian Representation
l = RegisterReadLong(100,0);  //Big Endian Representation
l = RegisterReadLong(100,1);  //Little Endian Representation
                              //(Byte order reversed)
l = RegisterReadLong(100,2);  //32 bit Word order reversed
```

Notes

The long data type occupies eight DVT registers.

RegisterReadFloat()

Reads a float data type starting at a specific register

Syntax

```
RegisterReadFloat(int regNum);
```

Arguments

An integer value indicating the register number

Return values

A float data type value starting at register location regNum.

Examples:

```
float f;
f = RegisterReadFloat(100);//Big Endian Representation
f = RegisterReadFloat(100,0);//Big Endian Representation
f = RegisterReadFloat(100,1);//Little Endian Representation
                              //(Byte order reversed)
f = RegisterReadFloat(100,2);//16 bit Word order reversed
```

Notes

The float data type occupies four DVT registers.

RegisterReadDouble()

Reads a double data type starting at a specific register

Syntax

```
RegisterReadDouble(int regNum);
```

Arguments

An integer value indicating the register number

Return values

A double data type value starting at register location regNum.

Examples:

```
double d;
d = RegisterReadDouble(100);  //Big Endian Representation
(default)
d = RegisterReadDouble(100,0);  //Big Endian Representation
d = RegisterReadDouble(100,1);  //Little Endian Representation
                              //(Byte order reversed)
d = RegisterReadDouble(100,2);  //32 bit Word order reversed
```

Notes

The double data type occupies eight DVT registers.

RegisterReadString()

Reads a String data type starting at a specific register

Syntax

```
RegisterReadString(int regNum);
```

Arguments

An integer value indicating the register number

Return values

A string data type value starting at register location regNum.

Examples:

```
String MyString;
MyString = RegisterReadString(25);
```

Notes

The String data type is ended by the null character and it takes one register per character plus one for the null character.

RegisterWriteByte()

Writes a byte data to a specific register

Syntax

```
RegisterWriteByte(int regnum, byte b);
```

Arguments

Int regnum: integer value indicating the register number

Byte b: byte variable to be written to the register

Return values

Int result: result = 0 for success, result = -1 for failure (invalid register number).

Examples:

```
byte b;
b = 35;
RegisterWriteByte(16,b);
```

RegisterWriteShort()

Writes a short data type starting at a specific register

Syntax

```
RegisterWriteShort(int regnum, short s);
```

Arguments

Int regnum: integer value indicating the register number

Short s: short variable to be written to the register

Return values

Integer result. Result = 0 for success, result = -1 for failure (invalid register number).

Examples:

```
short a;
a = 234;
RegisterWriteShort(100,a);  //Big Endian Representation
RegisterWriteShort(100,a,0);  //Big Endian Representation
RegisterWriteShort(100,a,1);  //Little Endian Representation
(Byte order reversed)
```

Notes

The write operation for the short data type takes two DVT registers (starting at regNum).

RegisterWriteInteger()

Writes an integer data type starting at a specific register

Syntax

```
RegisterWriteInteger(int regnum, int i);
```

Arguments

Int regnum: integer value indicating the register number

Int i: int variable to be written to the register

Return values

Int result: result = 0 for success, result = -1 for failure (invalid register number).

Examples:

```
int j;
j = 2345;
RegisterWriteInteger(110,j);  //Big Endian Representation
RegisterWriteInteger(110,j,0);  //Big Endian Representation
RegisterWriteInteger(110,j,1);  //Little Endian Representation
(Bytes order reversed)
RegisterWriteInteger(110,j,2);  //16 bit word order reversed
```

Notes

The int data type occupies four DVT registers (starting at regNum).

RegisterWriteLong()

Writes a long data type starting at a specific register

Syntax

```
RegisterWriteLong(int regnum, long k);
```

Arguments

Int regnum: integer value indicating the register number

Long k: long variable to be written to the register

Return values

Integer result. Result = 0 for success, result = -1 for failure (invalid register number).

Examples:

```
long k;
k = 123456789;
RegisterWriteLong(120,k);  //Big Endian Representation
RegisterWriteLong(120,k,0);  //Big Endian Representation
RegisterWriteLong(120,k,1);  //Little Endian Representation
                              //(Byte order reversed)
RegisterWriteLong(120,k,2);  //32 bit Word order reversed
```

Notes

The long data types occupy eight registers (starting at RegNum).

RegisterWriteFloat()

Writes a float data type starting at a specific register

Syntax

```
RegisterWriteFloat(int regnum, float f);
```

Arguments

Int regnum: integer value indicating the register number

Float f: float variable to be written to the register

Return values

Integer result. Result = 0 for success, result = -1 for failure (invalid register number).

Examples:

```
float g;
g = 234.34;
RegisterWriteFloat(130,g);  //Big Endian Representation
RegisterWriteFloat(130,g,0);  //Big Endian Representation
RegisterWriteFloat(130,g,1);  //Little Endian Representation
                              //(Byte order reversed)
RegisterWriteFloat(130,g,2);  //16 bit Word order reversed
```

Notes

The float data types occupy four DVT registers (starting at regNum).

RegisterWriteDouble()

Writes a double data type starting at a specific register

Syntax

```
RegisterWriteDouble(int regnum, double d);
```

Arguments

Int regnum: integer value indicating the register number

Double d: double variable to be written to the register

Return values

Integer result. Result = 0 for success, result = -1 for failure (invalid register number).

Examples:

```
double e;
e = 123456.4321;
RegisterWriteDouble(140,e);  //Big Endian Representation
RegisterWriteDouble(140,e,0);  //Big Endian Representation
RegisterWriteDouble(140,e,1);  //Little Endian Representation
                              //(Byte order reversed)
RegisterWriteDouble(140,e,2);  //32 bit Word order reversed
```

Notes

The double data types occupy eight DVT registers (starting at regNum).

RegisterWriteString()

Writes a String data type starting at a specific register

Syntax

```
RegisterWriteString(int regnum, String myString);
```

Arguments

Int regnum: integer value indicating the register number

String myString: String variable to be written to the register

Return values

Integer result. Result = 0 for success, result = -1 for failure (invalid register number).

Examples:

```
String MsgString;
MsgString = "Failed to execute!";
RegisterWriteString(25,MsgString);
```

Notes

String data types occupy one register per string character (starting at regNum) plus one for the null character.

Chapter 8 – Input/Output Functions

Scripts have the ability to directly access or modify inputs and outputs. This is a very valuable feature of scripts. In many cases, users need to perform the entire inspection from a background script. However, the trigger signal does not trigger a background script; a different input must be used. In situations like these one and many others is where the direct access to I/O lines makes scripts extremely versatile tools. This chapter describes the different commands available for that purpose.

Use of I/O commands

Scripts do not directly access the system I/O lines. They access a certain location in memory where all the system inputs and outputs are saved: the virtual I/O. These can be manually mapped to the physical I/O from FrameWork by the user or directly accessed via Ethernet by accessing specific system registers. Figure 14 shows the mapping of the I/O lines.

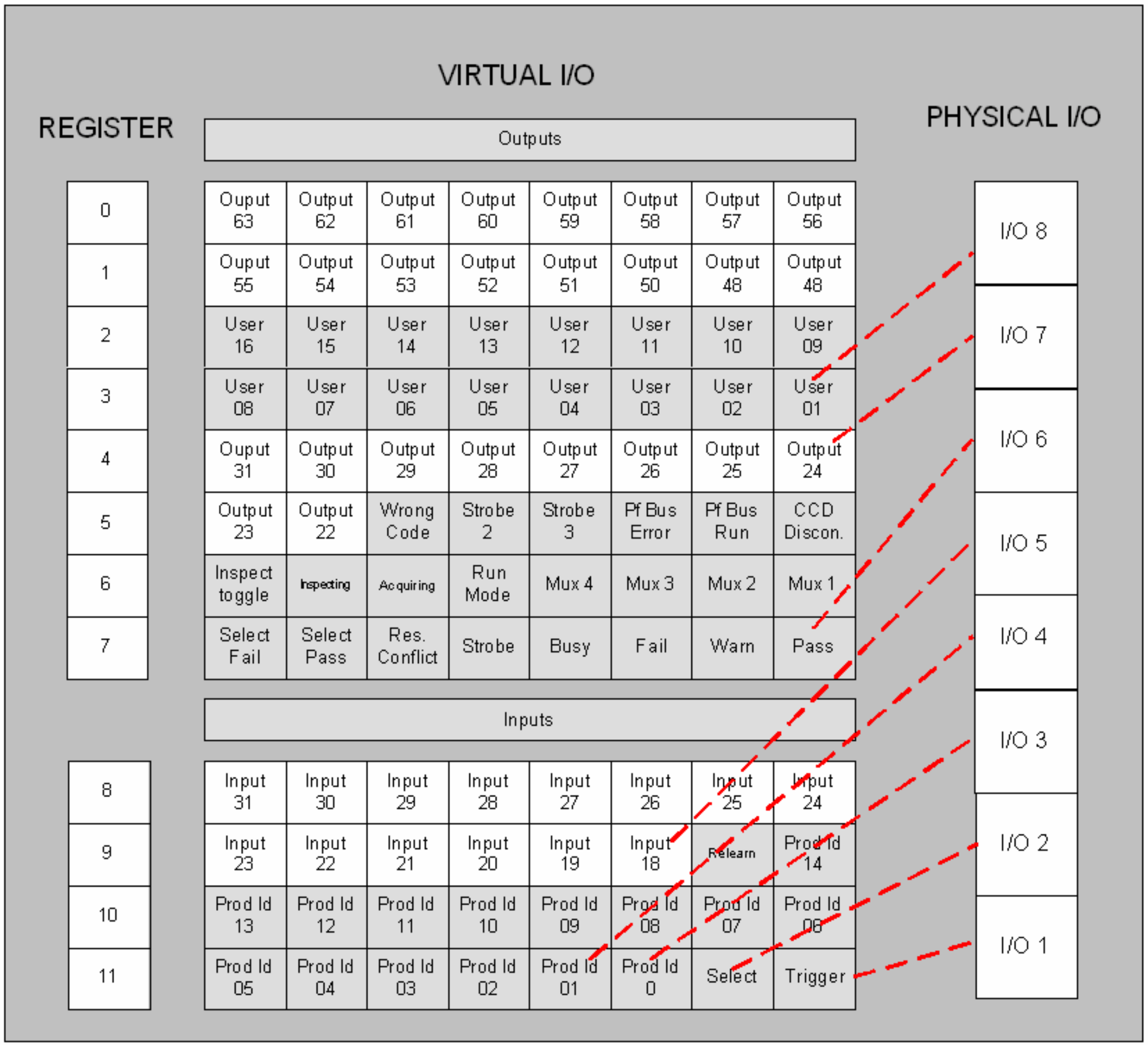


Figure 14: use of system I/O. All the I/O lines are mapped to system memory (first 11 registers) and any eight of them can be mapped to the physical I/O. The figure shows eight random I/O assigned to the physical I/O for illustration purposes only. The inputs and outputs highlighted in white are the ones reserved for scripts.

The figure shows one particular setup in which two lines are dedicated to script I/O. Input 18 and output 24 are assigned to physical I/O 4 and 7 respectively. This assignment will allow scripts in the SmartImage Sensor to monitor the physical I/O 4 and to set the value of I/O 7 as needed. Any of the I/O can be accessed at anytime via Ethernet by reading from or writing to the appropriate register. Notice that there are 32 virtual inputs and 64 virtual outputs. This is why in order to access each one; we use 32-bit variables for inputs and 64-bit variables for outputs in combination with bit manipulation commands. For example, if we wanted to check the status of input 24, we would assign the status of all 32 inputs to a variable and then mask out the bits we do not need. For details see bit manipulation.

There are two main types of input/output commands available from scripts: the ones that are executed and do not pause the execution of the program, and those who cause such delay. Foreground Scripts are part of an inspection, so they cannot be set to wait for an event or the results of the inspection will be delayed potentially causing serious timing issues. Thus, the only I/O commands available from Foreground Scripts are those that access the I/O to instantaneously get or set states. For Background Scripts both types of function are available, the standard group and those commands that pause execution waiting for a specific signal. All the commands are explained in this section.

GetInputs()

This function returns a 32-bit integer value representing the current state of the inputs bits. SmartImage Sensors have 32 inputs, each one of those inputs are assigned to one of the bits in this 32-bit variable. Bitwise operations must be used to mask out unnecessary data and extract only the desired bits. This function is available for both Foreground Scripts and Background Scripts.

Syntax

```
int i = GetInputs();
```

Arguments:

None

Return values

Int i: 32-bit integer value containing the status of the input lines.

Examples:

```
//the following script checks script input 19 (bit19)
//to determine if an value has been written
//to a certain register, if so, the register is read
int b=1;
byte myData;
if ((GetInputs() & (b<<19))!=0)
{
      myData = RegisterReadByte(20);//read the data
}
```

Note

For a complete list of the input/output mapping refer to the Input/Output Map.

GetOutputs()

This function returns a 64-bit integer value representing the current state of the outputs bits. SmartImage Sensors have 64 outputs, each one of those outputs are assigned to one of the bits in this 64-bit variable. Bitwise operations can be used to mask out unnecessary data and extract only the desired bits. This function is available for both Foreground Scripts and Background Scripts.

Syntax

```
long b = GetOutputs();
```

Arguments:

None

Return values

Long b: 64-bit integer value containing the status of the output lines.

Examples:

```
//wait for the busy output to go inactive
//busy output is output 3
long b=1;
while ((GetOutputs() & (b<<3))!=0)
{
      //empty loop
}
```

Note

For a complete list of the input/output mapping refer to the Input/Output Map.

SetInputs()

This function sets inputs high and low as needed. It takes in two arguments consisting of 32-bit integer values. One of them represents the bits to set to high and the other one the bits to set to low. SmartImage Sensors have 32 inputs, each one of those outputs are assigned to one of the bits in these 32-bit variables. This function is available for both Foreground Scripts and Background Scripts.

Syntax

```
SetIntputs(int on, int off);
```

Arguments:

Int on: represents the inputs to set to high. The bits in this variable that contain a 1 are the inputs that are set to high.

Int off: represents the inputs to set to low. The bits in this variable that contain a 1 are the inputs that are set to low.

If only ON or only OFF bits are used, the other argument should be set to zero.

Examples:

```
// toggle the digital relearn input bit (input 17)
int b=1;
SetInputs((b<<17),0);
sleep(5);//wait for 5 milliseconds
SetInputs(0, (b<<17));
```

Note

For a complete list of the input/output mapping refer to the Input/Output Map.

SetOutputs()

This function is use to set outputs either high or low. It takes in two arguments consisting of 64-bit integer values. One of them represents the bits to set to high and the other one the bits to set to low. SmartImage Sensors have 64 outputs, each one of those outputs are assigned to one of the bits in these 64-bit variables. This function is available for both Foreground Scripts and Background Scripts.

Syntax

```
SetOutputs(long on, long off);
```

Arguments:

Long on: represents the outputs to set to high. The bits in this variable that contain a 1 are the outputs that are set to high.

Long off: represents the outputs to set to low. The bits in this variable that contain a 1 are the outputs that are set to low.

If only ON or only OFF bits are used, the other argument can be a zero.

Examples:

```
//Hold both Outputs 24 and 25 active for 10 milliseconds
long b=1;
SetOutputs((b<<24) | (b<<25),0);
sleep(10);
SetOutputs(0, (b<<24) | (b<<25));
```

Note

For a complete list of the input/output mapping refer to the Input/Output Map.

SetOutputsAfterInspection()

This function can only set outputs high as needed. It takes in one argument consisting of a 64-bit integer value. It contains the bits indicating the outputs to set to high. SmartImage Sensors have 64 outputs, each one of those outputs is assigned to one of the bits in the 64-bit variable. This function is available for both Foreground Scripts and Background Scripts.

Syntax

```
SetOutputsAfterInspection(long on);
```

Arguments:

Long on: represents the outputs to set to high. The bits in this variable that contain a 1 are the outputs that are set to high.

Examples:

```
//Set an extra output (output 28) based on the result
//of a measurement SoftSensor called height.
long b=1;
float maxDistanceInPixels = 3.65;
if(height.Distance >= maxDistanceInPixels)
{
      SetOutputsAfterInspection(b<<28);
}
```

Notes

This function does not set the output immediately, it waits until the inspection is over and the inspection outputs are made available based on the I/O parameters setting for inspections. When the inspection outputs are reset based on user preferences (output pulse width), this output disappears as well (there is no need to reset it manually).

For a complete list of the input/output mapping refer to the Input/Output Map.

WaitOnInput ()

This function is allowed only in Background Scripts. When the Background Script gets to the line with this function, it blocks (non-busy wait) for at most a user definable number of milliseconds until the state of the virtual input indicated by the argument is set to active state. If the virtual input under consideration changes to high, the call returns with a success code and the execution continues. If the state of the virtual input under consideration does not change during the specified timeout, the call returns with a ‘timeout’ error code and the execution continues. If the virtual input under consideration is already at the desired state when the call is made, then it returns immediately with a success code and the execution continues.

Syntax

```
WaitOnInput(int inputToWaitOn, int timeout);
```

Arguments:

Int inputToWaitOn: virtual input to wait on represented by an integer value.

Int timeout: amount of time to wait in milliseconds. Timeout < 0 means wait forever.

Return value

Int result: result=0 for success, Result<0 for error

Error codes

Code -11: function called in a foreground script (JDIGITALIO_FOREGROUND)

Code -12: wait operation timed out (JDIGITALIO_TIMEDOUT)

Code -13: invalid parameters (JDIGITALIO_INVALID_PARAMETER)

Examples:

```
//Wait for input 31 to be set before reading a register.
//Wait at most 150 milliseconds
int result;
byte value;
result = WaitOnInput(31,150);
if(result == 0)
{
      value = RegisterReadByte(20);
}
```

Note

For a complete list of the input/output mapping refer to the Input/Output Map.

WaitOnAnyInput ()

This function is allowed only in background scripts. When the Background Script gets to the line with this function, it blocks (non-busy wait) for at most a user definable number of milliseconds until the state of ANY virtual input indicated by ActiveBits or InactiveBits becomes active or inactive respectively. If ANY virtual input under consideration changes to the desired state, the call returns with a success code and the execution continues. If the state of NONE of the virtual inputs under consideration changes during the specified timeout, the call returns with a 'timeout' error code and the execution continues. If ANY virtual input under consideration is already at the desired state when the call is made, then it returns immediately with a success code and the execution continues.

Syntax

```
WaitOnAnyInput(int activeIn, int inactiveIn, int timeout);
```

Arguments:

Int activeIn: virtual inputs to wait on represented by a 32-bit integer number. The input bits specified in this variable are expected to change to high.

Int inactiveIn: virtual inputs to wait on represented by a 32-bit integer number. The input bits specified in this variable are expected to change to low.

Int timeout: amount of time to wait in milliseconds. Timeout < 0 means wait forever.

Return value

Int result: result=0 for success, Result<0 for error

Error codes

Code -12: wait operation timed out (JDIGITALIO_TIMEDOUT)

Code -11: function called in a foreground script (JDIGITALIO_FOREGROUND)

Code -13: invalid parameters (JDIGITALIO_INVALID_PARAMETER)

Examples:

```
//Wait for inputs 30 or 31 to be set  or inputs
//28 or 29 to be reset before reading a register.
//Wait at most 150 milliseconds
int b=1, result;
byte value;
result = WaitOnAnyInput(b<<31|b<<30, b<<29|b<<28,150);
if(result == 0)
{
      value = RegisterReadByte(20);
}
```

Note

For a complete list of the input/output mapping refer to the Input/Output Map.

DVT Register and I/O Map

DVT Register No.	Bit Position								Type
	7	6	5	4	3	2	1	0	
0	Bit 63	Bit 62	Bit 61	Bit 60	Bit 59	Bit 58	Bit 57	Bit 56	Output
	Ouput 63	Output 62	Output 61	Output 60	Output 59	Output 58	Output 57	Output 56	
1	Bit 55	Bit 54	Bit 53	Bit 52	Bit 51	Bit 50	Bit 49	Bit 48	
	Ouput 55	Output 54	Output 53	Output 52	Output 51	Output 50	Output 49	Output 48	
2	Bit 47	Bit 46	Bit 45	Bit 44	Bit 43	Bit 42	Bit 41	Bit 40	
	User 16	User 15	User 14	User 13	User 12	User 11	User 10	User 09	
3	Bit 39	Bit 38	Bit 37	Bit 36	Bit 35	Bit 34	Bit 33	Bit 32	
	User 08	User 07	User 06	User 05	User 04	User 03	User 02	User 01	
4	Bit 31	Bit 30	Bit 29	Bit 28	Bit 27	Bit 26	Bit 25	Bit 24	
	Ouput 31	Output 30	Output 29	Output 28	Output 27	Output 26	Output 25	Output 24	
5	Bit 23	Bit 22	Bit 21	Bit 20	Bit 19	Bit 18	Bit 17	Bit 16	
	Output 23	Output 22	Wrong Code	Strobe 2	Strobe 3	Pf Bus Error	PfBus Run	CCD Discon.	
6	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	
	Inspect toggle	Inspecting	Acquiring	Run Mode	Mux 4	Mux 3	Mux 2	Mux 1	
7	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	
	Select Fail	Select Pass	Res. Conflict	Strobe	Busy	Fail	Warn	Pass	
8	Bit 31	Bit 30	Bit 29	Bit 28	Bit 27	Bit 26	Bit 25	Bit 24	Input
	Input 31	Input 30	Input 29	Input 28	Input 27	Input 26	Input 25	Input 24	
9	Bit 23	Bit 22	Bit 21	Bit 20	Bit 19	Bit 18	Bit 17	Bit 16	
	Input 23	Input 22	Input 21	Input 20	Input 19	Input 18	Releam	Prod Id 14	
10	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	
	Prod Id 13	Prod Id 12	Prod Id 11	Prod Id 10	Prod Id 09	Prod Id 08	Prod Id 07	Prod Id 06	
11	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	
	Prod Id 05	Prod Id 04	Prod Id 03	Prod Id 02	Prod Id 01	Prod Id 0	Select	Trigger	
12	Reg 12 bit 7	Reg 12 bit 6	Reg 12 bit 5	Reg 12 bit 4	Reg 12 bit 3	Reg 12 bit 2	Reg 12 bit 1	Reg 12 bit 0	General Purpose
...	...								
16382	Reg 16382 bit 7	Reg 16382 bit 6	Reg 16382 bit 5	Reg 16382 bit 4	Reg 16382 bit 3	Reg 16382 bit 2	Reg 16382 bit 1	Reg 16382 bit 0	
16383	Reg 16383 bit 7	Reg 16383 bit 6	Reg 16383 bit 5	Reg 16383 bit 4	Reg 16383 bit 3	Reg 16383 bit 2	Reg 16383 bit 1	Reg 16383 bit 0	

Chapter 9 - Communications

FrameWork contains a number of tools specifically designed for the implementation of communications with external devices. However, in some cases, users need a more custom process to be implemented to successfully exchange data with another device (for example when a specific handshake is needed to successfully communicate with an external device). For those cases, scripts expand the ability of FrameWork by adding the Socket, and Modbus Objects. Socket objects are used for TCP/IP communications over Ethernet connections, and Modbus Objects for using the Modbus TCP protocol over Ethernet connections. This chapter explains the functionality of the aforementioned objects.

Socket Object

This built-in object can used to perform generic socket read/write operations from a Background Scripts. The Socket Object has both methods and fields that allow the user to perform these actions and control some of the most common socket options. A socket object can have two different ways to establish a connection: as a client or as a server. As a client it only needs to be created and to connect to a server. As a server, it only needs to wait for a client to establish a connection to a certain port. After the connection is established in either way, bidirectional communication is viable. This section summarizes the functionality of the methods and fields and provides some examples.

Socket()

This is the constructor, it must be called to create and initialize the object.

Syntax

```
Socket(int type);
```

Arguments

Int type is an integer indicating the type of socket that should be created as indicated below:

No argument: creates a TCP socket.

1: creates a UDP socket.

The default value for type is 0.

Return values

Returns an object reference to the object or a null reference if operation failed or the function is called in a foreground script.

Example

```
Socket sock;
sock = new Socket();
if(sock == null)
{
      DebugPrint(“Creation of socket failed”);
}
```

Note

This function only works for Background Scripts.

Connect()

This method is used to establish a connection to a server. It is used only when the socket object created acts as a client.

Syntax

```
MySocket.Connect(string IPAddress, int port);
```

Arguments:

String IPAddress is a string containing the IP Address of the server.

Int Port is an integer value indicating the server port to connect to.

Return values

Int result: this function returns an integer. Result = 0 for success, and Result < 0 for failure.

Error codes

Code -11: already connected or connection error (SCRIPT_SOCKET_ERROR).

Code -21: Invalid IP address (SCRIPT_SOCKET_INVALID_IP)

Examples:

```
//create a socket and connect to IP Address 192.168.1.174 using
//port number 5001
Socket sock;
sock = new Socket();
Result = sock.Connect(“192.168.1.174”, 5001);
```

Bind()

This method is used to associate (‘Bind’) a socket object with a particular TCP port during the set-up sequence for a server. This method is used only when the socket object acts as a server.

Syntax

```
MySock.Bind(int port);
```

Arguments:

Int port: integer value indicating the port number to use.

Return values

Int result: this function returns an integer. Result = 0 for success, and Result < 0 for failure.

Error codes

Code -11: cannot bind to a connected socket (SCRIPT_SOCKET_ERROR).

Code -12: Failure to bind to port (SCRIPT_SOCKET_ERROR_BIND).

Example:

```
Socket sock;
sock = new Socket();
int result;
//bind socket to port 1000
result = sock.Bind(1000);
```

Listen()

This method is used to listen for connections on a socket that has been bound to a specific port. This method is used only when the socket object acts as a server.

Syntax

```
MySocket.Listen();
```

Arguments:

None

Return values

Int result: this function returns an integer. Result = 0 for success, and Result < 0 for failure.

Error codes

Code -11: Failure to use socket (SCRIPT_SOCKET_ERROR).

Code -13: Failure to listen on that port (SCRIPT_SOCKET_ERROR_LISTEN).

Example:

```
Socket sock;
sock = new Socket();
int result;
//bind original socket to port 1000
result = sock.Bind(1000);
if(result == 0)
{
      result = sock.Listen();
      //code continues here
}
```

Accept()

This method is used to accept connections on a socket that has been bound to a specific port. The result is a new socket that can be used to perform sends() and rcvs() to interchange data with another device. This method is used only when the socket object acts as a server.

Syntax

```
MySocket.Accept(NewSocket);
```

Arguments:

Socket NewSocket: socket object that will handle communications in this port.

Return values

Int result: this function returns an integer. Result = 0 for success, and Result < 0 for failure.

Error codes

Code -11: socket already connected or socket failed to accept connection (SCRIPT_SOCKET_ERROR).

Example:

```
Socket sock;
Socket newSock;
sock = new Socket();
newSock = new Socket();
int result;
//bind original socket to port 5005
result = sock.Bind(5005);
if(result == 0)//if no errors proceed
{
      result = sock.Listen();
      if(result == 0)//if no errors proceed
      {
            result = sock.Accept(newSock);
            //code continues here
      }
}
```

Recv()

This virtual method receives data into an array of bytes from a connected socket. The amount of data to be received is determined by the length of the predefined array. It is possible to set a timeout for this operation using the RecvTO field.

Syntax

```
MySocket.Recv(byte inData[]);
```

Arguments:

Byte inData: byte array that is populated as data is received.

Return values

Int result: this function returns an integer. Result >= 0 for success (the actual number indicates the number of bytes received) and Result < 0 for failure.

Error codes

Code -11: Failed to receive data (SCRIPT_SOCKET_ERROR).

Code -14: Timeout failure (SCRIPT_SOCKET_TIMEOUT).

Examples:

```
int result;
byte inData[];
inData = new byte[5]; //will receive 5 bytes of data
//assumes socket is created, initialized and connection
//is established
result = MySocket.Recv(inData);
```

Recv(, ,)

This method populates an array of bytes with data received from another device but instead of just filling up the array, it receives a limited number of bytes and it can be set to start at a predefined location in the array.

Syntax

```
MySocket.Recv(byte inData[], int offset, int len);
```

Arguments:

Byte inData: byte array that is populated as data is received.

Int offset: index of the array cell where data should start

Int len: maximum number of bytes to receive

Return values

Int result: this function returns an integer. Result >= 0 for success (the actual number indicates the number of bytes received) and Result < 0 for failure.

Error codes

Code -11: Failed to receive data or invalid arguments used (SCRIPT_SOCKET_ERROR).

Code -14: Timeout failure (SCRIPT_SOCKET_TIMEOUT).

Examples:

```
int result, len = 2, offset = 2;
byte inData[];
```

```
inData = new byte[5]; //will receive 5 bytes of data
//assumes socket is created, initialized and connection
//is established
result = MySocket.Recv(inData, offset, len);
```

Send()

This virtual method sends an array of bytes out using a connected socket. The amount of data to be sent is determined by the length of the predefined array.

Syntax

```
MySocket.Send(byte inData[]);
```

Arguments:

Byte inData: byte array that contains the data to be sent.

Return values

Int result: this function returns an integer. Result >= 0 for success (the actual number indicates the number of bytes sent) and Result < 0 for failure.

Error codes

Code -11: Error using socket or sending data (SCRIPT_SOCKET_ERROR).

Code -14: Timeout failure (SCRIPT_SOCKET_TIMEOUT).

Examples:

```
int result;
byte outData[];
outData = new byte[5]; //will send 5 bytes of data
//assumes socket is created, initialized and connection
//is established
result = MySocket.Send(outData);
```

Send(, ,)

This virtual method sends data from an array of bytes out using a connected socket. The amount of data to be sent is determined by the user.

Syntax

```
MySocket.Send(byte inData[], int offset, int len);
```

Arguments:

Byte inData: byte array that contains the data to be sent.

Int offset: index of the array cell where to start sending.

Int len: maximum number of bytes to send. If the end of the array is reached before this number, the operation stops.

Return values

Int result: this function returns an integer. Result >= 0 for success (the actual number indicates the number of bytes sent) and Result < 0 for failure.

Error codes

Code -11: Failed to receive data or invalid arguments used (SCRIPT_SOCKET_ERROR).

Code -14: Timeout failure (SCRIPT_SOCKET_TIMEOUT).

Examples:

```
int result, len = 2, offset = 2;
byte outData[];
outData = new byte[5]; //will send 5 bytes of data
//assumes socket is created, initialized and connection
//is established
result = MySocket.Send(inData, offset, len);
```

ConnectTO

This field of s Socket Object is used to set or to read the timeout setting for the connection. The default value is 5000 milliseconds (5 sec.).

Syntax

```
MySocketObject.ConnectTO;
```

Arguments:

None

Return values

The connection timeout setting

Example:

```
Socket Sock;
Sock = new Socket();
int getConnectTimeout;
getConnectTimeout = Sock.ConnectTO;//this line returns the value
Sock.ConnectTO = 3500;//this line sets it to 3.5 seconds
```

SendTO

This field of Socket Objects is used to set or to read the timeout setting for the process of sending data. The default value is 1000 milliseconds (1 sec.)

Syntax

```
MySocketObject.SendTO;
```

Arguments:

None

Return values

The timeout setting for sending data

Example:

```
Socket Sock;
Sock = new Socket();
int getSendTimeout;
getSendTimeout = Sock.SendTO;//this line returns the value
Sock.SendTO = 3500;//this line sets it to 3.5 seconds
```

RecvTO

This field of Socket Objects is used to set or to read the timeout setting for the process of receiving data. The default value is 1000 milliseconds (1 sec.)

Syntax

```
MySocketObject.RecvTO;
```

Arguments:

None

Return values

The timeout setting for receiving data

Example:

```
Socket Sock;
Sock = new Socket();
int getRecvTimeout;
getRecvTimeout = Sock.RecvTO;//this line returns the value
Sock.RecvTO = 3500;//this line sets it to 3.5 seconds
```

RecvFrom()

This virtual method receives a specified number of bytes into an array from a UDP socket. The data is placed in the array starting at a specified offset.

Syntax

```
MySocketObject.RecvFrom(byte b[], int offset, int len);
```

Arguments:

b[]: byte array to receive the data into.

offset: index to start from in b[].

len: maximum number of bytes to receive.

Return values

int result: Result >= 0 for success, actual number of bytes received. Result < 0 for error.

Error codes

Code -11: Error using socket or sending data (SCRIPT_SOCKET_ERROR).

Code -14: Timeout failure (SCRIPT_SOCKET_TIMEOUT).

Example:

```
// Create byte array to hold data
byte data[] = new byte[10];
// Create UDP socket
Socket UdpSocket = new Socket(1);
// Bind to UDP pot 50000
UdpSocket.Bind(50000);
// Set Receve timeout to 1 second
UdpSocket.RecvTO = 1000;
// Receive data sent to UDP port 50000
UdpSocket.Recv(data,0,data.length);
```

SendTo()

This virtual method sends a specified number of bytes from an array out from a UDP socket. The data is obtained from the array starting at a specified offset.

Syntax

```
MySocketObject.SendTo(String IpAddr, int UDPport, byte b[], int
offset, int len);
```

Arguments:

IpAddr[]: string with the IP address of the target in “xxx.xxx.xxx.xxx” format.

UDPport: UDP port to send the data to in the target node.

b[]: Byte array to send data from.

offset: index to start from in b[].

len: maximum amount of data to send. If the end of the array is reached before len bytes are sent, the operation stops.

Return values

int result: Result >= 0 for success, actual number of bytes sent. Result < 0 for error.

Error codes

Code -11: Failure to use socket (SCRIPT_SOCKET_ERROR).

Example:

```
// Create byte array to hold data
String data = "DVT";
// Create UDP socket
Socket UdpSocket = new Socket(1);
// Set Send Timeout to 1 second
UdpSocket.SendTO = 1000;
// Send to target IP and UDP port
UdpSocket.SendTo("192.168.0.1", 50000, data.toByteArray(), 0,
data.length());
```

Modbus Object

This built-in object can be used to perform single Modbus transfers of data from a Background Script. The new ‘MBtransfer’ type in script has both ‘methods’ and fields that allow the user to perform these actions and control some of the most common transfer parameters. Regular Modbus transfers can be implemented directly from the FrameWork user interface without using scripts; however, those transfers use a certain polling rate whereas the ones implemented via Modbus objects can be set to transfer data when certain events occur. This gives the user more control over the timing of the data transfer. The commands are explained here, an example is provided in a later chapter, for more information read the DVT integration notes available for download from the DVT website.

MBTransfer()

This is the constructor, it must be called to create and initialize the object.

Syntax

```
MyTransfer = MBTransfer();
```

Arguments:

None

Example:

```
MBtransfer MyTx;
MyTx = new MBTransfer();
//at this point the object has been created and initialized
```

Note

Note that the first statement uses MBtransfer (lowercase t) whereas the second statement uses MBTransfer() (uppercase t) that is because one is the declaration and the other one consists of the constructor

Connect()

This method initiates a Modbus connection to the slave IP indicated by a string passed as the argument. This connection remains open until the close() method is called.

Syntax

```
MyMbObject.Connect(String IP);
```

Arguments:

String IP: a string containing the IP address to connect to

Return values

Int result: result=0 for success, Result<0 for error

Error codes

Code -1: failure to open socket.(MBM_SOCKET_ERROR)

Code -11: socket already open (MBM_SCRIPT_SOCKET_OPEN)

Code -13: indicates IP contained an invalid IP address (MBM_SCRIPT_INVALID_IP)

Example:

```
int result;
MBtransfer MyTx;
MyTx = new MBTransfer();
result = MyTx.Connect("192.168.0.40");
```

Close()

This method closes the Modbus connection associated with the Modbus transfer object.

Syntax

```
MyMbObject.Close();
```

Arguments:

None

Return values

Int result: result=0 for success, Result<0 for error

Error codes

Code -1: socket already closed (MBM_SOCKET_ERROR)

Example:

```
int result;
MBtransfer MyTx;
MyTx = new MBTransfer();
result = MyTx.Connect("192.168.0.40");
if(result == 0)
{
      //execute transfers here
      result = MyTx.Close();
}
```

Note

Only use this function to close a socket before connecting to a new device. Sockets close automatically when the execution of the script ends.

Read()

This method performs a single Modbus read (FC= Read Multiple Regs) via the connection associated to the Modbus transfer object.

Syntax

```
MyTx.Read(int MasterReg, int SlaveReg, int NumRegs, int SendTO);
```

Arguments:

Int MasterReg: Modbus Register number in the MASTER (system containing the Background Script). Indicates where to start placing the data read.

Int SlaveReg: Modbus Register number of where to start reading the data from in the SLAVE.

Int NumRegs: Number of Modbus registers to read.

Int SendTO: max time in milliseconds to wait for a reply before assuming an error occurred:

Return values

Int result: result=0 for success, Result<0 for error

Error codes

Code -1: socket error occurred (MBM_SOCKET_ERROR).

Code -2: received response with wrong length (MBM_BAD_DATA_LENGTH).

Code -3: received response with bad header (MBM_BAD_HEADER).

Code -4: received response with bad word count (MBM_BAD_WORD_COUNT).

Code -5: received response with bad FC - error response (MBM_BAD_CMD).

Code -6: invalid Modbus address specified (MBM_ADDR_OUT_OF_RANGE).

Code -7:system Out of Memory (MBM_OUT_OF_MEMORY).

Example:

```
/*
Read 5 modbus regs from the slave’s Modbus reg #1000 into the
master’s modbus reg #500. Wait for response at most 1500
milliseconds
*/
MBtransfer MyTx;
MyTx = new MBTransfer();
result = MyTx.Connect("192.168.0.40");
if(result == 0)
{
      result = MyTx.Read(500,1000,5,1500);
}
```

Notes

Modbus registers are 16-bit long and DVT registers are 8-bit long. Transferring n Modbus registers into a SmartImage Sensor means that 2n registers will receive data. Likewise, transferring n DVT registers means transferring n/2 Modbus registers.

Write()

This method performs a single Modbus write (FC= Write Multiple Regs) via the connection associated to the Modbus transfer object.

Syntax

```
MyTx.Write(int MasterReg, int SlaveReg, int NumRegs, int SendTO);
```

Arguments:

Int MasterReg: Modbus Register number in the MASTER (system containing the Background Script). Indicates where to start reading the data to be transferred.

Int SlaveReg: Modbus Register number of where to start writing the data.

Int NumRegs: Number of Modbus registers to write.

Int SendTO: max time in milliseconds to wait for a reply before assuming an error occurred:

Return values

Int result: result=0 for success, Result<0 for error

Error codes

Code -1: socket error occurred (MBM_SOCKET_ERROR).

Code -2: received response with wrong length (MBM_BAD_DATA_LENGTH).

Code -3: received response with bad header (MBM_BAD_HEADER).

Code -4: received response with bad word count (MBM_BAD_WORD_COUNT).

Code -5: received response with bad FC - error response (MBM_BAD_CMD).

Code -6: invalid Modbus address specified (MBM_ADDR_OUT_OF_RANGE).

Code -7:system Out of Memory (MBM_OUT_OF_MEMORY).

Example:

```
/*
Write 4 modbus regs from the master’s Modbus reg #100 into the
slave’s modbus reg #200. Wait for response at most 1000
milliseconds
*/
MBtransfer MyTx;
MyTx = new MBTransfer();
result = MyTx.Connect("192.168.0.40");
if(result == 0)
{
      result = MyTx.Write(100,200,4,1000);
}
```

Notes

Modbus registers are 16-bit long and DVT registers are 8-bit long. Transferring n Modbus registers into a SmartImage Sensor means that 2n registers will receive data. Likewise, transferring n DVT registers means transferring n/2 Modbus registers.

ReadCoils()

This method performs a single Modbus read coils (FC= 1) via the connection associated to the Modbus transfer object.

Syntax

```
MyTx.ReadCoils(int MCoil, int SCoil, int NCoils, int SendTO);
```

Arguments:

Int MCoil: Modbus Coil number in the MASTER (system containing the Background Script). Indicates where to start placing the data read.

Int SCoil: Modbus Coil number of where to start reading the data (in the SLAVE).

Int NCoils: Number of Modbus coils to read.

Int SendTO: max time in milliseconds to wait for a reply before assuming an error occurred.

Return values

Int result: result=0 for success, Result<0 for error

Error codes

Code -1: socket error occurred (MBM_SOCKET_ERROR).

Code -2: received response with wrong length (MBM_BAD_DATA_LENGTH).

Code -3: received response with bad header (MBM_BAD_HEADER).

Code -4: received response with bad word count (MBM_BAD_WORD_COUNT).

Code -5: received response with bad FC - error response (MBM_BAD_CMD).

Code -6: invalid Modbus address specified (MBM_ADDR_OUT_OF_RANGE).

Code -7:system Out of Memory (MBM_OUT_OF_MEMORY).

Example:

```
/*
Read 5 modbus coils from the slave starting at coil #2 into the
master’s modbus coils starting at coil #1. Wait for response at
most 1500 milliseconds
*/
MBtransfer MyTx;
MyTx = new MBTransfer();
result = MyTx.Connect("192.168.0.40");
if(result == 0)
{
      result = MyTx.ReadCoils(1,2,5,1500);
}
```

Notes

Coil numbers are according to the Modbus TCP protocol. The documentation for the slave device should indicate the physical address that Modbus Coils are mapped to. In the Modicon 984 Coils are mapped to the 0-10000 address range. In order to simplify data manipulation, SmartImage Sensors support the use of coils. When the user writes to a number of coils in the registers, individual bits are accessed. See the register map at the end of this section for a reference on how to refer to individual bits by coil numbers.

WriteCoil()

This method performs a single Modbus write coil request (FC= 5) via the connection associated to the Modbus transfer object.

Syntax

```
MyTx.WriteCoil(int slaveCoil, int coilState, int SendTO);
```

Arguments:

Int slaveCoil: Modbus Coil number in the SLAVE. Indicates which coil to write to.

Int coilState: desired state for the coil: 0 turns coil OFF, 1 turn coil ON.

Int SendTO: max time in milliseconds to wait for a reply before assuming an error occurred:

Return values

Int result: result=0 for success, Result<0 for error

Error codes

Code -1: socket error occurred (MBM_SOCKET_ERROR).

Code -2: received response with wrong length (MBM_BAD_DATA_LENGTH).

Code -3: received response with bad header (MBM_BAD_HEADER).

Code -4: received response with bad word count (MBM_BAD_WORD_COUNT).

Code -5: received response with bad FC - error response (MBM_BAD_CMD).

Code -6: invalid Modbus address specified (MBM_ADDR_OUT_OF_RANGE).

Code -7:system Out of Memory (MBM_OUT_OF_MEMORY).

Example:

```
// Turn ON coil 5 in the slave with a timeout of 1.5 seconds
MBtransfer MyTx;
MyTx = new MBTransfer();
result = MyTx.Connect("192.168.0.40");
if(result == 0)
{
      result = MyTx.WriteCoil(5,1,1500);
}
```

Notes

Coil numbers are according to the Modbus TCP protocol. The documentation for the slave device should indicate the physical address that Modbus Coils are mapped to. In the Modicon 984 Coils are mapped to the 0-10000 address range. In order to simplify data manipulation, SmartImage Sensors support the use of coils. When the user writes to a number of coils in the registers, individual bits are accessed. See the register map at the end of this section for a reference on how to refer to individual bits by coil numbers.

ReadInputDiscretes()

This method performs a single Modbus read input discretes request (FC= 2) via the connection associated to the Modbus transfer object.

Syntax

```
MyTx.ReadInputDiscretes (int mDiscr, int sDiscr, int nBits, int
SendTO);
```

Arguments:

Int mDiscr: Modbus Input Discrete number in the MASTER (system containing the Background Script). Indicates where to start placing the data read.

Int sDiscr: Modbus Input Discrete number of where to start reading the data (in the SLAVE).

Int nBits: Number of Modbus Input Discretes to read.

Int SendTO: max time in milliseconds to wait for a reply before assuming an error occurred:

Return values

Int result: result=0 for success, Result<0 for error

Error codes

Code -1: socket error occurred (MBM_SOCKET_ERROR).

Code -2: received response with wrong length (MBM_BAD_DATA_LENGTH).

Code -3: received response with bad header (MBM_BAD_HEADER).

Code -4: received response with bad word count (MBM_BAD_WORD_COUNT).

Code -5: received response with bad FC - error response (MBM_BAD_CMD).

Code -6: invalid Modbus address specified (MBM_ADDR_OUT_OF_RANGE).

Code -7:system Out of Memory (MBM_OUT_OF_MEMORY).

Example:

```
/*
Read 5 Modbus Input Discretes from the slave, starting at Input
Discrete #2. Write them into the master starting at the master’s
Input Discrete #1. Wait for response at most 1500 milliseconds.
*/
MBtransfer MyTx;
MyTx = new MBTransfer();
result = MyTx.Connect("192.168.0.40");
if(result == 0)
{
      result = MyTx.ReadInputDiscretes(1,2,5,1500);
}
```

Notes

Input Discrete numbers are according to the Modbus TCP protocol. The documentation for the slave device should indicate the physical address that Modbus Input Discretes are mapped to. For example, in the Modicon 984, Input Discretes are mapped to the 10001-20000 address range.

ReadInputRegs()

This method performs a single Modbus read input registers request (FC= 4) via the connection associated to the Modbus transfer object.

Syntax

```
MyTx.ReadInputRegs (int mReg, int sReg, int nRegs, int SendTO);
```

Arguments:

Int mReg: Modbus register number in the MASTER (system containing the Background Script). Indicates where to start placing the incoming data.

Int sReg: Modbus register number of where to start reading the data (in the SLAVE).

Int nRegs: number of Modbus registers to read.

Int SendTO: max time in milliseconds to wait for a reply before assuming an error occurred:

Return values

Int result: result=0 for success, Result<0 for error

Error codes

Code -1: socket error occurred (MBM_SOCKET_ERROR).

Code -2: received response with wrong length (MBM_BAD_DATA_LENGTH).

Code -3: received response with bad header (MBM_BAD_HEADER).

Code -4: received response with bad word count (MBM_BAD_WORD_COUNT).

Code -5: received response with bad FC - error response (MBM_BAD_CMD).

Code -6: invalid Modbus address specified (MBM_ADDR_OUT_OF_RANGE).

Code -7:system Out of Memory (MBM_OUT_OF_MEMORY).

Example:

```
/*
Read 5 Modbus Input Registers from the slave’s Reg #100 into the
master’s Modbus Reg #50. Wait for response at most 1500
milliseconds
*/
MBtransfer MyTx;
MyTx = new MBTransfer();
result = MyTx.Connect("192.168.0.40");
if(result == 0)
{
      result = MyTx.ReadInputRegs(50,100,5,1500);
}
```

Notes

For this function code, the slave’s Input Registers are accessed (input registers are mapped starting at starting at address 30001 in a Modicon 984). Modbus registers are 16-bit long and DVT registers are 8-bit long. Transferring n Modbus registers into a SmartImage Sensor means that 2n registers will receive data. Likewise, transferring n DVT registers means transferring n/2 Modbus registers.

WriteRegister()

This method performs a single Modbus write single register request (FC= 6) via the connection associated to the Modbus transfer object.

Syntax

```
MyTx.WriteRegister(int mReg, int sReg, int SendTO);
```

Arguments:

Int mReg: Modbus Register number in the MASTER (system containing the Background Script). This is the register that contains the data to be written to the SLAVE.

Int sReg: Modbus Register number to write on the SLAVE.

Int SendTO: max time in milliseconds to wait for a reply before assuming an error occurred:

Return values

Int result: result=0 for success, Result<0 for error

Error codes

Code -1: socket error occurred (MBM_SOCKET_ERROR).

Code -2: received response with wrong length (MBM_BAD_DATA_LENGTH).

Code -3: received response with bad header (MBM_BAD_HEADER).

Code -4: received response with bad word count (MBM_BAD_WORD_COUNT).

Code -5: received response with bad FC - error response (MBM_BAD_CMD).

Code -6: invalid Modbus address specified (MBM_ADDR_OUT_OF_RANGE).

Code -7:system Out of Memory (MBM_OUT_OF_MEMORY).

Example:

```
/*
Write a single modbus register (master’s modbus reg#100) into the
slave’s modbus reg #200. Wait for response at most 1000
milliseconds.
*/
MBtransfer MyTx;
MyTx = new MBTransfer();
result = MyTx.Connect("192.168.0.40");
if(result == 0)
{
      result = MyTx.WriteRegister(100,200,1000);
}
```

Notes

For this function code, the slave’s holding registers are accessed (holding registers are mapped starting at starting at address 40001 in a Modicon 984). Modbus registers are 16-bit long and DVT registers are 8-bit long. Transferring 1 Modbus register into a SmartImage Sensor means that 2 DVT registers will receive data. Likewise, transferring 2 DVT registers means transferring 1 Modbus register.

ReadStatus()

This method performs a Modbus Read Exception Status request (FC =7) via the connection associated to the Modbus transfer object

Syntax

```
MyTx.ReadStatus(int mReg, int sentTO);
```

Arguments:

Int mReg: Modbus Register number in the MASTER (system containing the BG script.) Indicates where to place the status data.

Int sendTO: maximum time in milliseconds to wait for a reply before assuming an error occurred.

Return values

Int result: result=0 for success, Result<0 for error

Error codes

Code -1: socket error occurred (MBM_SOCKET_ERROR).

Code -2: received response with wrong length (MBM_BAD_DATA_LENGTH).

Code -3: received response with bad header (MBM_BAD_HEADER).

Code -4: received response with bad word count (MBM_BAD_WORD_COUNT).

Code -5: received response with bad FC - error response (MBM_BAD_CMD).

Code -6: invalid Modbus address specified (MBM_ADDR_OUT_OF_RANGE).

Code -7:system Out of Memory (MBM_OUT_OF_MEMORY).

Example:

```
/*
Read the status from a slave and place data in the master’s
modbus register #100. Wait for response at most 1500
milliseconds.
*/
MBtransfer MyTx;
MyTx = new MBTransfer();
result = MyTx.Connect("192.168.0.40");
if(result == 0)
{
      result = MyTx.ReadStatus (100,1500);
}
```

Notes

Status data is 1 byte in length and it is placed in the LSB of the Modbus register (which contains two bytes). Slave devices generally map 8 coils to the status and send this data in response to a read status function code. Consult the documentation of each slave for details. Modbus registers are 16-bit long and DVT registers are 8-bit long. Transferring 1 Modbus register into a SmartImage Sensor means that 2 DVT registers will receive data. Likewise, transferring 2 DVT registers means transferring 1 Modbus register. In this case, only the lowest 8 bits of the Modbus register contain data, so the actual status data occupies bits 0 to 7 of the DVT register 201 after execution of the command above.

Mapping DVT Registers to Modbus Coils

DVT Register No.	Modbus Register No.	Bit Position								Type
		7	6	5	4	3	2	1	0	
0	0	Coil 7	Coil 6	Coil 7	Coil 4	Coil 3	Coil 2	Coil 1	Coil 0	
		Ouput 63	Output 62	Output 61	Output 60	Output 59	Output 58	Output 57	Output 56	
1		Coil 15	Coil 14	Coil 13	Coil 12	Coil 11	Coil 10	Coil 9	Coil 8	
		Ouput 55	Output 54	Output 53	Output 52	Output 51	Output 50	Output 49	Output 48	
2	1	Coil 23	Coil 22	Coil 21	Coil 20	Coil 19	Coil 18	Coil 17	Coil 16	
		User 16	User 15	User 14	User 13	User 12	User 11	User 10	User 09	
3		Coil 31	Coil 30	Coil 29	Coil 28	Coil 27	Coil 26	Coil 25	Coil 24	
		User 08	User 07	User 06	User 05	User 04	User 03	User 02	User 01	Output
4	2	Coil 39	Coil 38	Coil 37	Coil 36	Coil 35	Coil 34	Coil 33	Coil 32	
		Ouput 31	Output 30	Output 29	Output 28	Output 27	Output 26	Output 25	Output 24	
5		Coil 47	Coil 46	Coil 45	Coil 44	Coil 43	Coil 42	Coil 41	Coil 40	
		Output 23	Output 22	Wrong Code	Strobe 2	Strobe 3	Pf Bus Error	Pf Bus Run	CCD Discon.	
6	3	Coil 55	Coil 54	Coil 53	Coil 52	Coil 51	Coil 50	Coil 49	Coil 48	
		Inspect toggle	Inspecting	Acquiring	Run Mode	Mux 4	Mux 3	Mux 2	Mux 1	
7		Coil 63	Coil 62	Coil 61	Coil 60	Coil 59	Coil 58	Coil 57	Coil 56	
		Select Fail	Select Pass	Res. Conflict	Strobe	Busy	Fail	Warn	Pass	
8	4	Coil 71	Coil 70	Coil 69	Coil 68	Coil 67	Coil 66	Coil 65	Coil 64	
		Input 31	Input 30	Input 29	Input 28	Input 27	Input 26	Input 25	Input 24	
9		Coil 79	Coil 78	Coil 77	Coil 76	Coil 75	Coil 74	Coil 73	Coil 72	
		Input 23	Input 22	Input 21	Input 20	Input 19	Input 18	Relearn	Prod Id 14	Input
10	5	Coil 87	Coil 86	Coil 85	Coil 84	Coil 83	Coil 82	Coil 81	Coil 80	
		Prod Id 13	Prod Id 12	Prod Id 11	Prod Id 10	Prod Id 09	Prod Id 08	Prod Id 07	Prod Id 06	
11		Coil 95	Coil 94	Coil 93	Coil 92	Coil 91	Coil 90	Coil 89	Coil 88	
		Prod Id 05	Prod Id 04	Prod Id 03	Prod Id 02	Prod Id 01	Prod Id 0	Select	Trigger	
12	6	Coil 103	Coil 102	Coil 101	Coil 100	Coil 99	Coil 98	Coil 97	Coil 96	
		Reg 12 bit 7	Reg 12 bit 6	Reg 12 bit 5	Reg 12 bit 4	Reg 12 bit 3	Reg 12 bit 2	Reg 12 bit 1	Reg 12 bit 0	
...		...								General Purpose
16382	8191	Coil 131063	Coil 131062	Coil 131061	Coil 131060	Coil 131059	Coil 131058	Coil 131057	Coil 131056	
		Reg 16382 bit 7	Reg 16382 bit 6	Reg 16382 bit 5	Reg 16382 bit 4	Reg 16382 bit 3	Reg 16382 bit 2	Reg 16382 bit 1	Reg 16382 bit 0	
16383		Coil 131071	Coil 131070	Coil 131069	Coil 131068	Coil 131067	Coil 131066	Coil 131065	Coil 131064	
		Reg 16383 bit 7	Reg 16383 bit 6	Reg 16383 bit 5	Reg 16383 bit 4	Reg 16383 bit 3	Reg 16383 bit 2	Reg 16383 bit 1	Reg 16383 bit 0	

Chapter 10 – OEM Functions

FrameWork includes a number of commands that are used for the interaction of SmartImage Sensors with specific external devices. They provide a way to share data with them using simple script commands. This chapter illustrates the three types of OEM functions that scripts use.

FrameWork includes specific drivers to that can be used to share data with external devices. The three types of commands that are explained here are: AB functions (used to communicate with Allen Bradley devices using the Ethernet/IP protocol), Fanuc functions (used to exchange data with Fanuc robots) and Motoman functions (which exchange data with Motoman robots). The syntax and basic use of the commands are explained here. For more specification on the setup please see the DVT integration notes.

Allen Bradley Functions

The Allen Bradley (or AB) functions are used in conjunction with the Ethernet/IP protocol. To transfer information between SmartImage Sensors and Allen Bradley products using the aforementioned protocol.

AB_RegisterReadDINT()

The AB_Register functions are used in conjunction with the EtherNet/IP protocol to transfer information between the DVT SmartImage Sensor and an Allen Bradley system running EtherNet/IP. The AB_RegisterReadDINT() function allows the user to read from individual registers in the DINTS block of memory.

Syntax

```
 value = AB_RegisterReadDINT ( int index );
```

Arguments

Int index: index in the DINTS data block. The range for this parameter is determined by the data type and the size of the data block. In this case (32-bit signed integer) the valid range goes from 0 to 63 indices.

Return values

int value. Value stored at the specified index in the DINTS data block.

Error codes:

Code -1: Index was out of range

Example

```
int value;
 value = AB_RegisterReadDINT( 0 );
```

Notes

When this function is executed, data is updated only in registers within the DVT system. Communication with the Allen Bradley device only occurs when initiated by a MSG instruction from the PLC.

AB_RegisterReadINT()

The AB_Register functions are used in conjunction with the EtherNet/IP protocol to transfer information between the DVT SmartImage Sensor and an Allen Bradley system running EtherNet/IP. The AB_RegisterReadINT() function allows the user to read from individual registers in the INTS block of memory.

Syntax

```
      value = AB_RegisterReadINT ( int index );
```

Arguments

Int index: index in the INTS data block. The range for this parameter is determined by the data type and the size of the data block. In this case (16-bit signed integer) the valid range goes from 0 to 127 indices.

Return values

Short value. Value stored at the specified index in the INTS data block.

Error codes:

Code -1: Index was out of range

Example

```
short value;
value = AB_RegisterReadINT( 0 );
```

Note

When this function is executed, data is updated only in registers within the DVT system. Communication with the Allen Bradley device only occurs when initiated by a MSG instruction from the PLC.

AB_RegisterReadREAL()

The AB_Register functions are used in conjunction with the EtherNet/IP protocol to transfer information between the DVT SmartImage Sensor and an Allen Bradley system running EtherNet/IP. The AB_RegisterReadREAL() function allows the user to read from individual registers in the REALS block of memory.

Syntax

```
 value = AB_RegisterReadREAL (int index );
```

Arguments

Int index: index in the REALS data block. The range for this parameter is determined by the data type and the size of the data block. In this case (32-bit floating point) the valid range goes from 0 to 63 indices.

Return values

float value. Value stored at the specified index in the REALS data block.

Error codes:

Code -1: Index was out of range

Example

```
float value;
 value = AB_RegisterReadREAL( 0 );
```

Notes

When this function is executed, data is updated only in registers within the DVT system. Communication with the Allen Bradley device only occurs when initiated by a MSG instruction from the PLC.

AB_RegisterReadSINT()

The AB_Register functions are used in conjunction with the EtherNet/IP protocol to transfer information between the DVT SmartImage Sensor and an Allen Bradley system running EtherNet/IP. The AB_RegisterReadSINT() function allows the user to read from individual registers in the SINTS block of memory.

Syntax

```
bit value = AB_RegisterReadSINT ( int index );
```

Arguments

Int index: index in the SINTS data block. The range for this parameter is determined by the data type and the size of the data block. In this case (8-bit signed integer) the valid range goes from 0 to 255 indices.

Return values

byte value. Value stored at the specified index in the SINTS data block.

Error codes:

Code -1: Index was out of range

Example

```
int value;
 value = AB_RegisterReadSINT( 0 );
```

Notes

When this function is executed, data is updated only in registers within the DVT system. Communication with the Allen Bradley device only occurs when initiated by a MSG instruction from the PLC.

AB_RegisterReadString()

The AB_Register functions are used in conjunction with the EtherNet/IP protocol to transfer information between the DVT SmartImage Sensor and an Allen Bradley system running EtherNet/IP. The AB_RegisterReadString() function allows the user to read data from the SINTS block of memory.

Syntax

```
 str = AB_RegisterReadString ( int index );
```

Arguments

Int index: index in the SINTS data block. The range for this parameter is determined by the data type and the size of the data block. In this case (8-bit signed integer) the valid range goes from 0 to 255 indices.

Return values

String str. Value stored at the specified index in the SINTS data block.

Error codes:

Code -1: Index was out of range

Example

```
String str;
 str = AB_RegisterReadString( 0 );
```

Notes

When this function is executed, data is updated only in registers within the DVT system. Communication with the Allen Bradley device only occurs when initiated by a MSG instruction from the PLC.

AB_RegisterWriteINT()

The AB_Register functions are used in conjunction with the EtherNet/IP protocol to transfer information between the DVT SmartImage Sensor and an Allen Bradley system running EtherNet/IP. The AB_RegisterWriteINT() function allows the user to write to individual registers in the INTS block of memory.

Syntax

```
 result = AB_RegisterWriteINT ( int index, short value);
```

Arguments

Int index: index in the INTS data block.

Short value: value to write at the specified index.

Notes

The ranges for the index and value parameters are determined by the data type and the size of the data block.

Return values

int result. result = 0 for success, result < 0 if error occurred

Error codes:

Code -1: Index was out of range

Example

```
int result;
 result  = AB_RegisterWriteINT( 0, 150 );
```

Notes

When this function is executed, data is updated only in registers within the DVT system. Communication with the Allen Bradley device only occurs when initiated by a MSG instruction from the PLC.

AB_RegisterWriteDINT()

The AB_Register functions are used in conjunction with the EtherNet/IP protocol to transfer information between the DVT SmartImage Sensor and an Allen Bradley system running EtherNet/IP. The AB_RegisterWriteDINT() function allows the user to write to individual registers in the DINTS block of memory.

Syntax

```
 result = AB_RegisterWriteDINT ( int index, int value);
```

Arguments

Int index: index in the DINTS data block.

Int value: value to write at the specified index.

Notes

The ranges for the index and value parameters are determined by the data type and the size of the data block.

Return values

int result. result = 0 for success. result < 0 for error

Error codes:

Code -1: Index was out of range

Example

```
int result;
 result  = AB_RegisterWriteDINT( 0, FeatCount.NumFeatures );
```

Notes

When this function is executed, data is updated only in registers within the DVT system. Communication with the Allen Bradley device only occurs when initiated by a MSG instruction from the PLC.

AB_RegisterWriteREAL()

The AB_Register functions are used in conjunction with the EtherNet/IP protocol to transfer information between the DVT SmartImage Sensor and an Allen Bradley system running EtherNet/IP. The AB_RegisterWriteREAL() function allows the user to write to individual registers in the REALS block of memory.

Syntax

result = AB_RegisterWriteREAL (int index, float value);

Arguments

Int index: index in the REALS data block.

Float value: value to write at the specified index.

Notes

The ranges for the index and value parameters are determined by the data type and the size of the data block.

Return values

int result. result = 0 for success, result < 0 if error occurred.

Error codes:

Code -1: Index was out of range

Example

```
int result;
 result  = AB_RegisterWriteREAL( 0, Meas.Distance );
```

Notes

When this function is executed, data is updated only in registers within the DVT system. Communication with the Allen Bradley device only occurs when initiated by a MSG instruction from the PLC.

AB_RegisterWriteSINT()

The AB_Register functions are used in conjunction with the EtherNet/IP protocol to transfer information between the DVT SmartImage Sensor and an Allen Bradley system running EtherNet/IP. The AB_RegisterWriteSINT() function allows the user to write to individual registers in the SINTS block of memory.

Syntax

```
 result = AB_RegisterWriteSINT ( int index, byte value);
```

Arguments

Int index: index in the SINTS data block.

Byte value: value to write at the specified index.

Notes

The ranges for the index and value parameters are determined by the data type and the size of the data block.

Return values

int result. result = 0 for success, result < 0 if error occurred.

Error codes:

Code -1: Index was out of range

Example

```
int result;
 result  = AB_RegisterWriteSINT( 0, Gen.NumBlobs );
```

Notes

When this function is executed, data is updated only in registers within the DVT system. Communication with the Allen Bradley device only occurs when initiated by a MSG instruction from the PLC.

AB_RegisterWriteString()

The AB_Register functions are used in conjunction with the EtherNet/IP protocol to transfer information between the DVT SmartImage Sensor and an Allen Bradley system running EtherNet/IP. The AB_RegisterWriteString() function allows the user to write to individual registers in the SINTS block of memory.

Syntax

```
result = AB_RegisterWriteString ( int index, String str );
```

Arguments

Int index: index in the SINTS data block.

String str: value to write at the specified index.

Notes

The ranges for the index and value parameters are determined by the data type and the size of the data block.

Return values

int result. result = 0 for success, result < 0 if error occurred.

Error codes:

Code -1: Index was out of range

Example

```
int result;
 result  = AB_RegisterWriteString( 0, Reader.String );
```

Note

When this function is executed, data is updated only in registers within the DVT system. Communication with the Allen Bradley device only occurs when initiated by a MSG instruction from the PLC.

Motoman Functions

DVT Scripts offer a number of functions to send data to Motoman robots. The commands used are explained below. For a complete description of this procedure (including physical connections) see the DVT integration notes.

MotoWriteByte()

This function writes a byte variable to the Motoman controller

Syntax

```
result = MotoWriteByte(byte varNum, byte Value);
```

Arguments

byte varNum specifies the variable number to use

byte Value specifies the value to transfer

Return values

0 = success

-1 = invalid Index or varNum (MR_INVALID_INDEX)

-2 = write error (MR_WRITE_ERROR)

-3 = other error (GENERAL_ERROR)

Example:

```
int result;
byte b;
b = 225;
Result = MotoWriteByte(15,b);
```

Notes

The Byte variable type includes values from 0 to 25, and the varNum parameter must be a number from 0 to 39.

MotoWriteInt()

This function writes an integer Motoman data type variable(equivalent to `short` DVT script data type)

Syntax

```
result = MotoWriteInt(byte VarNum, short Value);
```

Arguments

byte varNum specifies the variable number to use

short Value specifies the value to transfer

Return values

0 = success

-1 = invalid Index or varNum (MR_INVALID_INDEX)

-2 = write error (MR_WRITE_ERROR)

-3 = other error (GENERAL_ERROR)

Example:

```
int result;
short s;
s = 26;
result = MotoWriteInt(15,s);
```

Notes

This function writes a DVT short variable type to a Motoman Integer variable type (both variables are integer data types of 16 bits) ranging from –32,768 to 32,767.The varNum parameter must be a number from 0 to 39.

MotoWriteDouble()

This function writes a Double precision integer Motoman data type variable (equivalent to `int` DVT script data type)

Syntax

```
result = MotoWriteDouble(byte varNum, int Value);
```

Arguments

byte varNum specifies the variable number to use

int Value specifies the value to transfer

Return values

0 = success

-1 = invalid Index or varNum (MR_INVALID_INDEX)

-2 = write error (MR_WRITE_ERROR)

-3 = other error (GENERAL_ERROR)

Example:

```
int i, result;
i = 2500;
result = MotoWriteDouble(25,i);
```

Notes

This function writes a DVT Integer variable type to a Motoman Double Precision integer double variable type. The varNum parameter must be a number from 0 to 39.

MotoWriteReal()

This function writes a Real Motoman data type variable (equivalent to `float` DVT script data type)

Syntax

```
result = MotoWriteReal(byte varNum, float Value);
```

Arguments

byte varNum specifies the variable number to use

float Value specifies the value to transfer

Return values

0 = success

-1 = invalid Index or varNum (MR_INVALID_INDEX)

-2 = write error (MR_WRITE_ERROR)

-3 = other error (GENERAL_ERROR)

Example:

```
float f;
int Result;
f = 3256.258;
Result = MotoWriteReal(29,f);
```

Notes

This function writes a DVT Float variable type to a Motoman Real variable type. The varNum parameter must be a number from 0 to 39.

MotoWritePvar()

This function writes a Pvar Motoman data type variable (equivalent to six `float` DVT script data type variables)

Syntax

```
MotoWritePvar(byte varNum,float x, float y, float z,float
thetax, float thetay,float thetaz);
```

Arguments

byte varNum specifies the variable number to use

float x specifies the x position

float y specifies the y position

float z specifies the z position

float thetax specifies the x angle

float thetay specifies the y angle

float thetaz specifies the z angle

Return values

0 = success

-1 = invalid Index or varNum (MR_INVALID_INDEX)

-2 = write error (MR_WRITE_ERROR)

-3 = other error (GENERAL_ERROR)

Example:

```
float x,y,tx;
int Result;
x = 123;
y = 256.36;
tx =1.235;
Result = MotoWritePvar(9,x,y,0,tx,0,0);
```

Notes

This function writes a series of DVT Float variable types to a Motoman P variable having a format of (x, y, z, tx, ty, tz) were each element is a Motoman Real data type. The varNum parameter must be a number from 0 to 127.

Fanuc Functions

DVT SmartImage Sensors can implement Fanuc's Sensor Interface Protocol. This Protocol can be used to transfer information between Fanuc's Robot controllers (RJ-2/RJ-3) and other sensing devices such as DVT SmartImage Sensors. The information is transferred through a point-to-point serial connection (RS-232C). Using special function calls in DVT's script tool, information from the inspections (e.g. position of an object or number of objects found) is placed into the registers were the RJ-2/RJ-3 can access them with the SEND[] and REC[] Teach Pendant commands. The functions supported by DVT scripts are included below. For a complete reference including physical connections, refer to the DVT integration notes.

FanucSetup()

Syntax

```
FanucSetup(int regsToSend, int posRegsToSend)
```

Arguments:

int regsToSend specifies the number of general registers to send during the communication session. The valid range is 0-255. The range is defined by the number of general registers in the Fanuc RJ-2 controller.

int posRegsToSend specifies the number of position registers to send during the communication session. The valid range is 0-10.

Return values

Int result. Result = 0 for success, result < 0 or result > 0 for error.

Example:

```
FanucSetup(1,1);
```

This instruction tells driver to send 1 data register and 1 position register. The contents of general register 1 in the DVT sensor are sent to the controller's general register 1. Position register 1 in the DVT sensor is then sent to the position register indicated by the SENS_IF$ parameter in the controller.

Notes

Registers are sent in ascending order only. For instance: `FanucSetup(5,0)` will cause the driver to send general registers 1-5.

FanucWriteReg()

Syntax

```
FanucWriteReg(int regNum, int value);
```

Arguments:

int regNum specifies the address of the Fanuc general register where value will be placed. It also specifies the address of the general register in the controller where the data will be sent during the communication session. The valid range is 0-255. The range is defined by the number of general registers in the Fanuc RJ-2 controller.

int value specifies the data. It may be a literal number or a value from the sensors. The valid range is + 8388607 to -8388608. Internally, the Fanuc Controller stores general registers in signed 3-byte numbers. This is why the range is smaller than the valid range for a DVT 4-byte Integer.

Return values

Int result: result = 0 for success, result < 0 or result > 0 for error.

Example:

```
FanucWriteReg(1,1267);
```

This command writes 1267 to Fanuc's general register #1 in DVT's registers. During the comm. session, the value 1267 is written to Fanuc's general register #1 on the RJ-2 controller.

FanucWritePReg()

Syntax

```
FanucWritePReg(int pRegNum, double x, double y, double z, double
tx, double ty, double tz);
```

Arguments:

int pRegNum specifies the address of the Fanuc position register where the data will be placed. The Valid range is 0-10.

double x, y, and z specify the position data. They may be literals or values from the sensors. The units for these values must be mm. The valid range for positions is +83886.07 mm to –83886.08 mm. Internally, the Fanuc Controller stores each element of a position register as a scaled, signed, 3-byte number. This explains the range and units requirement.

doubles tx,ty,tz specify the rotation. They may be literals or values from the sensors. The units for these values are radians. The valid range for these angles is **+ 83.8607 rad to –83.88608 rad**. Internally, the Fanuc Controller stores each element of a position register as a scaled, signed, 3-byte number. This explains the range and units requirement.

Return values

Int result: result = 0 for success, result < 0 or result > 0 for error. Errors may occur for invalid index passed or passing out-of-range register numbers.

Examples:

```
FanucWritePReg(1,1.0,2.0,0,0,0,PI/4);
FanucWritePReg(1, sel.BlobTransformedPoint.X[1],
sel.BlobTransformedPoint.Y[1],0,0,0
(SoftSensor.BlobAngle[1])*(PI/180));
```

Writes the position to Fanuc's Position register #1 in DVT's registers. During the comm. session, it is sent to a position register defined by $SENS_IS on the RJ-2 controller.

Notes

The variables z, tx, and ty are currently forced to zero.

This function only works if the inspection mode is ON (inspections are running).

Chapter 11 – Working with Images

Scripts offer extra options to work with images. They contain two main groups of function for that purpose: the imaging functions and the Image Object. Imaging functions are functions that can be called at anytime to get specific data from the image or to mark the image. These functions must be used in Foreground Scripts only. They are designed to allow the user to perform special marking in the image. Using these functions users can draw points and lines in the image. Image objects are exclusive of Background Scripts, they have a number of built-in methods that allow the preprocessing of the image before the SoftSensors analyze it. Image objects are used to control the flow of inspections by determining when to acquire an image and which product to use to inspect it. This chapter discusses both imaging functions and Image objects.

Imaging Functions

Imaging functions provide three types of functionality to Foreground Scripts: image marking, access to image ID and access to intensity levels of pixels in the image.

Image(,)

The Image(,) function returns the intensity value of the specified coordinate position. The value returned will be a number between 0 and 255.

Syntax

```
Image(int row, int col);
```

Arguments:

Int row: indicates the X-coordinate of the pixel to be scanned.

Int col: indicates the Y-coordinate of the pixel to be scanned.

Return values

Boolean result: result = intensity level of the pixel at the specified coordinate position.

Examples:

```
//Get the intensity level of the first blob centroid
int x,y,b1_intensity;
x=blobSensor.BlobPosition.X[1];
y=blobSensor.BlobPosition.Y[1];
b1_intensity=Image(x,y);
```

Notes

This command retrieves the intensity level of a single pixel every time it is called.

MarkImage(, ,)

The MarkImage(, ,) function allows for the marking of single pixels of the image sent to the Sampled image display. Any pixel in the image can be marked with a corresponding intensity value between 0 and 255 except for the values from 1 through 7 which represent the following specific colors:

Intensity Value	Color	Intensity Value	Color
1	Dark Blue	5	Light Blue
2	Red	6	Yellow
3	Purple	7	Brown
4	Green		

Syntax

```
MarkImage(int row, int col, int value);
```

Arguments:

Int row: indicates the X-coordinate of the pixel to be marked.

Int col: indicates the Y-coordinate of the pixel to be marked.

Int value: represents the intensity level to use.

Return values

Boolean result: result = true for success, result = false for errors (function called from a background script or index out of range)

Examples:

```
//Mark the position of the first blob centroid with a yellow dot
int x,y,v;
x=blobSensor.BlobPosition.X[1];
y=blobSensor.BlobPosition.Y[1];
v=6;
MarkImage(x,y,v);
```

Notes

This command marks a single pixel every time it is called.

MarkImage(, , , ,)

The MarkImage(, , , ,) function allows for the marking of a line of pixels in the image sent to the Sampled image display. The line can be marked with a corresponding intensity value between 0 and 255 except for the values from 1 through 7 which represent the following specific colors:

Intensity Value	Color	Intensity Value	Color
1	Dark Blue	5	Light Blue
2	Red	6	Yellow
3	Purple	7	Brown
4	Green		

Syntax

```
MarkImage(int x0, int y0, int x1, int y1, int value);
```

Arguments:

Int x0: indicates the X-coordinate of the starting point of the line.

Int y0: indicates the Y-coordinate of the starting point of the line.

Int x1: indicates the X-coordinate of the ending point of the line.

Int y1: indicates the Y-coordinate of the ending point of the line.

Int value: represents the intensity level to use.

Return values

Boolean result: result = true for success, result = false for errors (function called from a background script or index out of range)

Examples:

```
//Draw a line from the center of blob 1 to the center
//of blob 2 (blobs found by the blobSensor SoftSensor)in red
int x0,x1,y0,y1;
int value = 2;
x0=blobSensor.BlobPosition.X[1];
y0=blobSensor.BlobPosition.Y[1];
x1=blobSensor.BlobPosition.X[2];
y1=blobSensor.BlobPosition.Y[2];
MarkImage(x0,y0,x1,y1,value);
```

Notes

This command marks a single line every time it is called.

MarkImage(, , [ ], [ ],)

The MarkImage(, , [ ], [ ],) function allows for the marking of a polygonal line of pixel in the image sent to the Sampled image display. The line, which is given by a set of corner points, can be marked with a corresponding intensity value between 0 and 255 except for the values from 1 through 7 which represent the following specific colors:

Intensity Value	Color	Intensity Value	Color
1	Dark Blue	5	Light Blue
2	Red	6	Yellow
3	Purple	7	Brown
4	Green		

The coordinates of the corner points (given by the arrays) are relative to the position indicated by the first two arguments (x and y coordinates of an offset). This allows users to move the entire marking by simply changing the x and y values of the offset. If the coordinates of the offset are (0,0) the coordinates in the array are absolute (actual pixel coordinates).

Syntax

```
MarkImage(int x0, int y0, int x[], int y[], int value);
```

Arguments:

Int x0: indicates the X-coordinate of the point from where the coordinates in the array of points are measured.

Int y0: indicates the Y-coordinate of the point from where the coordinates in the array of points are measured.

Int x[ ]: array of X-coordinate of the corner points of the line. These coordinates are relative to the point with coordinates (x0, y0).

Int y[ ]: array of Y-coordinate of the corner points of the line. These coordinates are relative to the point with coordinates (x0, y0).

Int value: represents the intensity level to use.

Return values

Boolean result: result = true for success, result = false for errors (function called from a background script or index out of range)

Examples:

```
//Draw a red line through the center of the first four blobs
//found by the blobSensor SoftSensor.
int x0 = 0,y0 = 0, x[], y[],value = 2;
x = new int[4];
y = new int[4];
for(int i = 1; i<5; i=i+1)
{
      x[i] = blobSensor.BlobPosition.X[i];
      y[i] = blobSensor.BlobPosition.Y[i];
}
MarkImage(x0,y0,x,y,value); //the
```

Notes

This command marks a single polygonal line every time it is called.

If the sizes of the arrays are different, the function uses the size of the smallest array ignoring any extra data from the longer array.

MarkImagePoints(, , [ ], [ ],)

The MarkImage(, , [ ], [ ],) function allows for the marking of a number of pixels in the image sent to the Sampled image display. Each pixel, determined by its X and Y coordinates, can be marked with a corresponding intensity value between 0 and 255 except for the values from 1 through 7 which represent the following specific colors:

Intensity Value	Color	Intensity Value	Color
1	Dark Blue	5	Light Blue
2	Red	6	Yellow
3	Purple	7	Brown
4	Green		

The coordinates of the corner points (given by the arrays) are relative to the position indicated by the first two arguments (x and y coordinates of an offset). This allows users to move the entire marking by simply changing the x and y values of the offset. If the coordinates of the offset are (0,0) the coordinates in the array are absolute (actual pixel coordinates).

Syntax

```
MarkImagePoints(int x0, int y0, int x[], int y[], int value);
```

Arguments:

Int x0: indicates the X-coordinate of the point from where the coordinates in the array of points are measured.

Int y0: indicates the Y-coordinate of the point from where the coordinates in the array of points are measured.

Int x[ ]: array of X-coordinate of the pixels to be marked. These coordinates are relative to the point with coordinates (x0, y0).

Int y[ ]: array of Y-coordinate of the pixels to be marked. These coordinates are relative to the point with coordinates (x0, y0).

Int value: represents the intensity level to use.

Return values

Boolean result: result = true for success, result = false for errors (function called from a background script or index out of range)

Examples:

```
//Mark a red dot in the center of four blobs
int x0 = 0,y0 = 0, x[], y[],value = 2;
x = new int[4];
y = new int[4];
for(int i = 1; i<5; i=i+1)
{
      x[i] = blobSensor.BlobPosition.X[i];
      y[i] = blobSensor.BlobPosition.Y[i];
}
MarkImagePoints(0,0,x,y,value);
```

Notes

This command marks a set of pixels every time it is called.

If the size of the arrays are different, the function uses the size of the smallest array ignoring any extra data from the longer array.

Window Object

The Window Object is used to pass as an argument to several methods of the image object. A Window Object defines a reduced area of the image. When an Image Object is used, some methods can be called to act on the image or just a reduced area of it. When the option is a reduced area of the image, a Window Object is passed as an argument to specify the area to be affected by the Image Object method.

Window()

This method is used to initialize the Window Object

Syntax

```
MyWinObject = new Window(int x1, int y1, int x2, int y2);
```

Arguments

Int x1: the x-coordinate of the upper left corner of the window

Int y1: the y-coordinate of the upper left corner of the window

Int x2: the x-coordinate of the lower right corner of the window

Int y2: the y-coordinate of the lower right corner of the window

Examples

Create a window from pixel (100,100) to pixel (540,380)

```
//first method: using variables. This is the best method
//regarding good programming practice
int x1, x2, y1, y2;
x1 = 100;
y1 = 100;
x2 = 540;
y2 = 380;
Window MyWin;
MyWin = new Window(x1, y1, x2, y2);

//second method:declare and initialize the window
//with hard coded values
Window new MyWindow;
MyWindow = Window(100,100,540,380);
```

Note

The sole purpose of this object is to pass the region of the image as an argument to Image Object functions.

Image Object

The Image Object is used in background scripts to give the user more control over the inspection application. It can be used to control the flow of an inspection

preprocess the image before performing an inspection

perform multiple inspections on the same image

acquire and perform multiple inspections to achieve on overall result.

Since the inspection flow is controlled in the background script the user has a high degree of flexibility in what can be achieved. However, with this flexibility come some complexities. The user must control some or all of the elements associated with an inspection in the Background Script. This may include monitoring an input as a trigger, setting outputs manually from Background Script, using registers to pass data between the inspection and the background script, etc.

Focusing just on the image object and its methods, the steps for using an image object are

Declare the image object

Construct the image object

Set the properties of the image object

Acquire an image

Preprocess the image

Inspect the image with a product

Send the image to a Sample Image Display.

In its simplest form the code for using an image object looks like this. This piece of code assumes there is a product called "SampleProduct" in the system.

```
Image img;
Product P;
P=GetProduct("SampleProduct");
img = new Image();
img.ExposureTime=3000;
img.Acquire();
img.Inspect(P);
img.Save();
```

Since the above code would be executed in a background script, it would only get executed once when the script is started. In order to make it take multiple inspects we would need to include it in a loop. This is the basic use of the Image Object. This section discusses all the fields and methods of the Image Object. For more examples see the Examples chapter of this guide.

Image Object Fields

Image Objects have a number of fields associated with them. A field allows the user to monitor or in some cases change the parameters associated with the image. The available fields are:

ID: Read-Only. Integer value - unique id number for each image.

ExposureTime: int Exposure time in microseconds.

AntiBloomingEnabled: Boolean indicates whether anti-blooming is enabled.

IlluminationEnabled: Boolean indicates whether illumination is enabled during acquisition.

Gain: int digitizing gain, a number from 0 to 240. This internal gain number relates to the value in the user interface by the equation (user interface gain) = 255/(255 - internal gain).

AfterSyncDelay: int delay in milliseconds between the call to acquire and the beginning of acquire. Equivalent to the user interface parameter delay after trigger.

X1: int x coordinate of upper left corner of acquisition window.

Y1: int y coordinate of upper left corner of acquisition window.

X2: int x coordinate of lower right corner of acquisition window.

Y2: int y coordinate of lower right corner of acquisition window.

DigitizingRate: provides access to the different digitizing times allowed by SmartImage Sensors. Some SmartImage Sensors use CCDs that allow for different levels of digitizing times. When that is the case, there are three different digitizing times. Assigning a 1 to this parameter would choose the one that gives the highest image quality. Assigning a 7 would select the fastest one whereas assigning a 3 would select the standard one. The SmartImage Sensors with different digitizing times are the Legend 520, Legend 530, Legend 540, Legend IS (intelligent scanner) and the Legend SC (SpectroCam).

ActiveStrobes: this field allows detailed control of the strobe options from script. It is an int bitfield indicating the desired strobe option. Setting one, some or all of the first 5 bits indicates the option. The functionality of the bits is as follows:

Bit 0 (LSB)	Strobe1
Bit 1	Strobe2
Bit 2	Strobe3
Bit 3	Integrated Light Strobe
Bit 5	Integrated Light Continuous

Example:

```
Image img;
img = new Image();
String str;
int tmp;
DebugPrint("Digitizing Rate: "+img.DigitizingRate);
DebugPrint("Gain: "+ img.Gain);
DebugPrint("AfterSyncDelay: "+img.AfterSyncDelay);
DebugPrint("ExposureTime: "+img.ExposureTime);
//check illumination
if(img.IlluminationEnabled)
{
      DebugPrint("Illumination enabled");
}else
{
      DebugPrint("Illumination disabled");
}
//check antiblooming
if(img.AntiBloomingEnabled)
{
      DebugPrint("Antiblooming enabled!");
}else
{
      DebugPrint("Antiblooming disabled");
}
//change Window Parameters
img.X1 = 100;
img.Y1 = 100;
img.X2 = 300;
img.Y2 = 300;
```

```
DebugPrint("Window corners (100,100,300,300): " + img.X1 + ", "+
img.Y1 + ", "+ img.X2 + ", "+ img.Y2);
//check and change digitizing rate
img.DigitizingRate = 3;
DebugPrint("DigTime: " + img.DigitizingRate);
//set bits 0 and 2 of the ActiveStrobes field
img.ActiveStrobes = (1 << 0) | (1 << 2);
```

Image()

This is the constructor and must be called to create and initialize the object. At this point all the image acquisition parameters are initialized with the parameters corresponding to the 'none' product. However, there is NO association between the image object and any product.

Syntax

```
MyImageObject = new Image();
```

Arguments:

None

Return values

A reference to the image object or null if used in Foreground Scripts.

Examples:

```
Image MyImg;
MyImg = new Image();
```

Acquire()

This method causes the system to acquire an image with the image acquisition parameters currently contained in the image object. This does NOT perform an inspection. The acquired image can be then be accessed and manipulated in the system. This method should only be called when the system is in external trigger.

Syntax

```
MyImg.Acquire();
```

Arguments:

None

Return values

Int result: result=0 for success, Result<0 for error

Error codes

Code -16: Indicates that the image is already in the Save Queue waiting to be sent out, so another image cannot be acquired in this image object (JIMG_IN_SAVED_Q).

Code -18: problem acquiring an image (JIMG_ACQUISITION_ERROR)

Examples:

```
int result;
Image MyImg;
MyImg = new Image();
result = MyImg.Acquire();
```

Save()

This method causes the system to place the image contained in the Image object into the save queue. This queue contains the images to be sent out the Background channels for display in UI, SmartLink and ActiveX. The image will be sent out when the system has time. If the save queue is full, then the image is not sent. After a call to this method the image object can be used to acquire another image because the image object is released.

Syntax

```
MyImg.Save();
```

Arguments:

None

Return values

Int result: result=0 for success, Result<0 for error

Error codes

Code -11: Image has not been acquired yet (JIMAGE_INVALID_PARAMETER)

Code -12: No more memory for images (JIMAGE_OUT_OF_MEMORY)

Code -13: Imaging error or resource conflict (JIMAGE_NO_IMAGE)

Code -16: Indicates that the image is already in the Save Queue waiting to be sent out, so another image cannot be acquired in this image object (JIMG_IN_SAVED_Q).

Code -17: Image not available (JIMAGE_IN_CAPTURE_QUEUE)

Code -18: problem acquiring an image (JIMG_ACQUISITION_ERROR)

Examples:

```
int result;
Image MyImg;
MyImg = new Image();
result = MyImg.Acquire();
if (result == 0)
{
      MyImg.Save();
}
```

Negate()

This method can be used to negate the whole image before it is inspected. When the image is negated, the intensity values of every pixel are irreversibly changed. The new intensity value of each pixel is determined by subtracting its current intensity from 255. The resulting image is a negative of the original.

Syntax

```
MyImg.Negate();
```

Arguments:

None

Return values

Int result: result=0 for success, Result<0 for error

Error codes

Code -11: Image has not been acquired yet (JIMAGE_INVALID_PARAMETER)

Code -16: Indicates that the image is already in the Save Queue waiting to be sent out, so another image cannot be acquired in this image object (JIMG_IN_SAVED_Q).

Example:

```
int result;
Image MyImg;
MyImg = new Image();
result = MyImg.Acquire();
if(result >= 0)
{
      MyImg.Negate();
}
```

Negate(Window)

This method can be used to negate a specific window of the image before it is inspected. When the window is negated, the intensity values of every pixel inside that window are irreversibly changed. The new intensity value of each pixel is determined by subtracting its current intensity from 255. The resulting image is a negative of the original.

Syntax

```
MyImg.Negate(Window Win);
```

Arguments:

Window Win: a Window Object specifying the window to be used for the operation.

Return values

Int result: result=0 for success, Result<0 for error

Error codes

Code -11: Image has not been acquired yet (JIMAGE_INVALID_PARAMETER)

Example:

```
int result;
Image MyImg;
MyImg = new Image();
result = MyImg.Acquire();
Window Win;
Win = new Window(50,125,250,375);
if(result >= 0)
{
      MyImg.Negate(Win);
}
```

Add()

This method can be used to add two images before they are inspected. When two images are added, the intensity level of every pixel in one image is added to the intensity level of the corresponding pixel in the other image. When the resulting intensity value exceeds 255 it is set to 255.

Syntax

```
MyImg.Add(Image AnotherImage);
```

Arguments:

Image AnotherImage: Image to be added to the image MyImg.

Return values

Int result: result=0 for success, Result<0 for error

Error codes

Code -11: Image has not been acquired yet (JIMAGE_INVALID_PARAMETER)

Example:

```
int result;

Image MyImg;
MyImg = new Image();
result = MyImg.Acquire();

Image AnotherImage;
AnotherImage = new Image();
result = AnotherImage.Acquire();

result = MyImg.Add(AnotherImage);
```

Add(,Window)

This method can be used to add specific areas of two images before they are inspected. When two images are added, the intensity level of every pixel in one image is added to the intensity level of the corresponding pixel in the other image. When the resulting intensity value exceeds 255 it is set to 255.

Syntax

```
MyImg.Add(Image AnotherImage, Window Win);
```

Arguments:

Image AnotherImage: Image to be added to the image MyImg.

Window Win: the actual region of the image that is added

Return values

Int result: result=0 for success, Result<0 for error

Error codes

Code -11: Image has not been acquired yet (JIMAGE_INVALID_PARAMETER)

Example:

```
int result;

Image MyImg;
MyImg = new Image();
result = MyImg.Acquire();

Image AnotherImage;
AnotherImage = new Image();
result = AnotherImage.Acquire();

Window Win;
Win = new Window(100,100,250,360);

result = MyImg.Add(AnotherImage, Win);
```

Subtract()

This method can be used to subtract one image from another over the entire image window. The intensity values are irreversibly changed in the image from which another one is being subtracted. The new intensity value of each pixel in the base image is determined by subtracting the intensity of the corresponding pixel in the second image from its current intensity. If the resulting value is negative it is clamped at 0. The intensity values in the second image (image being subtracted) are unchanged.

Syntax

```
MyImg.Subtract(Image AnotherImage);
```

Arguments:

Image AnotherImage: Image to be subtracted from the image MyImg.

Return values

Int result: result=0 for success, Result<0 for error

Error codes

Code -11: Image has not been acquired yet (JIMAGE_INVALID_PARAMETER)

Example:

```
int result;

Image MyImg;
MyImg = new Image();
result = MyImg.Acquire();

Image AnotherImage;
AnotherImage = new Image();
result = AnotherImage.Acquire();

result = MyImg.Subtract(AnotherImage);
```

Subtract(,Window)

This method can be used to subtract one image from another over a specific area instead of the entire image window. The intensity values in the affected region are irreversibly changed in the image from which another one is being subtracted. The new intensity value of each pixel in the base image is determined by subtracting the intensity of the corresponding pixel in the second image from its current intensity. If the resulting value is negative it is clamped at 0. The intensity values in the second image (image being subtracted) are unchanged.

Syntax

```
MyImg.Subtract(Image AnotherImage, Window Win);
```

Arguments:

Image AnotherImage: Image to be subtracted from the image MyImg.

Window Win: the actual region of the image that is added

Return values

Int result: result=0 for success, Result<0 for error

Error codes

Code -11: Image has not been acquired yet (JIMAGE_INVALID_PARAMETER)

Example:

```
int result;

Image MyImg;
MyImg = new Image();
result = MyImg.Acquire();

Image AnotherImage;
AnotherImage = new Image();
result = AnotherImage.Acquire();

Window Win;
Win = new Window(100,100,250,360);

result = MyImg.Subtract(AnotherImage, Win);
```

Erode(, , ,)

This method can be used to binarize and subsequently erode light areas of the whole image before inspection. The intensity values are irreversibly changed. Before erosion the image is first binarized according to a threshold. Based on that, the Erosion operation is applied (a certain number of pixels is removed from the edges of light objects to shrink their size).

Syntax

```
MyImg.Erode(int size, int threshold, int low, int high);
```

Arguments:

Int size: the number of pixels for the Erosion operation (the number of pixels taken away from the edges of light objects)

Int threshold: intensity level used to binarize the image. Pixels with intensity levels above this will be classified as light pixels, those with intensity level below this will be classified as dark pixels. Acceptable values range from 0 to 255.

Int low: After applying the threshold, the image is divided in two groups of pixels: dark and light pixels. Dark pixels are assigned this intensity level. This is like grayscale image marking but the user chooses the marking grayscale values. Acceptable values range from 0 to 255.

Int high: After applying the threshold, the image is divided in two groups of pixels: dark and light pixels. Light pixels are assigned this intensity level. This is like grayscale image marking but the user chooses the marking grayscale values. Acceptable values range from 0 to 255.

Return values

Int result: result=0 for success, Result<0 for error

Error codes

Code -11: Image has not been acquired yet (JIMAGE_INVALID_PARAMETER)

Example:

```
int result;

Image MyImg;
MyImg = new Image();
result = MyImg.Acquire();
//binarize the image with a threshold level of 125, then mark
```

```
//light pixels in white (intensity value of 255) and dark pixels
//in light gray (intensity level of 150. After that Erode the
//light pixel areas 5 pixels.
result = MyImg.Erode(5, 125, 150, 255);
```

Notes

This definition of Erosion is slightly different from the one used in blob tools. In this case the Erosion is always applied to light areas, so if we wanted to expand dark objects we should still use Erosion because it will shrink light areas thus expanding dark areas.

Erode(Window, , , ,)

This method can be used to binarize and subsequently erode light areas of a specific window of the image before inspection. The intensity values are irreversibly changed. Before erosion the window is first binarized according to a threshold. Based on that, the Erosion operation is applied (a certain number of pixels is removed from the edges of light objects to shrink their size).

Syntax

```
MyImg.Erode(Window win, int size, int threshold, int low, int
high);
```

Arguments:

Window Win: a window object specifying the region of the image where to apply the erosion.

Int size: the number of pixels for the Erosion operation (the number of pixels taken away from the edges of light objects)

Int threshold: intensity level used to binarize the image. Pixels with intensity levels above this will be classified as light pixels, those with intensity level below this will be classified as dark pixels. Acceptable values range from 0 to 255.

Int low: After applying the threshold, the image is divided in two groups of pixels: dark and light pixels. Dark pixels are assigned this intensity level. This is like grayscale image marking but the user chooses the marking grayscale values. Acceptable values range from 0 to 255.

Int high: After applying the threshold, the image is divided in two groups of pixels: dark and light pixels. Light pixels are assigned this intensity level. This is like grayscale image marking but the user chooses the marking grayscale values. Acceptable values range from 0 to 255.

Return values

Int result: result=0 for success, Result<0 for error

Error codes

Code -11: Image has not been acquired yet (JIMAGE_INVALID_PARAMETER)

Example:

```
int result;

Image MyImg;
MyImg = new Image();
result = MyImg.Acquire();

Window Win;
Win = new Window(150, 150, 360, 220);
//binarize the image with a threshold level of 125, then mark
```

```
//light pixels in white (intensity value of 255) and dark pixels
//in light gray (intensity level of 150. After that Erode the
//light pixel areas 5 pixels.
result = MyImg.Erode(Win, 5, 125, 150, 255);
```

Notes

This definition of Erosion is slightly different from the one used in blob tools. In this case the Erosion is always applied to light areas, so if we wanted to expand dark objects we should still use Erosion because it will shrink light areas thus expanding dark areas.

Dilate(, , ,)

This method can be used to binarize and subsequently dilate light areas of the whole image before inspection. The intensity values are irreversibly changed. Before dilation the image is first binarized according to a threshold. Based on that, the dilation operation is applied (a certain number of pixels is added to the edges of light objects to expand their size).

Syntax

```
MyImg.Dilate(int size, int threshold, int low, int high);
```

Arguments:

Int size: the number of pixels for the Dilation operation (the number of pixels added to the edges of light objects)

Int threshold: intensity level used to binarize the image. Pixels with intensity levels above this will be classified as light pixels, those with intensity level below this will be classified as dark pixels. Acceptable values range from 0 to 255.

Int low: After applying the threshold, the image is divided in two groups of pixels: dark and light pixels. Dark pixels are assigned this intensity level. This is like grayscale image marking but the user chooses the marking grayscale values. Acceptable values range from 0 to 255.

Int high: After applying the threshold, the image is divided in two groups of pixels: dark and light pixels. Light pixels are assigned this intensity level. This is like grayscale image marking but the user chooses the marking grayscale values. Acceptable values range from 0 to 255.

Return values

Int result: result=0 for success, Result<0 for error

Error codes

Code -11: Image has not been acquired yet (JIMAGE_INVALID_PARAMETER)

Example:

```
int result;

Image MyImg;
MyImg = new Image();
result = MyImg.Acquire();
//binarize the image with a threshold level of 125, then mark
//light pixels in white (intensity value of 255) and dark pixels
//in light gray (intensity level of 150. After that Dilate the
//light pixel areas 5 pixels.
result = MyImg.Dilate(5, 125, 150, 255);
```

Notes

This definition of Dilation is slightly different from the one used in blob tools. In this case the Dilation is always applied to light areas, so if we wanted to shrink dark objects we should still use Dilation because it will expand light areas thus shrinking dark areas.

Dilate(Window, , , ,)

This method can be used to binarize and subsequently dilate light areas of a specific area of the image before inspection. The intensity values in that area are irreversibly changed. Before dilation the area of the image is first binarized according to a threshold. Based on that, the dilation operation is applied (a certain number of pixels is added to the edges of light objects to expand their size).

Syntax

```
MyImg.Dilate(Window Win, int size, int threshold, int low, int
high);
```

Arguments:

Window Win: window object specifying the area where the operation is to be applied.

Int size: the number of pixels for the Dilation operation (the number of pixels added to the edges of light objects)

Int threshold: intensity level used to binarize the image. Pixels with intensity levels above this will be classified as light pixels, those with intensity level below this will be classified as dark pixels. Acceptable values range from 0 to 255.

Int low: After applying the threshold, the image is divided in two groups of pixels: dark and light pixels. Dark pixels are assigned this intensity level. This is like grayscale image marking but the user chooses the marking grayscale values. Acceptable values range from 0 to 255.

Int high: After applying the threshold, the image is divided in two groups of pixels: dark and light pixels. Light pixels are assigned this intensity level. This is like grayscale image marking but the user chooses the marking grayscale values. Acceptable values range from 0 to 255.

Return values

Int result: result=0 for success, Result<0 for error

Error codes

Code -11: Image has not been acquired yet (JIMAGE_INVALID_PARAMETER)

Example:

```
int result;

Image MyImg;
MyImg = new Image();
result = MyImg.Acquire();

Window Win;
Win = new Window(50,50,260,380);
//binarize the image with a threshold level of 125, then mark
//light pixels in white (intensity value of 255) and dark pixels
//in light gray (intensity level of 150. After that Dilate the
//light pixel areas 5 pixels.
result = MyImg.Dilate(Win,5, 125, 150, 255);
```

Notes

This definition of Dilation is slightly different from the one used in blob tools. In this case the Dilation is always applied to light areas, so if we wanted to shrink dark objects we should still use Dilation because it will expand light areas thus shrinking dark areas.

Filter(double filter[ ])

This method can be used to apply a 3X3 convolution filter to the whole image before inspection. The intensity values are irreversibly changed. A filter is an array with nine values that correspond to the coefficients of a user-defined filter. These coefficients are limited to a range of ± 1024.0 for speed of execution. In this case, the filter coefficients are floating point numbers (doubles).

Syntax

```
MyImg.Filter(double myFilter[]);
```

Arguments:

Double myFilter[ ]: array containing the coefficients for the convolution filter.

Return values

Int result: result=0 for success, Result<0 for error

Error codes

Code -11: Image has not been acquired yet (JIMAGE_INVALID_PARAMETER)

Example:

```
int result;

//create an edge enhancement filter
//the 3x3 matrix defining the filter should look like this:
//    |-2.5 -2.5  -2.5|
//    |-2.5  21         -2.5|
//    |-2.5 -2.5  -2.5|
//so we need to create an array of 9 elements and populate
//it with a 1/9 in every cell.
double MyFilter[] = new double[9];
//Create filter - Laplace Edge enhancement
MyFilter[1] = -2.5;
MyFilter[2] = -2.5;
MyFilter[3] = -2.5;
MyFilter[4] = -2.5;
MyFilter[5] = 21;
MyFilter[6] = -2.5;
MyFilter[7] = -2.5;
MyFilter[8] = -2.5;
MyFilter[9] = -2.5;


DebugPrint("Cell 5= " + MyFilter[5]);
Image MyImg;
MyImg = new Image();
result = MyImg.Acquire();

//filter the image now
result = MyImg.Filter(MyFilter);
```

Notes

In order to translate a 1D array into a 2D matrix, the elements are taken in order from the array to populate the first row of the matrix, then the second row and then the third row.

Filter(int filter[ ])

This method can be used to apply a 3X3 convolution filter to the whole image before inspection. The intensity values are irreversibly changed. A filter is an array with nine values that correspond to the coefficients of a user-defined filter. These coefficients are limited to a range of ± 1024.0 for speed of execution. In this case, the filter coefficients are integers.

Syntax

```
MyImg.Filter(int myFilter[]);
```

Arguments:

Int myFilter[ ]: array containing the coefficients for the convolution filter.

Return values

Int result: result=0 for success, Result<0 for error

Error codes

Code -11: Image has not been acquired yet (JIMAGE_INVALID_PARAMETER)

Example:

```
int result;

//create a Laplace Edge enhancement filter
//the 3x3 matrix defining the filter should look like this:
//    |-2   0     -2|
//    |0    8     0|
//    |-2   0     -2|
//so we need to create an array of 9 elements and populate
//it with the corresponding coefficients.
int MyFilter[] = new int[9];
//Create filter - Laplace Edge enhancement
MyFilter[1] = -2;
MyFilter[2] = 0;
MyFilter[3] = -2;
MyFilter[4] = 0;
MyFilter[5] = 8;
MyFilter[6] = 0;
MyFilter[7] = -2;
MyFilter[8] = 0;
MyFilter[9] = -2;

Image MyImg;
MyImg = new Image();
result = MyImg.Acquire();

//filter the image now
result = MyImg.Filter(MyFilter);
```

Notes

In order to translate a 1D array into a 2D matrix, the elements are taken in order from the array to populate the first row of the matrix, then the second row and then the third row.

Filter(Window, double filter[ ])

This method can be used to apply a 3X3 convolution filter to a specific area of the image before inspection. The intensity values are irreversibly changed. A filter is an array with nine values that correspond to the coefficients of a user-defined filter. These coefficients are limited to a range of ± 1024.0 for speed of execution.

Syntax

```
MyImg.Filter(Window Win, double myFilter[]);
```

Arguments:

Window Win: window object specifying the area of the image where the operation is to take place.

Double myFilter[ ]: array containing the coefficients for the convolution filter.

Return values

Int result: result=0 for success, Result<0 for error

Error codes

Code -11: Image has not been acquired yet (JIMAGE_INVALID_PARAMETER)

Example:

```
int result;

//create an edge enhancement filter
//the 3x3 matrix defining the filter should look like this:
//    |-2.5 -2.5  -2.5|
//    |-2.5  21         -2.5|
//    |-2.5 -2.5  -2.5|
//so we need to create an array of 9 elements and populate
//it with a 1/9 in every cell.
double MyFilter[] = new double[9];
//Create filter - Laplace Edge enhancement
MyFilter[1] = -2.5;
MyFilter[2] = -2.5;
MyFilter[3] = -2.5;
MyFilter[4] = -2.5;
MyFilter[5] = 21;
MyFilter[6] = -2.5;
MyFilter[7] = -2.5;
MyFilter[8] = -2.5;
MyFilter[9] = -2.5;

Image MyImg;
MyImg = new Image();
result = MyImg.Acquire();

Window Win;
Win = new Window(150,150,360,220);
//filter the image now
result = MyImg.Filter(Win, MyFilter);
```

Notes

In order to translate a 1D array into a 2D matrix, the elements are taken in order from the array to populate the first row of the matrix, then the second row and then the third row.

Filter(Window, int filter[ ])

This method can be used to apply a 3X3 convolution filter to a specific area of the image before inspection. The intensity values are irreversibly changed. A filter is an array with nine values that correspond to the coefficients of a user-defined filter. In this case the filter coefficients are integer values.

Syntax

```
MyImg.Filter(Window Win, int myFilter[]);
```

Arguments:

Window Win: window object specifying the area of the image where the operation is to take place.

Int myFilter[ ]: array containing the coefficients for the convolution filter.

Return values

Int result: result=0 for success, Result<0 for error

Error codes

Code -11: Image has not been acquired yet (JIMAGE_INVALID_PARAMETER)

Example:

```
int result;

//create a Laplace Edge enhancement filter
//the 3x3 matrix defining the filter should look like this:
//    |-2   0     -2|
//    |0    8     0|
//    |-2   0     -2|
//so we need to create an array of 9 elements and populate
//it with the corresponding coefficients.
int MyFilter[] = new int[9];
//Create filter - Laplace Edge enhancement
MyFilter[1] = -2;
MyFilter[2] = 0;
MyFilter[3] = -2;
MyFilter[4] = 0;
MyFilter[5] = 8;
MyFilter[6] = 0;
MyFilter[7] = -2;
MyFilter[8] = 0;
MyFilter[9] = -2;

Image MyImg;
MyImg = new Image();
result = MyImg.Acquire();

Window Win;
Win = new Window(150,150,360,220);
//filter the image now
result = MyImg.Filter(Win, MyFilter);
```

Notes

In order to translate a 1D array into a 2D matrix, the elements are taken in order from the array to populate the first row of the matrix, then the second row and then the third row.

Map()

This method can be used to perform preprocessing on the whole image before it is inspected. The intensity values are irreversibly changed according to a passed-in intensity Map. The new intensity value of each pixel is determined by performing a look-up into the map array using the current intensity value as an index. That is, the argument is an array with 256 cells. Each index in the array corresponds to a current intensity level in the image. The content of each cell corresponds to what the new intensity level should be.

Syntax

```
MyImg.Map(byte myMap[]);
```

Arguments:

Byte myMap: 256-cell array containing the new intensity levels.

Return values

Int result: result=0 for success, Result<0 for error

Error codes

Code -11: Image has not been acquired yet (JIMAGE_INVALID_PARAMETER)

Examples:

```
//declare array
byte MyMap[] = new byte[256];
int i;
//initialize array: first 240 intensity levels remain unchanged
for(i = 0; i < 240; i=i+1)
MyMap[i+1] = i;
//the last 15 leves are set to 125 (used to minimize glare)
for( ; i<= 255 ; i=i+1)
MyMap[i+1] = 125;

int result;
Image MyImg;
MyImg = new Image();
result = MyImg.Acquire();

MyImg.Map(MyMap);  // changes all values with intensity
                  // above 240 to 125;
```

Map(, Window)

This method can be used to perform preprocessing on a specific area of the image before it is inspected. The intensity values are irreversibly changed according to a passed-in intensity Map. The new intensity value of each pixel is determined by performing a look-up into the map array using the current intensity value as an index. That is, the argument is an array with 256 cells. Each index in the array corresponds to a current intensity level in the image. The content of each cell corresponds to what the new intensity level should be.

Syntax

```
MyImg.Map(byte myMap[], Window Win);
```

Arguments:

Byte myMap: 256-cell array containing the new intensity levels.

Window Win: window object specifying the area where the operation is to take place.

Return values

Int result: result=0 for success, Result<0 for error

Error codes

Code -11: Image has not been acquired yet (JIMAGE_INVALID_PARAMETER)

Examples:

```
//declare array
byte MyMap[] = new byte[256];
int i;
//initialize array: first 240 intensity levels remain unchanged
for(i = 0; i < 240; i=i+1)
MyMap[i+1] = i;
//the last 15 leves are set to 125 (used to minimize glare)
for( ; i<= 255 ; i=i+1)
MyMap[i+1] = 125;

Window Win;
Win = new Window(50,50,550,400);

int result;
Image MyImg;
MyImg = new Image();
result = MyImg.Acquire();



MyImg.Map(MyMap, Win);  // changes all values with intensity
                  // above 240 to 125;
```

Threshold(, , , ,)

This method can be used to threshold the whole image before it is inspected. The intensity values are irreversibly changed according to the parameters passed. The method allows for two different threshold levels, resulting in 3 different groups of pixels: above thresholds, between thresholds, and below thresholds. The threshold levels are specified using a level and a range. The upper threshold is the selected level plus the range, while the lower threshold is the selected level minus the range.

Syntax

```
MyImg.Threshold(int low, int high, int mid, int level, int
range);
```

Arguments:

Int low: the new intensity level assigned to pixels below thresholds.

Int high: the new intensity level assigned to pixels above thresholds.

Int mid: the new intensity level assigned to pixels between thresholds.

Int level: intensity level used to determine the thresholds

Int range: intensity range used to determine the thresholds.

Upper threshold: level + range

Lower threshold: level - range

Return values

Int result: result=0 for success, Result<0 for error

Error codes

Code -11: Image has not been acquired yet (JIMAGE_INVALID_PARAMETER)

Examples:

```
int result;
Image MyImg;
MyImg = new Image();
result = MyImg.Acquire();
//threshold the image at levels 100 and 200, this will require
//a level of 150 and a range of 50. Mark the resulting
//groups of pixels with intensity levels 10, 50, and 90
MyImg.Threshold(10,90,50,150,50);
```

Note

Intensity levels range from 0 (black) to 255 (white).

Threshold(Window, , , , ,)

This method can be used to threshold a specific area of the image before it is inspected. The intensity values in that area are irreversibly changed according to the parameters passed. The method allows for two different threshold levels, resulting in 3 different groups of pixels: above thresholds, between thresholds, and below thresholds. The threshold levels are specified using a level and a range. The upper threshold is the selected level plus the range, while the lower threshold is the selected level minus the range.

Syntax

```
MyImg.Threshold(Window Win, int low, int high, int mid, int
level, int range);
```

Arguments:

Window Win: window object specifying the area of the image where the operation is to be applied.

Int Low: the new intensity level assigned to pixels below thresholds.

Int high: the new intensity level assigned to pixels above thresholds.

Int mid: the new intensity level assigned to pixels between thresholds.

Int level: intensity level used to determine the thresholds

Int range: intensity range used to determine the thresholds.

Upper threshold: level + range

Lower threshold: level - range

Return values

Int result: result=0 for success, Result<0 for error

Error codes

Code -11: Image has not been acquired yet (JIMAGE_INVALID_PARAMETER)

Examples:

```
int result;
Image new MyImg;
MyImg = Image();
```

```
Window Win;
Win = new Window(50,50,550,400);

result = MyImg.Acquire();
//threshold the image at levels 100 and 200, this will require
//a level of 150 and a range of 50. Mark the resulting
//groups of pixels with intensity levels 10, 50, and 90
MyImg.Threshold(Win, 10,90,50,150,50);
```

Note

Intensity levels range from 0 (black) to 255 (white).

Inspect()

This method causes the system to Inspect the image using a predefined inspection product. This Method should only be called when the system is in external trigger.

Syntax

```
MyImg.Inspect(Product P);
```

Arguments:

Product P: product object representing the inspection product to be used.

Return values

Int result: result contains the inspection result as follows:

Result = -1 for PASS

16 <= result <= 31 for FAIL

32 <= result <= 47 for WARN

otherwise it returns an error code

Error codes

Code -11: image not acquired (JIMAGE_INVALID_PARAMETER)

Code -16: cannot inspect an image sent to the save queue (JIMAGE_IN_SAVED_QUEUE)

Examples:

```
//inspect the same image with 2 different products
int result;
Image MyImg;
Product P1;
Product P2;
MyImg = new Image();
MyImg.Acquire();
P1 = GetProduct(“test”);
P2 = GetProduct(“test2”);
result = MyImg.Inspect(P1);
sleep(250);//pause for 250 msec to read outputs
result = MyImg.Inspect(P2);
```

SetWindow(Window win)

This function is used to set a new acquisition window for a particular image object

Sytnax

```
MyImageObject.SetWindow(Window Win);
```

Arguments

Window Win: a window object that needs to be previously initialized

Return values

Int result: this value is returned only if the operation is successful, in which case its value is 0.

Example

```
Image Img;
Img = new Image();
Window Win;
Win = new Window(10,10,630,470);
Img.SetWindow(Win);
Img.Acquire();
```

GetWindow()

This function is used to access the acquisition window of a particular image object

Sytnax

```
MyImageObject.GetWindow();
```

Arguments

None

Return values

This method returns a window object when it is a successful operation, otherwise it returns null.

Example

```
Image Img;
Img = new Image();
Window Win;
Win = new Window(10,10,630,470);
Window MyWin;
MyWin = Img.GetWindow();
```

Clear()

This function is used to clear the image contained in the image object. An error will be raised unless the Acquire() Method is called first.

Sytnax

```
MyImageObject.Clear();
```

Arguments

None

Return values

Int result: result=0 for success, Result<0 for error

Error codes

Code -11: image not acquired (JIMAGE_INVALID_PARAMETER)

Example

```
Image img = new Image();
int res = img.Acquire();
res = img.Clear();
```

Copy()

This function is used to copy a region from one image object to another. An image must be acquired into both objects for this method to be successful.

Sytnax

```
MyImageObject.Copy(Image Src, Window win, int x0, int y0);
```

Arguments

Src: the image object we wish to copy from.

win: defines a window in Src that we wish to copy.

x0,y0: define the absolute position of the origin for the data in the destination image (where we want the data to be copied to).

Return values

Int result: result=0 for success, Result<0 for error

Error codes

Code -11: image not acquired (JIMAGE_INVALID_PARAMETER)

Example

```
Image img_dest = new Image();
Image img_src = new Image();
Window w = new Window(0,0,100,100);
int res = img_dest.Acquire();
res = img_dest.Clear();
res = img_src.Acquire();
res = img_dest.Copy(img_src,w,0,0);
```

Chapter 12 – Flash Object

The Flash Object is a recent addition to FrameWork that allows users to back up data to the SmartImage Sensor flash memory. Data saved to flash memory can be restored after the power is cycled. Common applications for this functionality are saving data from the registers to flash memory and restoring it on power-up, saving the last product used to flash memory and selecting it on power-up. This chapter describes the methods associated with the Flash Object and provides some basic examples on how to use them.

Flash()

This method is the constructor for the Flash Object. After the declaration of the Flash Object, this method must be called to properly initialize it.

Syntax

```
MyFlashObject = new Flash();
```

Return Values

This method returns a Flash object when the operation is successful, otherwise it returns null.

Example

```
Flash MyFlashObject;
MyFlashObject = new Flash();
```

SaveRegs()

This method is used to save a block of registers to flash memory so they are available when the power is cycled. Only one block of registers can be stored to flash. It is recommended that after restoring registers flash memory is reclaimed. To reclaim the used flash memory you must manually click on the Reclaim Flash button in the Product Management dialog.

Syntax

```
MyFlashObject.SaveRegs(int firstReg, int lastReg);
```

Arguments

Int firstReg: integer value indicating the first register of the block to be saved.

Int lastReg: integer value indicating the last register of the block to be saved.

Return values

int Result. Result=0 for success, Result<0 for error.

Example

```
Flash F;
F = new Flash();
//save 51 registers to flash
F.SaveRegs(100,150);
```

Error codes

Code -11: Invalid parameters (SCRIPT_FLASH_ERROR)

Notes

Every time that SaveRegisters is used, blocks of 512 bytes are used. Thus, even if only a few registers need to be saved, the system will still use 512 bytes of flash memory. This increases significantly the amount of flash memory being used.

Flash memory has a limited lifetime depending on the number of times it has been used and reclaimed, so these features should be used with caution.

RestoreRegs ()

This method is used to restore a block of registers that have been saved to flash memory. This method would be used in a background script that is configured to run at startup to restore a block of registers that were previously stored with the SaveRegs Method. In this case it is not necessary to indicate to which registers to restore the data, the system already know it. It is recommended that after restoring registers flash memory is reclaimed. To reclaim the used flash memory you must manually click on the Reclaim Flash button in the Product Management dialog.

Syntax

```
MyFlashObject.RestoreRegs();
```

Arguments

None.

Return values

int Result. Result=0 for success, Result<0 for error.

Error codes

Code -11: Corrupt data (SCRIPT_FLASH_ERROR)

Example

```
//The following background script when run at power up
//will restore the registers saved by the SaveRegs Method
//before the system was powered down
Flash f;
f = new Flash();
f.RestoreRegs();
```

Notes

Every time that SaveRegisters is used, blocks of 512 bytes are used. Thus, even if only a few registers need to be saved, the system will still use 512 bytes of flash memory. This increases significantly the amount of flash memory being used.

Flash memory has a limited lifetime depending on the number of times it has been used and reclaimed, so these features should be used with caution.

Chapter 13 – Script Examples

This chapter contains complete and fully documented examples for users to utilize in their applications or simply practice some scripting.

```
class myScript
{
  public void inspect()
  {
    //this script will compute the total area
    //of the blobs. It needs to cycle through
    //all the blobs and get the area from each.

    int totalArea = 0;

    //establish a loop to cycle through all the blobs
    //regardless of the number of blobs found.
    //This loop will be executed exactly once
    //per blob found.
    for(int i = 0; i < myBlobSelector.NumBlobs; i++)
    {
      totalArea += myBlobSelector.BlobArea[i];
    }

    //assign the value to the script string to display it
    //in the result table
    this.String = "Total Area = " + totalArea + " pixels.";
  }
}
```

Extracting Detailed Information from a Measurement SoftSensor.

```
class myScript
{
  public void inspect()
  {
    //this script reads the edge points found by a
    //measurement softsensor fitting a line along
    //a horizontal edge, and determines the position
    //of the top point of the edge.

    int topPtIndex = 0;

    //assign a large number to yTop to begin the
    //process of checking every point
    float xTop, yTop = 479;

    //cycle through all the points to find the one with
    //the lowest Y coordinate
    for(int i = 0; i < topEdge.NumEdgePoints; i++)
    {
      //only update data if the point is higher than
      //the highest one so far
      if(topEdge.EdgePoint.Y[i] < yTop)
      {
        topPtIndex = i;
        yTop = topEdge.EdgePoint.Y[i];
        xTop = topEdge.EdgePoint.X[i];
      }
    }

    if(topPtIndex == 0)
    {
      this.String = "Error, point not found";
    }
    else
    {
      this.String = "Top point: "+"("+xTop+","+yTop+")";
    }
  }
}
```

Image Object Example: Preprocessing an image

This Background Script example determines a number of windows and it performs different processing operations in those windows. Most of the functionality of the image object is used.

```
class myScript
{
  public static void main()
  {
    //this example creates 8 windows inside the image and
    //performs a different operation in each one to
    //illustrate the functionality of the image object

    while(true)
    {
      int res,i,time;
      //declare and construct 2 image objects
      Image img1;
      Image img2;
      img1 = new Image();
      img2 = new Image();

      //declare 8 Window objects, one for each operation
      Window WinAdd, WinSub, WinNeg, WinThres;
      Window WinMap, WinFilt, WinErode, WinDilate;

      //------- Define Windows -------------------------

      //construct the windows of specific sizes
      //the numbers were chosen to end up with
      //eight different well defined windows

      WinAdd = new Window(32,32,152,224);
      WinSub = new Window(184,32,304,224);
      WinNeg = new Window(336,32,456,224);
      WinThres = new Window(488,32,608,224);
      WinMap = new Window(32,256,152,448);
      WinFilt = new Window(184,256,304,448);
      WinErode = new Window(336,256,456,448);
      WinDilate = new Window(488,256,608,448);

      // ------ Acquire image 1 and debugprint timing ---

      DebugPrint("----------- ");
      time= clock();
      res=img1.Acquire();
      DebugPrint("Acquire: " + res + " time: " +(clock()-time));

      // ------ Acquire image 2 and debugprint timing ---

      time= clock();
      res=img2.Acquire();
      DebugPrint("Acquire: " + res + " time: " +(clock()-time));
      sleep(1000);//this simply pauses the execution
      // Add a specific window of image 2 to image 1 -----------

      time= clock();
      res = img1.Add(img2,WinAdd);
      DebugPrint("Addition: " + res + " time: " +(clock()-time));

      // Subtract a specific window of image 2 from image 1 ----

      time= clock();
      res = img1.Subtract(img2,WinSub);
      DebugPrint("Subtraction: "+res+" time: "+(clock()-time));

      // ------ Negate image1 specifying window ----------------

      time= clock();
      res = img1.Negate(WinNeg);
      DebugPrint("Negation:  "+res+" time: "+(clock()-time));

      // ------ Threshold image 1 specifying window ------------

      time= clock();
      res = img1.Threshold( WinThres, 50, 125, 200, 125, 25);
      DebugPrint("Thresholding: "+res+" time: "+(clock()-time));

      // ------ Map image 1 specifying window ------------------

      byte MyMap[] = new byte[256];
      for(i = 0; i < 125; i++)
        MyMap[i] = 255-i;
      for( ; i< 255 ; i++)
        MyMap[i] = 50;

      time= clock();
      res = img1.Map(MyMap, WinMap);
      DebugPrint("Mapping: " + res + " time: " +(clock()-time));

      // ------ Filter image 1 specifying window ---------------

      //Create filter - Laplace Edge enhancement
      int MyFilter[] = new int[9];
      MyFilter[0] = -2;
      MyFilter[1] = 0;
      MyFilter[2] = -2;
      MyFilter[3] = 0;
      MyFilter[4] = 4;
      MyFilter[5] = 0;
      MyFilter[6] = -2;
      MyFilter[7] = 0;
      MyFilter[8] = -2;
      time= clock();
      res = img1.Filter(WinFilt, MyFilter);
      DebugPrint("Filtering: "+res+" time: "+(clock()-time));

      // ------ Erode image 1 specifying window ----------------
      time= clock();
      res = img1.Erode( WinErode, 3, 125, 50, 200);
      DebugPrint("Erosion " + res + " time: " +(clock()-time));
```

```
      // ------ Dilate image 1 specifying window ---------------
      time= clock();
      res = img1.Dilate( WinDilate, 3, 125, 50, 200);
      DebugPrint("Dilation " + res + " time: " +(clock()-time));

      // ------ Send image out ---------------------------------
      time= clock();
      res = img1.Save();
      DebugPrint("Save " + res + " time: " +(clock()-time));
      DebugPrint("----------- ");

      sleep(1000);//pause execution to see images

    } // closes while(true)
  }
}
```

Establishing communications with Socket Object as a server

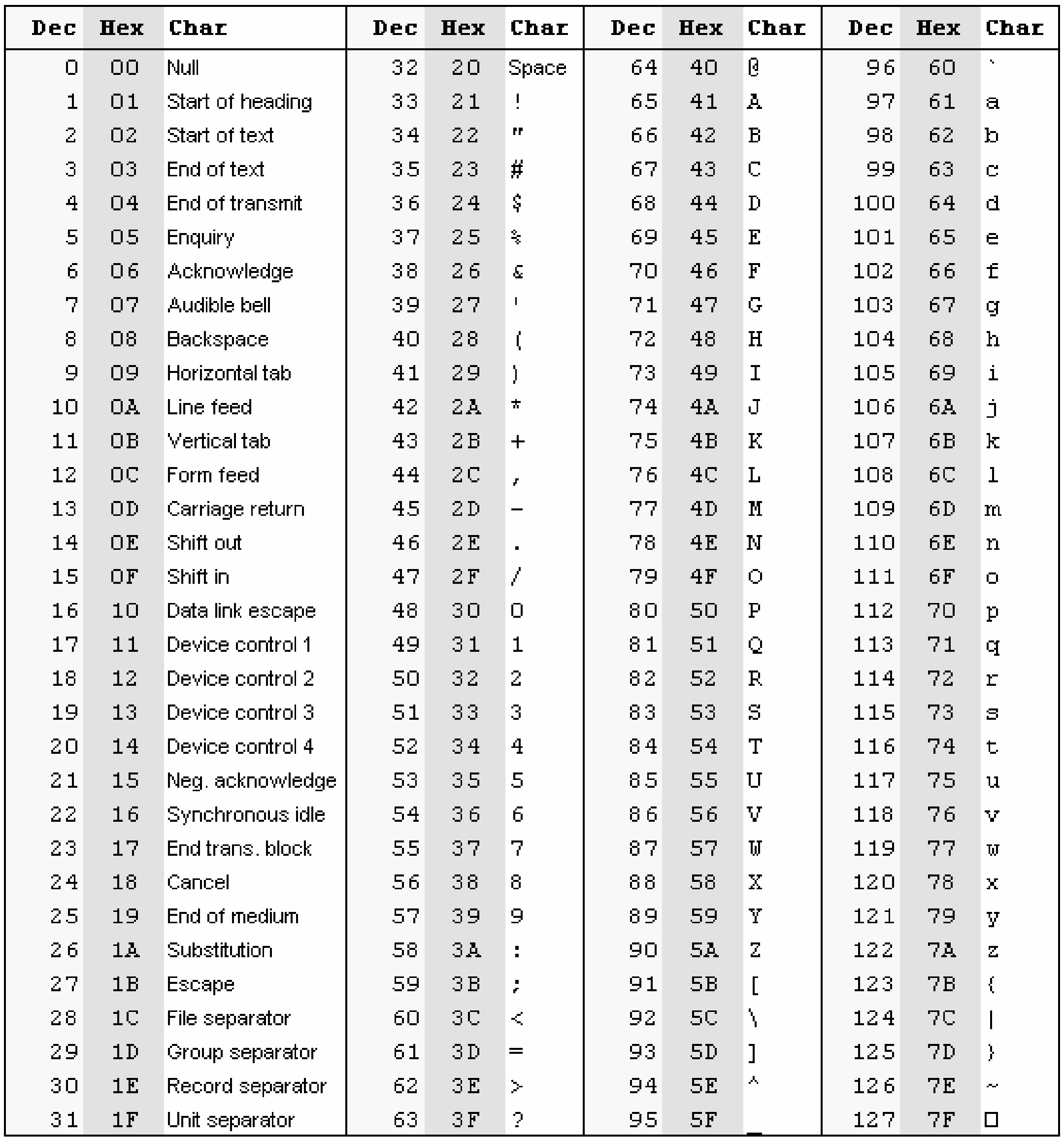
This example turns the SmartImage sensor into a terminal server. After you upload this script into the SmartImage sensor, open a telnet program (like Hyperterminal) and connect to the SmartImage sensor on port 5005. Only one command, `exit`, is currently implemented (more can be added by adding more if-then branches).

```
class myScript
{
  public static void main()
  {
    //declare socket objects to use and the necessary variables
    Socket sock;
    Socket newSock;

    int Result;
    int i;

    byte out[] = new byte[6];
    byte in[] = new byte[15];
    byte exit[] = new byte[4];

    boolean connection = true;

    //initialize the array to send out using ASCII codes
    out[0]=10;
    out[1]=13;
    out[2]=68;
    out[3]=86;
    out[4]=84;
    out[5]=62;

    //construct socket objects
    sock = new Socket();
    newSock = new Socket();

    //bind to a port and listen in that port. DebugPrint the
    //outputs
    Result = sock.Bind(5005);
    DebugPrint ("Bind Result:   " + Result);
    Result = sock.Listen();
    DebugPrint ("Listen Result: " + Result);

    //wait for the response and when succesful, assign it to
    //the other socket for communications
    do
    {
      Result=sock.Accept(newSock);
    } while(Result!=0);

    //enter a loop to handle the data exchange
    while(connection)
    {
      //send out the preformatted array with the DVT prompt
      Result=newSock.Send(out);
      DebugPrint ("Terminal Send Result:   " + Result);
      DebugPrint("Before Recv");

      //reset the array used for receiving data to 0
      i=0;
      while(i<15)
      {
        in[i]=0;
        i++;
      }

      //receive
      i=-1;
      do
      {
        //if the last read timed out then do not increment
        if (Result!=-14)
        {
          i++;
        }
        Result = newSock.Recv(in,i,1);
        //keep reading until a LF ASCII character is found.
      } while(in[i]!=10 || Result==-14);

      DebugPrint("AfterRecv");

      /*The String function converts an array of Bytes to a
        String.
        This should only be used if your array only contains
        printable ASCII characters*/
      DebugPrint("The recieved string is "+String(in));

      //output bytes to debug window
      i=0;
      while(i<15)
      {
        DebugPrint("Char at "+i+" is "+ in[i]);
        if(in[i]==13)
        {
          i = 15;
        }
        i++;
      }

      //setup array to exit
      i=0;
      while(i<4)
      {
        exit[i]=in[i];
        i++;
      }

      DebugPrint ("Terminal Receive result:   " + String(exit));

      //compare string to exiting command
      if(String(exit).compareTo("exit")==0)
      {
        sock.Close();
        newSock.Close();
        connection = false;
      }

      sleep (100);
    }
  }
}
```

This page intentionally left blank

Appendix A – ASCII Table of Characters

Basic table of characters (0 to 127)



Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	`
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	"	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmit	36	24	$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(	72	48	H	104	68	h
9	09	Horizontal tab	41	29	)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	Data link escape	48	30	0	80	50	P	112	70	p
17	11	Device control 1	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	50	32	2	82	52	R	114	72	r
19	13	Device control 3	51	33	3	83	53	S	115	73	s
20	14	Device control 4	52	34	4	84	54	T	116	74	t
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v
23	17	End trans. block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	63	3F	?	95	5F	_	127	7F	□

Extended table of characters (values 128 to 255)

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
128	80	Ç	160	A0	á	192	C0	└	224	E0	α
129	81	ü	161	A1	í	193	C1	┴	225	E1	ß
130	82	é	162	A2	ó	194	C2	┬	226	E2	Γ
131	83	â	163	A3	ú	195	C3	├	227	E3	π
132	84	ä	164	A4	ñ	196	C4	─	228	E4	Σ
133	85	à	165	A5	Ñ	197	C5	┼	229	E5	σ
134	86	å	166	A6	ª	198	C6	╞	230	E6	µ
135	87	ç	167	A7	º	199	C7	╟	231	E7	τ
136	88	ê	168	A8	¿	200	C8	╚	232	E8	Φ
137	89	ë	169	A9	⌐	201	C9	╔	233	E9	Θ
138	8A	è	170	AA	¬	202	CA	╩	234	EA	Ω
139	8B	ï	171	AB	½	203	CB	╦	235	EB	δ
140	8C	î	172	AC	¼	204	CC	╠	236	EC	∞
141	8D	ì	173	AD	¡	205	CD	═	237	ED	φ
142	8E	Ä	174	AE	«	206	CE	╬	238	EE	ε
143	8F	Å	175	AF	»	207	CF	╧	239	EF	∩
144	90	É	176	B0	░	208	D0	╨	240	F0	≡
145	91	æ	177	B1	▒	209	D1	╤	241	F1	±
146	92	Æ	178	B2	▓	210	D2	╥	242	F2	≥
147	93	ô	179	B3	│	211	D3	╙	243	F3	≤
148	94	ö	180	B4	┤	212	D4	╘	244	F4	⌠
149	95	ò	181	B5	╡	213	D5	╒	245	F5	⌡
150	96	û	182	B6	╢	214	D6	╓	246	F6	÷
151	97	ù	183	B7	╖	215	D7	╫	247	F7	≈
152	98	ÿ	184	B8	╕	216	D8	╪	248	F8	°
153	99	Ö	185	B9	╣	217	D9	┘	249	F9	∙
154	9A	Ü	186	BA	║	218	DA	┌	250	FA	·
155	9B	¢	187	BB	╗	219	DB	█	251	FB	√
156	9C	£	188	BC	╝	220	DC	▄	252	FC	ⁿ
157	9D	¥	189	BD	╜	221	DD	▌	253	FD	²
158	9E	₧	190	BE	╛	222	DE	▐	254	FE	■
159	9F	ƒ	191	BF	┐	223	DF	▀	255	FF	□

Index