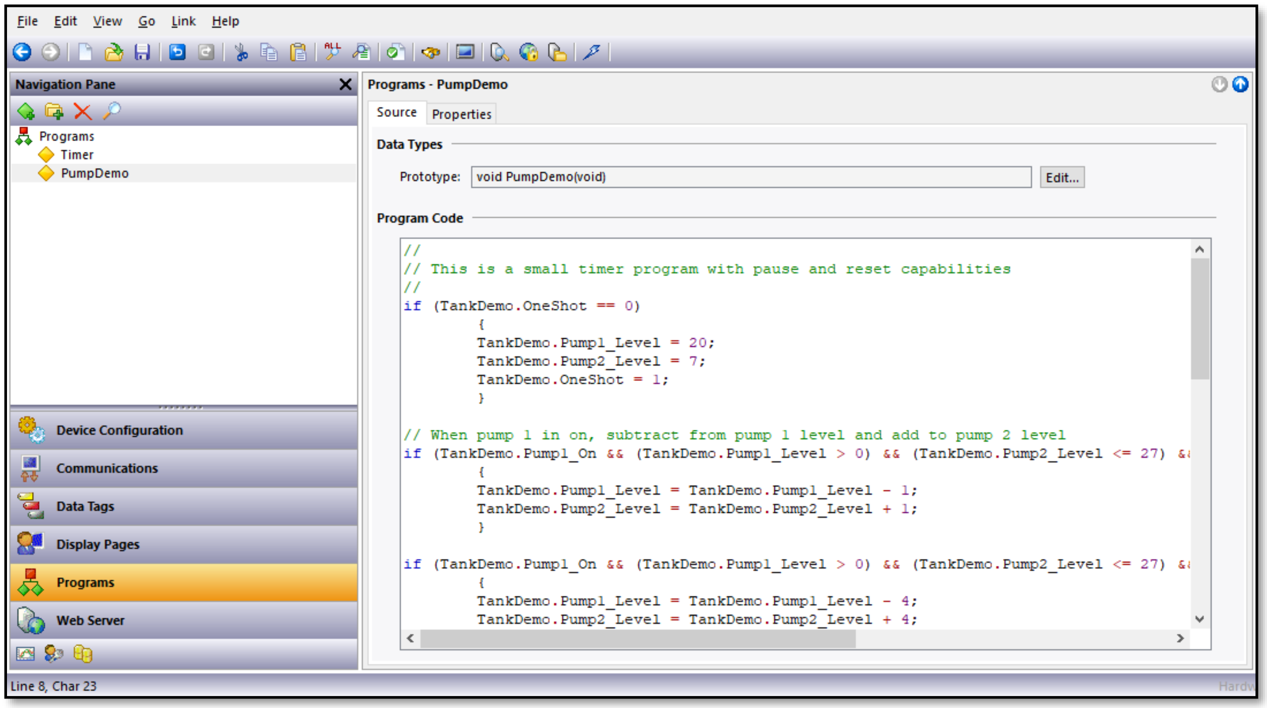
Programming in Crimson 3.X

Introduction & Syntax Primer

By Joe Wagner - Field Applications Engineer



Data Types

Prototype: void PumpDemo(void) Edit...

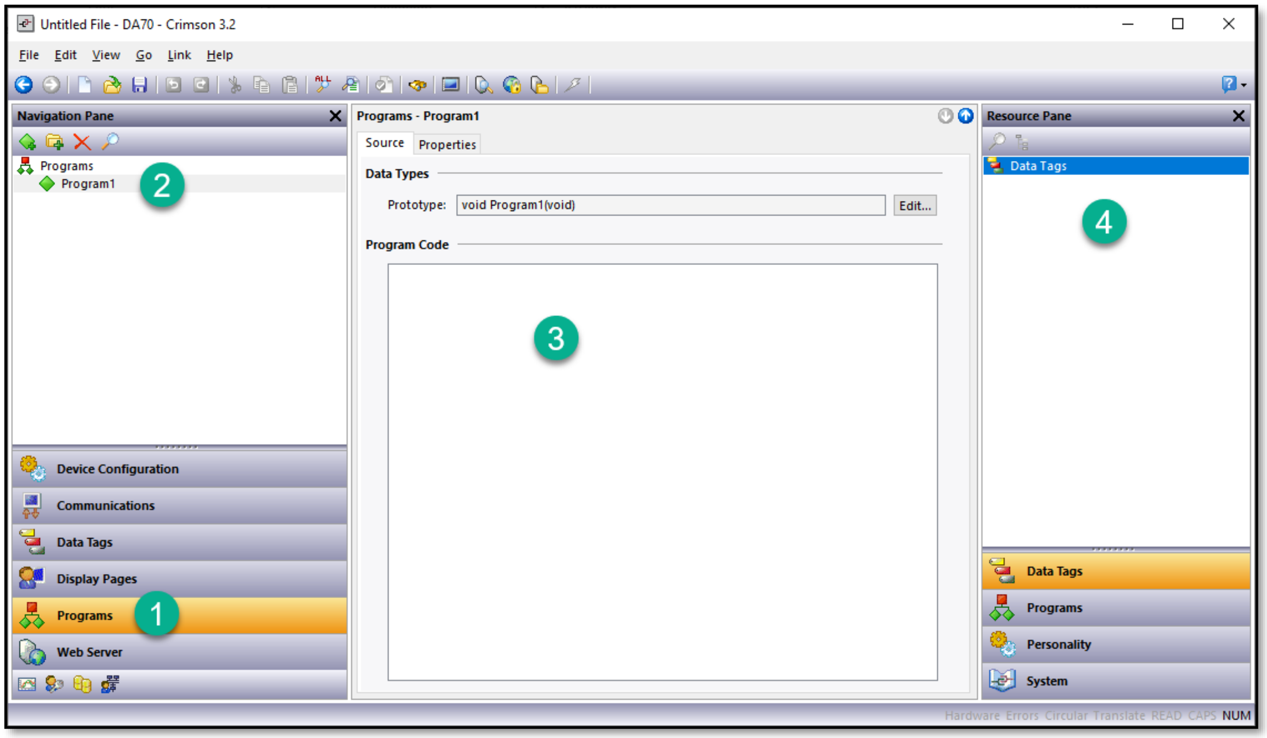
Program Code

```
//
// This is a small timer program with pause and reset capabilities
//
if (TankDemo.OneShot == 0)
        {
        TankDemo.Pump1_Level = 20;
        TankDemo.Pump2_Level = 7;
        TankDemo.OneShot = 1;
        }

// When pump 1 in on, subtract from pump 1 level and add to pump 2 level
if (TankDemo.Pump1_On && (TankDemo.Pump1_Level > 0) && (TankDemo.Pump2_Level <= 27) &
        {
        TankDemo.Pump1_Level = TankDemo.Pump1_Level - 1;
        TankDemo.Pump2_Level = TankDemo.Pump2_Level + 1;
        }

if (TankDemo.Pump1_On && (TankDemo.Pump1_Level > 0) && (TankDemo.Pump2_Level <= 27) &
        {
        TankDemo.Pump1_Level = TankDemo.Pump1_Level - 4;
        TankDemo.Pump2_Level = TankDemo.Pump2_Level + 4;
```

Contents

Section 1: Introduction and Basics

Programming Environment



1. Click once on the *Programs* section in the lower half of the *Navigation Pane* to get started.

2. In the upper half of the *Navigation Pane* you will see a *Programs* tree with an automatically generated program named *"Program1"* within it. Right clicking on *Program1* allows you to rename the program, create additional programs, create folders for organizing programs, and more.

3. In the *Editing Pane* (middle), you will see a *Source* tab and a *Properties* tab at the top, with *Source* selected by default.
   - Within the *Source* tab you will see sections for *Data Types* and *Program Code. Data Types* is only used when using the `return` function (covered later) and can otherwise be left as it is. *Program Code* contains the white text-editing area where we will be entering our program code and spending the most time.
   - Within the *Properties* tab you will see a section for *Environment* with options for execution of the program and reading external data, and a section for *Debugging* with options for trace output. See manual for more info on this tab if needed, otherwise it can be left alone.

4. The *Resource Pane* contains a variety of items that can be dragged into your code. The *Data Tags* and *Programs* categories are self-explanatory and provide quick access to those aspects of your database. The *System* category provides access to Crimson's extensive library of system variables and functions.

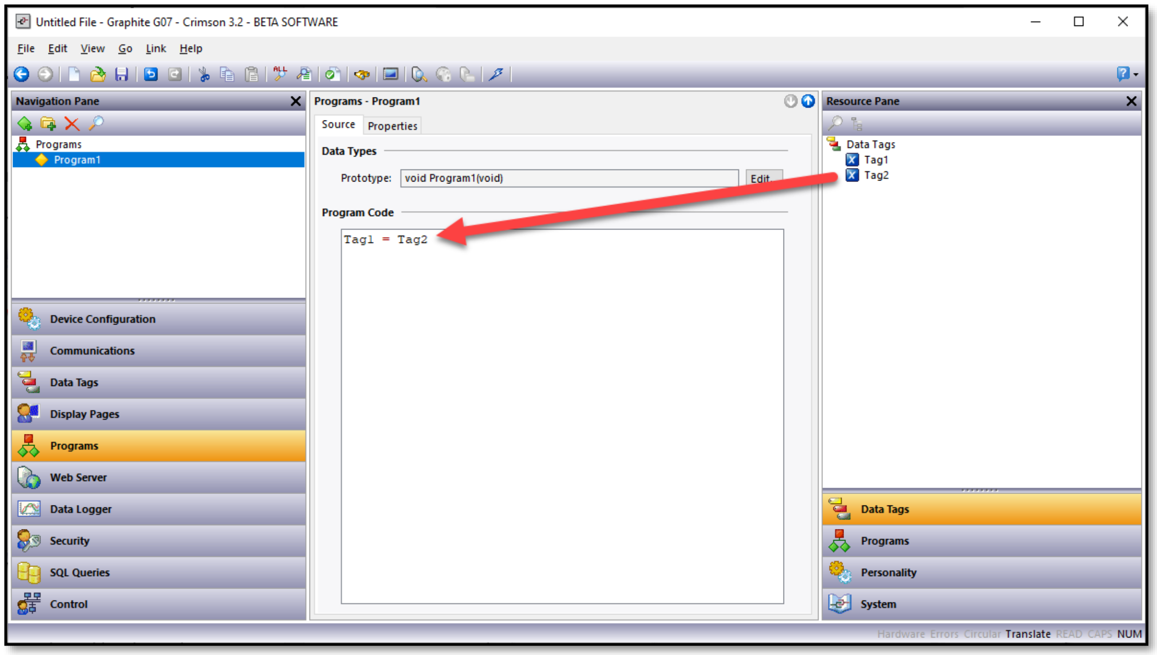
Adding Comments

You can add comments to your code by using `//` for single-line comments or `/* comment here */` for multi-line comments. See below for example:

```
// This is a single-line comment

/* This is line 1 of a multi-line comment
   This is line 2 of a multi-line comment
   This is line 3 of a multi-line comment
*/
```

Using Data Tags

You can use Data Tags in your programs either by typing out the name of the Data Tag, or by simply dragging the Data Tag from the *Resource Pane* and dropping it into the *Program Code* portion of the *Editing Pane*.



```
Tag1 = Tag2
```

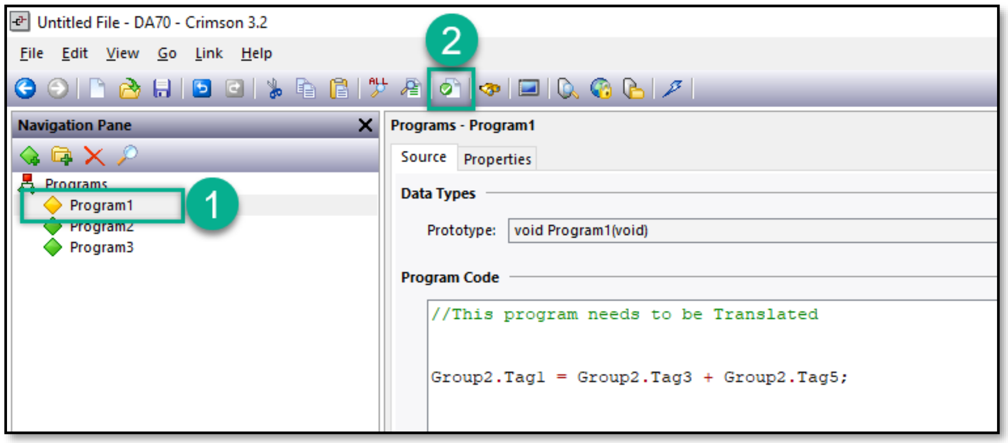
Using Local Variables

You also have the ability to declare local variables which will not appear in Data Tags and cannot be referenced anywhere other than the program which they are declared in. These variables can optionally be initialized by using the = operator followed by a value. See examples below:

```
//This is an example of declaring local variables.

int x,y,z;           // declare local integers x, y, and z
int q;               // declare local integer q
float b = 44.7;      // declare local float b initialized to 44.7
cstring c = "test"; // declare local string c initialized to "test"
```

Translating Programs

When you have finished writing the syntax of your program, you will need to *Translate* it. This will check the program for errors and allow the program to run when called. If the diamond icon next to the program name is green, this indicates that the program has been translated and validated. Yellow indicates that a program has been edited but not yet translated. Red indicates that a program contains one or more errors.



Data Types

Prototype: void Program1(void)

Program Code

```
//This program needs to be Translated

Group2.Tag1 = Group2.Tag3 + Group2.Tag5;
```

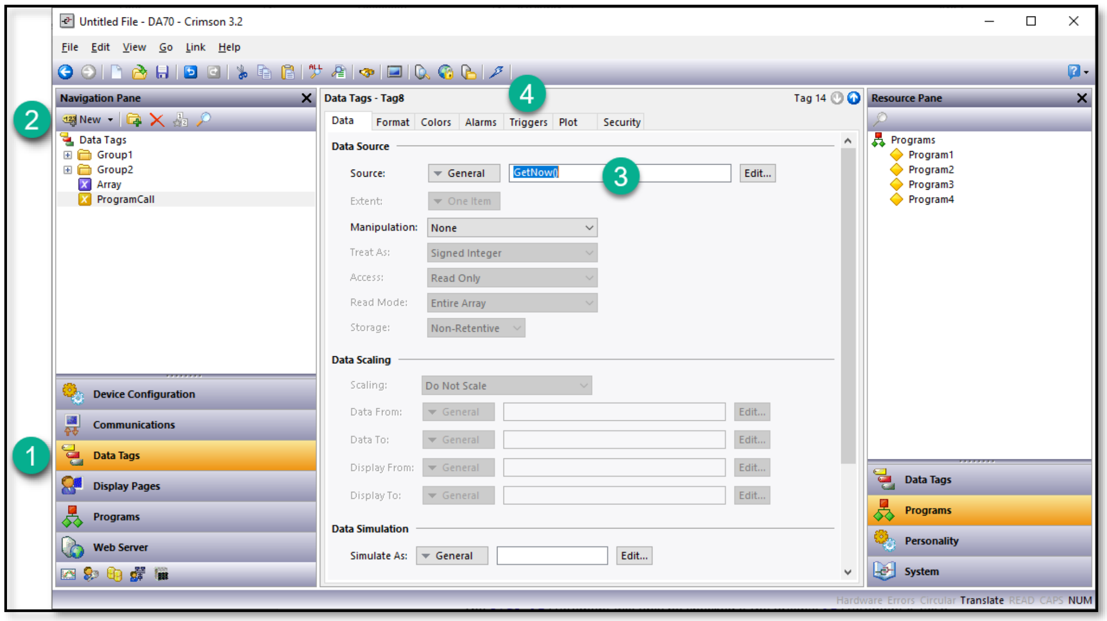
1. The diamond next to *Program1* is yellow, indicating that it has been edited but not yet translated.
2. Pressing this *Translate* button or pressing **CTRL+T** will check the program code for errors. If no errors are present, the diamond next to *Program1* will turn green and the program will be ready to be called.

Calling Programs

All programs will need to be invoked to run. This works slightly differently for programs that return values and those that do not. Below are some common methods for calling standard programs that do not use the **`return`** function:

Calling Programs with a GetNow() trigger

This is an example of creating a GetNow() trigger tag to call multiple programs every second.



Data Source

Source: General GetNow()

Extent: One Item

Manipulation: None

Treat As: Signed Integer

Access: Read Only

Read Mode: Entire Array

Storage: Non-Retentive

Data Scaling

Scaling: Do Not Scale

Data From: General

Data To: General

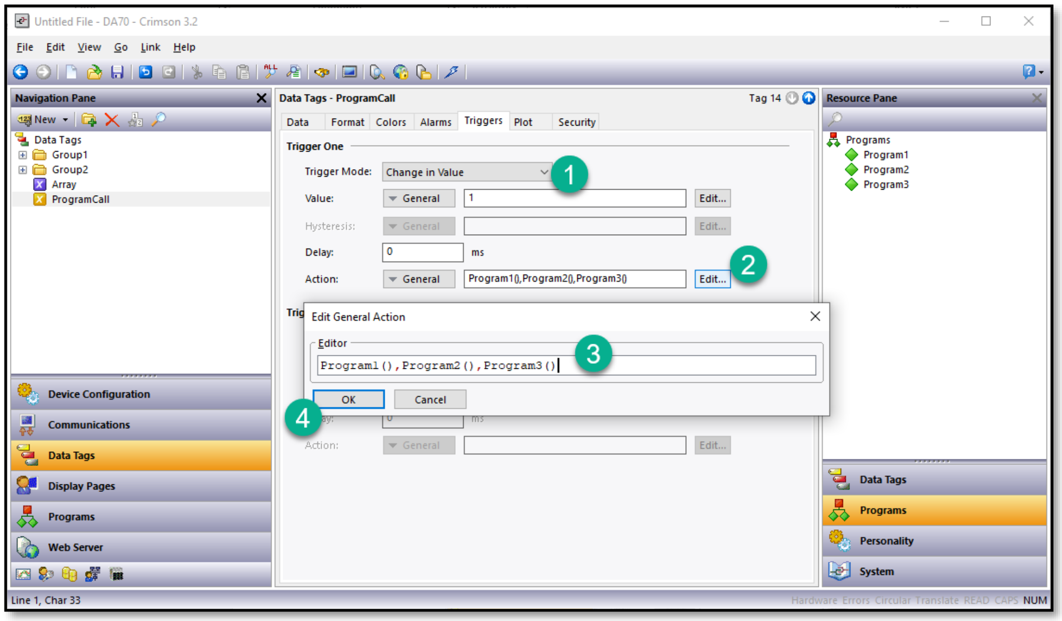
Display From: General

Display To: General

Data Simulation

Simulate As: General

1. Click on the *Data Tags* section
2. Create a new Data Tag (in this example, it is named '*ProgramCall*')
3. Set the *Source* of this tag to *General*, and enter `GetNow()` in the blank field as shown above
4. Click on the *Triggers* tab of this tag *(continued on next page)*



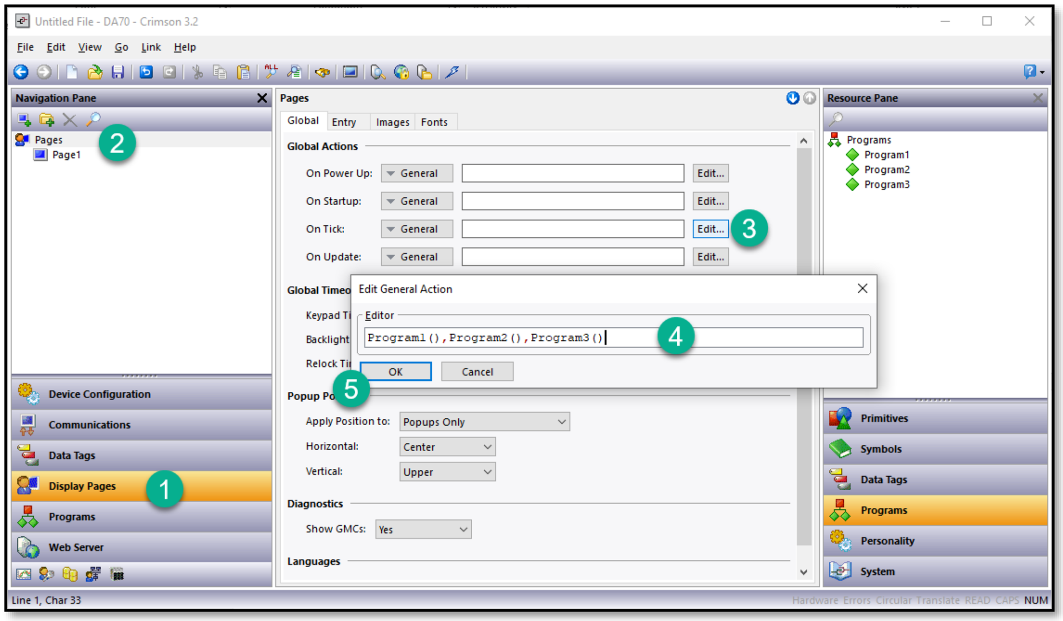
Trigger Mode: Change in Value
Value: General 1
Delay: 0 ms
Action: General Program1(),Program2(),Program3()

```
Program1(),Program2(),Program3()
```

1. From the *Triggers* tab, set the *Trigger Mode* to *Change in Value* and set the *Value* to 1.
2. With the *Action* set to *General*, click *Edit…*
3. Drag the programs you would like to call from the *Programs* section of the *Resource Pane* into the *Editor*, separated by commas.
4. Click *OK*

Calling Programs using Display Pages -> Global Actions

This is an example of using Display Pages settings to call multiple programs every 'tick', which is every second.



On Power Up: General
On Startup: General
On Tick: General
On Update: General

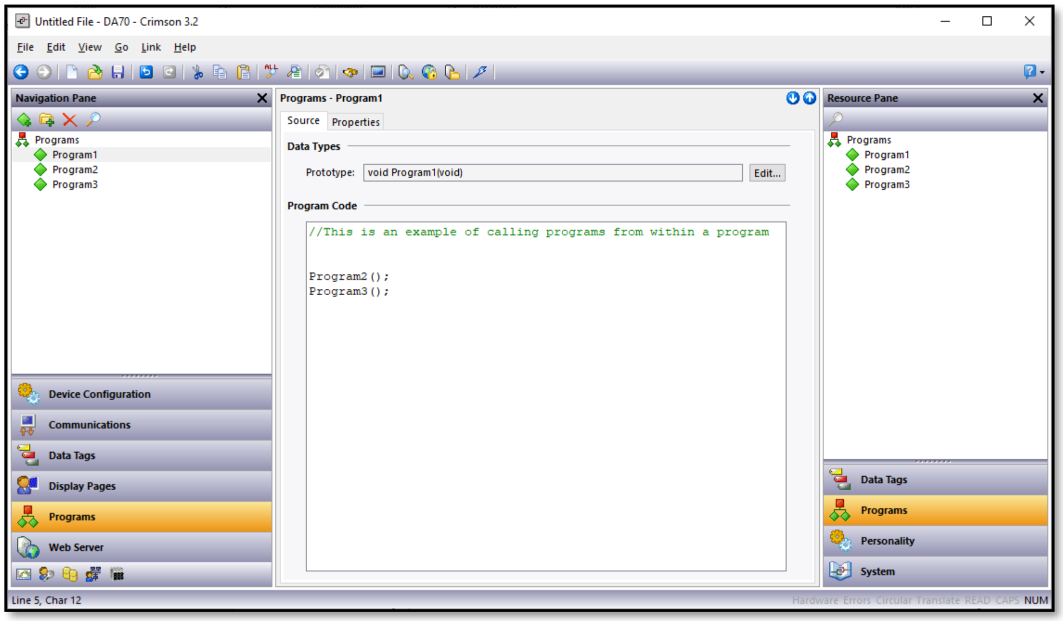
```
Program1(),Program2(),Program3()
```

Apply Position to: Popups Only
Horizontal: Center
Vertical: Upper
Show GMCs: Yes

1. Click on the *Display Pages* section
2. Click on *Pages* (top level of Navigation Pane)
3. Within the *Global* tab, find *On Tick* and click *Edit…*
4. Drag the programs you would like to call from the *Programs* section of the *Resource Pane* into the *Editor*, separated by commas.
5. Click *OK*

Calling Programs from other Programs

It is also possible to call programs within the syntax of other programs, as shown below. In this case, *Program1* is calling *Program2* and *Program3*. Note that *Program1* would still need to be called somewhere else in order for this to function.



Prototype: void Program1(void)

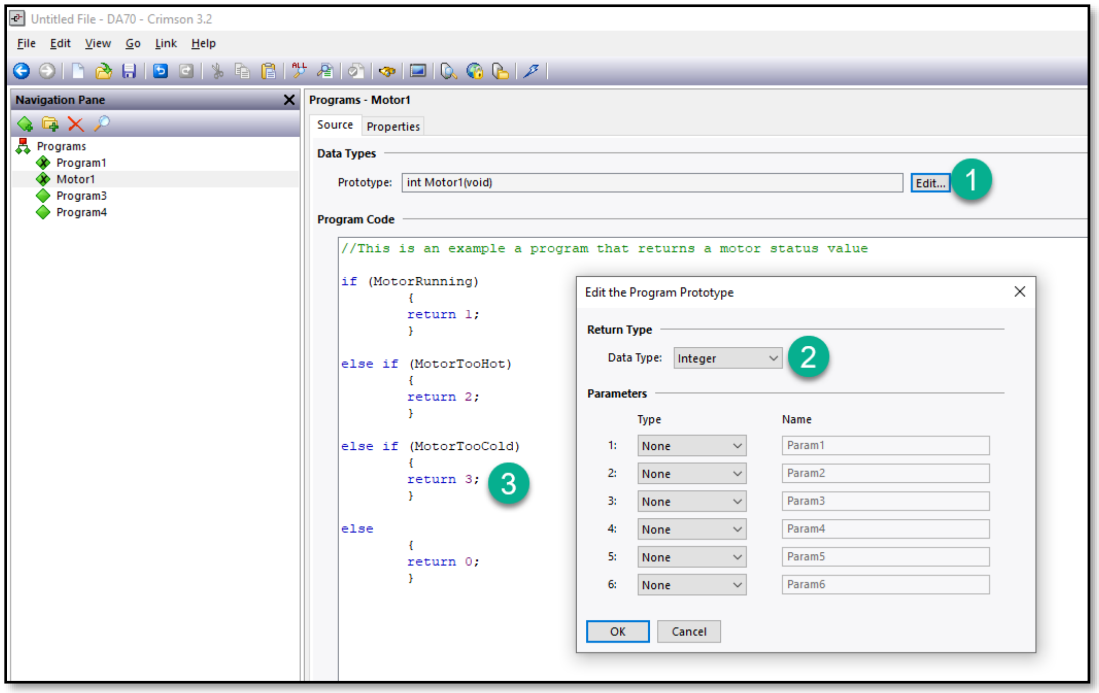
```
//This is an example of calling programs from within a program


Program2();
Program3();
```

Returning Values

As mentioned above, programs can alternatively be configured to return a single value. These programs are typically invoked via tag configuration by setting the tag's *Value* property to *Program(),* where *Program* is the name of the program in question. For example, if you want to analyze a number of conditions relating to a motor and return a value to indicate the current state, you could create a program that returns an integer like this:

```
//This is an example a program that returns a motor status value
//The data type for this program must be set to INT

if(MotorRunning)
       {
       return 1;
       }

else if (MotorTooHot)
       {
       return 2;
       }

else if (MotorTooCold)
       {
       return 3;
       }

else
       {
       return 0;
       }
```

Keep in mind that in order to use the `return` function in a program, the program must have a *Data Type* that is compatible with the value being returned. See below for instructions:



Data Types

Prototype: int Motor1(void)

Program Code

```
//This is an example a program that returns a motor status value

if (MotorRunning)
        {
        return 1;
        }

else if (MotorTooHot)
        {
        return 2;
        }

else if (MotorTooCold)
        {
        return 3;
        }

else
        {
        return 0;
        }
```

Edit the Program Prototype

Return Type

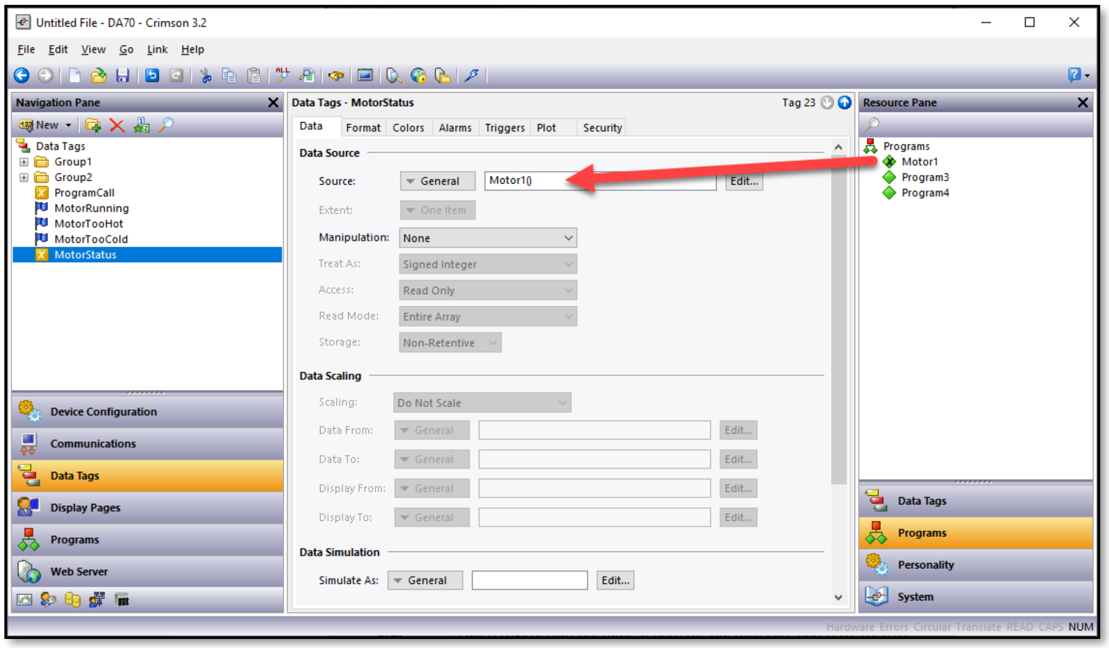
Data Type: Integer

Parameters

	Type	Name
1:	None	Param1
2:	None	Param2
3:	None	Param3
4:	None	Param4
5:	None	Param5
6:	None	Param6

OK Cancel

1. With your program selected, in the *Data Types* section of the *Editing Pane*, click *Edit...*
2. In the *Return Type* section, use the drop-down menu to select a compatible *Data Type*
3. Use the `return` function to return values
4. See below where a motor status tag has been configured to be equal to the return value of this *Motor1* program



Data Source

Source: General Motor1() Edit...

Extent: One Item

Manipulation: None

Treat As: Signed Integer

Access: Read Only

Read Mode: Entire Array

Storage: Non-Retentive

Data Scaling

Scaling: Do Not Scale

Data From: General Edit...

Data To: General Edit...

Display From: General Edit...

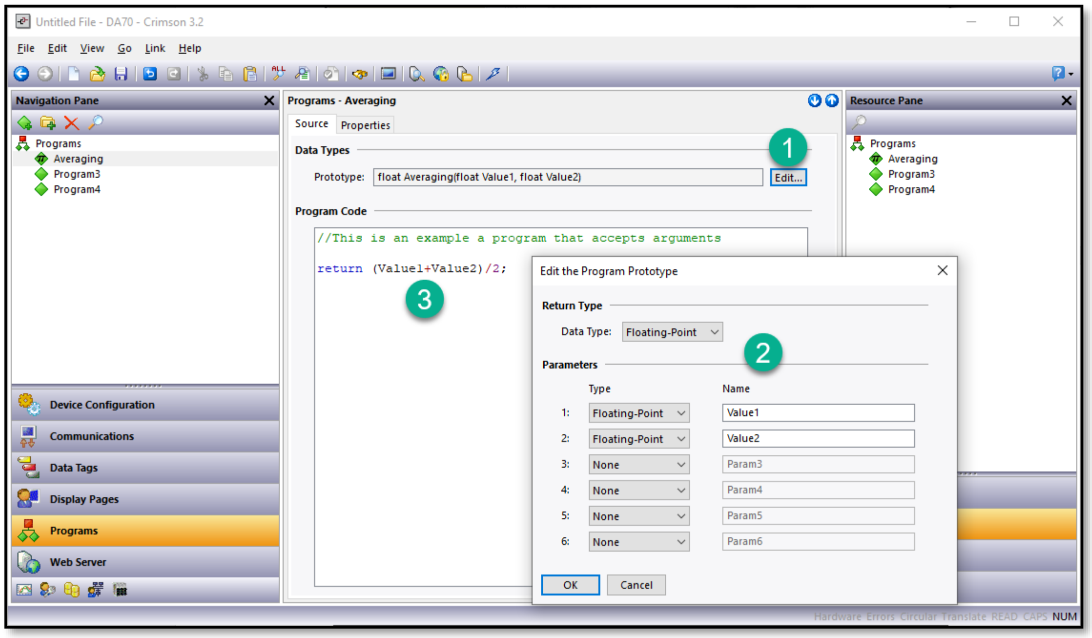
Display To: General Edit...

Data Simulation

Simulate As: General Edit...

Passing Arguments

Programs are also capable of accepting arguments. Suppose you want to write a program called *Averaging* to return the average of two floating-point values. The program could be configured to accept two floating-point arguments, *Value1* and *Value2*. See below for instructions:



Data Types

Prototype: float Averaging(float Value1, float Value2) Edit...

Program Code

```
//This is an example a program that accepts arguments

return (Value1+Value2)/2;
```

Edit the Program Prototype

Return Type

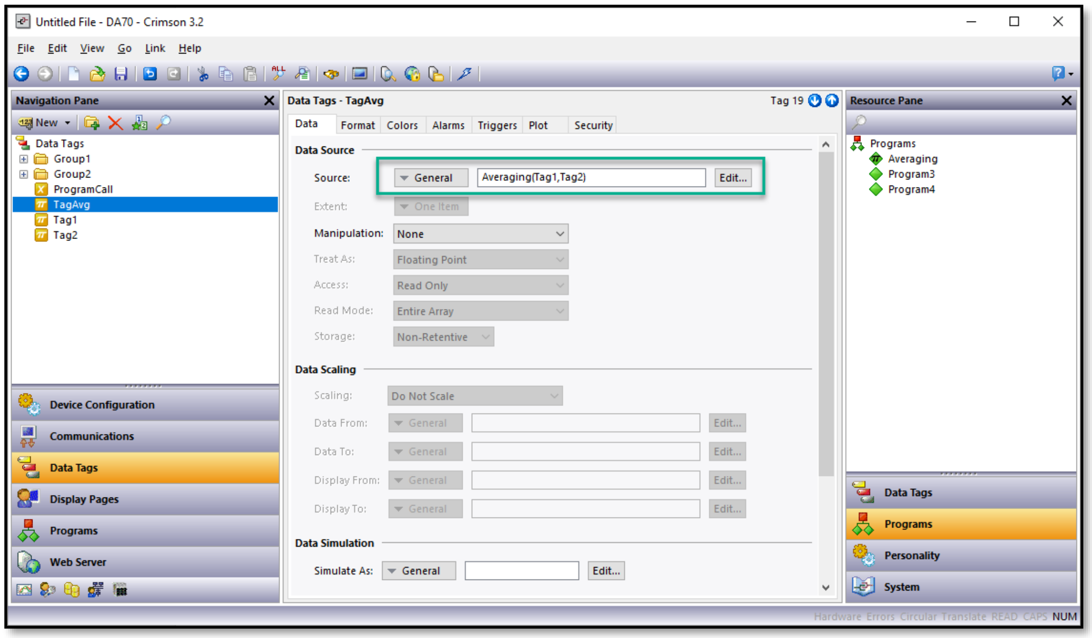
Data Type: Floating-Point

Parameters

	Type	Name
1:	Floating-Point	Value1
2:	Floating-Point	Value2
3:	None	Param3
4:	None	Param4
5:	None	Param5
6:	None	Param6

OK Cancel

1. With your program selected, in the *Data Types* section of the *Editing Pane*, click *Edit…*
2. In the *Return Type* section, use the drop-down menu to select the *Floating-Point* Data Type
   In the *Parameters* section, set the *Type* of parameters 1 and 2 to *Floating-Point*. Name them *Value1* and *Value2*
3. Now we are able to reference the *Value1* and *Value2* arguments in the program
4. See below where we invoke our *Averaging* program with *Tag1* and *Tag2* as dynamic arguments. *Value1* will be equal to *Tag1*, and *Value2* will be equal to *Tag2*.



Data Source

Source: General Averaging(Tag1,Tag2) Edit...

Extent: One Item

Manipulation: None

Treat As: Floating Point

Access: Read Only

Read Mode: Entire Array

Storage: Non-Retentive

Data Scaling

Scaling: Do Not Scale

Data From: General Edit...

Data To: General Edit...

Display From: General Edit...

Display To: General Edit...

Data Simulation

Simulate As: General Edit...

Section 2: Syntax Cheat-Sheet

Operators

Logical Operators

Operator	Description
&&	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.
||	Called Logical OR Operator. If any of the two operands is non-zero, then the condition becomes true
!	Called Logical NOT Operator. It is used to reverse the logical state of its operand. If a condition is true, then Logical NOT operator will make it false.

Relational Operators

Operator	Description
==	Checks if the values of two operands are equal or not. If yes, then the condition becomes true.
!=	Checks if the values of two operands are equal or not. If the values are not equal, then the condition becomes true.
>	Checks if the value of left operand is greater than the value of right operand. If yes, then the condition becomes true.
<	Checks if the value of left operand is less than the value of right operand. If yes, then the condition becomes true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand. If yes, then the condition becomes true.
<=	Checks if the value of left operand is less than or equal to the value of right operand. If yes, then the condition becomes true.

Arithmetic Operators

Operator	Description
+	Adds two operands.
-	Subtracts second operand from the first.
*	Multiplies both operands.
/	Divides numerator by de-numerator.
%	Modulus Operator and remainder of after an integer division.
++	Increment operator increases the integer value by one.
--	Decrement operator decreases the integer value by one.

Assignment Operators

Operator	Description
=	Simple assignment operator. Assigns values from right side operands to left side operand.
+=	Add AND assignment operator. It adds the right operand to the left operand and assign the result to the left operand.
-=	Subtract AND assignment operator. It subtracts the right operand from the left operand and assigns the result to the left operand.
*=	Multiply AND assignment operator. It multiplies the right operand with the left operand and assigns the result to the left operand.
/=	Divide AND assignment operator. It divides the left operand with the right operand and assigns the result to the left operand.
%=	Modulus AND assignment operator. It takes modulus using two operands and assigns the result to the left operand.

If/Else Statements

- `if` statements are used to check for specific true/false conditions and execute code if true.
- `else if` can be optionally added beneath an `if` statement to check for another specific condition. The `else if` statement will only be checked if the original `if` statement is false.
- `else` can also be optionally added beneath an `if` statement, or beneath an `else if` statement if one exists. The `else` statement will only be checked if the original `if` and `else if` statements are false.
- See **page 9** for descriptions of the operators used in the below examples.

If/Else Syntax Example 1:

```
//This is a simple If Else statement
//The condition (true/false) lives in the parentheses
//Inside the brackets {} is the action that happens on a true
condition

if(InputTag1 > InputTag2)
{
        OutputTag1 = InputTag2 + 10;
}

else
{
        OutputTag1 = 0;
}
```

If/Else Syntax Example 2:

```
//This is a complex If Else statement
//The conditions (true/false) live in the parentheses
//Inside the brackets {} is the action that happens on a true
condition

if((InputTag1 <= 100) && (InputTag2 != 100))
{
        OutputTag1 = 0;
        OutputTag2 = 100;
}

else if((InputTag1 > 100) && (InputTag2 == 100))
{
        OutputTag1 = 50;
        OutputTag2 = 200;
}

else if((InputTag1 >= 300) || (InputTag2 >= 300))
{
        OutputTag1 = 75;
        OutputTag2 = 300;
}

else
{
        OutputTag1 = 500;
        OutputTag2 = 500;
}
```

If/Else Syntax Example 3:

```
//This is an If Else statement utilizing flag tags
//The conditions (true/false) live in the parentheses
//Inside the brackets {} is the action that happens on a true
condition
//Note that “Flag1 == false and “!Flag1” would have the same effect
//Note that “Flag2 == true” and “Flag2” would have the same effect

if (Flag1 && !Flag2)
       {
       OutputTag1 = 1000;
       }

else if ((Flag1 == false) && (Flag2 == true))
       {
       OutputTag1 = 5000;
       }

else
       {
       OutputTag1 = 99;
       Flag3 = true;
       }
```

Switch Statements

A `switch` statement is used to compare an integer value against a number of possible constants, and to perform an action based upon which value is matched. Looking at the example below, if `State1` value is equal to `3`, `TagA` will be `1` and `TagB` will be `1`. The default state will be active if `State1` is not equal to `1`, `2`, or `3`. Note that the `break` statement is used at the end of each case statement. Also note that the argument (`State1` in this case) must be an integer data type in order for the `switch` statement to work.

```
//This is a simple Switch statement
//The state in () is checked and compared to the cases
//When a match is found the code in the appropriate case is executed

switch(State1)
{
       case 1:
              TagA = 1;
              TagB = 0;
              break;

       case 2:
              TagA = 0;
              TagB = 1;
              break;

       case 3:
              TagA = 1;
              TagB = 1;
              break;

       default:
              TagA = 0;
              TagB = 0;
              break;

}
```

Loops

There are 3 different types of loops which can execute a section of code while a certain condition is true. The `while` loop tests the condition before the code is executed. The `do` loop tests the condition afterwards. The `for` loop is essentially a compact way of defining a while loop. The `break` statement can be used to terminate the loop early, while the `continue` statement can be used to skip the balance of the loop body. Note that all 3 of the code examples that follow will have the exact same output.

While Loops

This type of loop repeats the action that follows it while the condition in the `while` statement remains true.

```
//This is a while loop which writes to 100 array elements
//Tag1 will increment to 100 and the Array elements will populate with
the Tag1 value. For example, Array[42] will have a value of 42.

while(Tag1 < 100)
{
Array[Tag1] = Tag1;
Tag1++;
}
```

For Loops

This type of loop contains three elements separated by semicolons and an action:

- *Initialization*- This step allows you to declare and initialize any loop control variables.
- *Condition*- This is the statement is evaluated at the start of each loop iteration. If it is true, the body of the loop is executed.
- *Induction*- This step is used to make a change to the control variable to move the loop on to its next iteration.
- Between the curly brackets, there is an action which the loop will execute for each iteration.

```
//This is a for loop which writes to 100 array elements
//Tag1 will increment to 100 and the Array elements will populate with
the Tag1 value. For example, Array[42] will have a value of 42.

for( Tag1=0; Tag1<100; Tag1++ )
{
Array[Tag1] = Tag1;
}
```

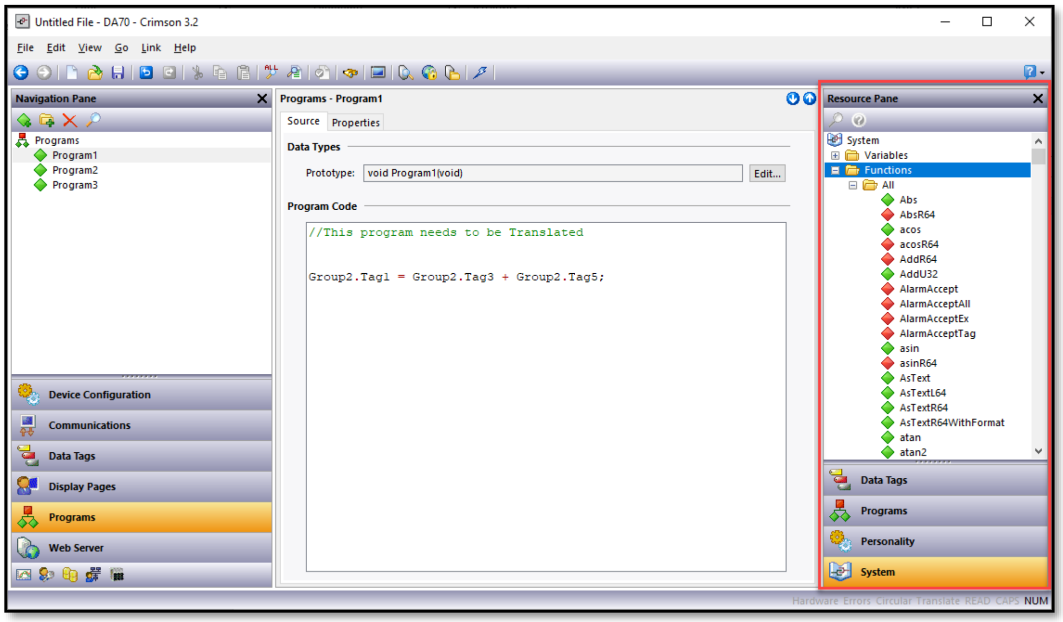
Do Loops

This type of loop is very similar to the while loop, except that the condition is not tested until the end of the loop. Because of this, the loop will always execute at least once.

```
//This is a do loop which writes to 100 array elements
//Tag1 will increment to 100 and the Array elements will populate with
the Tag1 value. For example, Array[42] will have a value of 42.

do
{
Array[Tag1] = Tag1;
Tag1++;
}

while(Tag1 < 100);
```

Using System Functions

There are over 250 System Functions built into Crimson which can be used within your program code. To use these functions, go to the *System* section of the *Resource Pane*, and it expand the *Functions* folder. From here, you can expand any of the sub-folders and drag functions directly into your program code. Functions generally require some configuration in the form of arguments. If you're not sure how to use a function or how to structure the arguments, right click on the function and click once on '*Get Help*' which will open up a manual explaining the function along with an example.



Prototype: void Program1(void)

```
//This program needs to be Translated

Group2.Tag1 = Group2.Tag3 + Group2.Tag5;
```

System Function Examples

- **GotoPage(***name***)** – Selects page *name* to be shown on the Crimson device's display. Used for navigation.

Argument	Type	Description
`name`	Display Page	The page to be displayed.

Example:
```
GotoPage(Page1);
```

- **HasAccess(***rights***)** – Returns a value of true or false depending on whether the current user has access rights defined by the *rights* parameter.

Argument	Type	Description
`rights`	`int`	The required access rights.

Example:
```
If (HasAccess(1))
        {
        Data1 = 0;
        }
```

- **IsDeviceOnline(*device*)** – Reports if device is online or not.

Argument	Type	Description
`device`	`int`	The index of the device to be checked

Example:
```
Device1_Comms = IsDeviceOnline(1);
```

- **IsLoggingActive()** – Returns true or false, indicating whether data logging is active in the current database.

Argument	Type	Description
`none`		

Example:
```
Device1_Log = IsLoggingActive();
```

- **Scale(*data, r1, r2, e1, e2*)** – This function linearly scales the `data` argument, assuming it to contain values between `r1` and `r2`, and producing a return value between `e1` and `e2`.

Argument	Type	Description
`data`	`int`	The value to be scaled
`r1`	`int`	The min raw value stored in `data`
`r2`	`int`	The max raw value stored in `data`
`e1`	`int`	The engineering value corresponding to `r1`
`e2`	`int`	The engineering value corresponding to `r2`

Example:
```
ScaledData = Scale(MyValue, 0, 4095, 0, 100);
```

- **SendFile(*rcpt, file*)** – Sends an email from the Crimson device with the file specified attached. The message will be sent using the appropriate mail transport as configured in the database.

Argument	Type	Description
`rcpt`	`int`	The recipient's index in the database's address book
`file`	`cstring`	The path and file name to be sent

Example:
```
SendFile(0, "/LOGS/LOG1/260706.csv");
```

- **Flash(*freq*)** – Returns an alternating true or false value that completes a cycle `freq` times per second. This function is useful when animating display primitives or changing their colors.

Argument	Type	Description
`freq`	`int`	The number of times per second to flash

Example:
```
BlinkingBit = Flash(2);
```

- **GetUpDownData(***data, limit***)** – This function takes a steadily increasing value and converts it to a value that oscillates between 0 and `limit`–1. It is typically used within a demonstration database to generate realistic looking animation, often by passing `DispCount` as the data parameter so that the resulting value changes on each display update.

Argument	Type	Description
`data`	`int`	A steadily increasing value source
`limit`	`int`	The number of values to generate

Example:
```
Data = GetUpDownData(DispCount, 100);
```

- **PrintScreenToFile(***path, name, res***)** – Saves a bitmap copy of the current display to the indicated file. Passing an empty string for name will allow Crimson to select a unique filename for the new image. The res argument can be set to one to create an 8 bits-per-pixel bitmap, while a value of zero will create a 16 bits-per-pixel bitmap.

Argument	Type	Description
`path`	`cstring`	The directory in which the file should be created
`name`	`cstring`	The filename to be used
`res`	`int`	The required color resolution of the image

Example:
```
PrintScreenToFile("/", "Screen_Capture", 0);
```