CRIMSON 3 REFERENCE MANUAL

Written by Mike Granby and Jesse Benefiel.

TABLE OF CONTENTS

CRIMSON 3 REFERENCE MANUAL

FUNCTION REFERENCE

The following pages describe the various standard functions that provided by Crimson. These functions can be invoked within programs, actions or expressions as described in the previous chapters. Functions that are marked as *active* may not be used in expressions that are not allowed to change values, such as in the controlling expression of a display primitive. Functions that are marked as *passive* may be used in any context.

ABS(*VALUE*)

ARGUMENT	TYPE	DESCRIPTION
value	**int** / **float**	The value to be processed.

DESCRIPTION

Returns the absolute value of the argument. In other words, if *value* is a positive value, that value will be returned; if *value* is a negative value, a value of the same magnitude but with the opposite sign will be returned.

FUNCTION TYPE

This function is passive.

RETURN TYPE

`int` or `float`, depending on the type of the *value* argument.

EXAMPLE

```
Error := abs(PV - SP)
```

ABSR64(RESULT, TAG)

ARGUMENT	TYPE	DESCRIPTION
result	**int**	The result.
tag	**int**	The tag for which to compute the absolute value.

DESCRIPTION

Calculates the absolute value of *tag* using 64-bit (double precision) floating point math and stores the result in *result*.

The input operand *tag* should be obtained from one of the 64-bit conversion functions provided or from a driver that can read double precision values. All arguments to this function must be integer arrays with lengths of 2. An in depth example is provided in the entry for AddR64.

FUNCTION TYPE

This function is active.

RETURN TYPE

```
void
```

EXAMPLE

```
AbsR64(result[0], tag[0])
```

ACOS(*VALUE*)

ARGUMENT	TYPE	DESCRIPTION
value	**float**	The value to be processed.

DESCRIPTION

Returns the angle *theta* in radians such that `cos(`*theta*`)` is equal to *value*.

FUNCTION TYPE

This function is passive.

RETURN TYPE

`float`

EXAMPLE

```
theta := acos(1.0)
```

ACOSR64(RESULT, TAG)

ARGUMENT	TYPE	DESCRIPTION
result	int	The result.
tag	int	The value to be processed.

DESCRIPTION

Calculates the arccosine of *tag* using 64-bit (double precision) floating point math and stores the result in *result*.

The input operand *tag* should be obtained from one of the 64-bit conversion functions provided or from a driver that can read double precision values. All arguments to this function must be integer arrays with lengths of 2. An in depth example is provided in the entry for AddR64.

FUNCTION TYPE

This function is active.

RETURN TYPE

```
void
```

EXAMPLE

```
acosR64(result[0], tag[0])
```

ADDR64(RESULT, TAG1, TAG2)

ARGUMENT	TYPE	DESCRIPTION
result	**int**	The result.
tag1	**int**	The first addend tag.
tag2	**int**	The second addend tag.

DESCRIPTION

Calculates the value of *tag1* plus *tag2* using 64-bit (double precision) floating point math and stores the result in *result.*

The input operands *tag1* and *tag2* should be obtained from one of the 64-bit conversion functions provided or from a driver that can read double precision values. The value of *result* can be used for further 64-bit calculations or formatted for display as a string using the AsTextR64 function.

FUNCTION TYPE

This function is active.

RETURN TYPE

```
void
```

EXAMPLE

This example shows how to calculate π + 2 using 64-bit math. In this example Operand1, Operand2 and Result are integer array tags, each with an extent of 2.

```
int NumberTwo := 2;

cstring PiString := "3.14159265358979";

IntToR64(Operand1[0], NumberTwo);

TextToR64(PiString, Operand2[0]);

AddR64(Result[0], Operand1[0], Operand2[0]);

cstring PiPlusTwo := AsTextR64(Result[0]);
```

PiPlusTwo now contains " 5.141592654", the approximate value of π + 2 represented as a string.

ADDU32(TAG1, TAG2)

ARGUMENT	TYPE	DESCRIPTION
tag1	**int**	The first addend tag.
tag2	**int**	The second addend tag.

DESCRIPTION

Returns the value of *tag1* plus *tag2* in an unsigned context.

FUNCTION TYPE

This function is passive.

RETURN TYPE

```
int
```

EXAMPLE

```
Result := AddU32(tag1, tag2)
```

ALARMACCEPT(*ALARM*)

ARGUMENT	TYPE	DESCRIPTION
`alarm`	`int`	A value encoding the alarm to be accepted.

DESCRIPTION

This function is not implemented in the current build.

FUNCTION TYPE

This function is active.

RETURN TYPE

This function does not return a value.

ALARMACCEPTALL()

ARGUMENT	TYPE	DESCRIPTION
none		

DESCRIPTION

Accepts all active alarms.

FUNCTION TYPE

This function is passive.

RETURN TYPE

This function does not return a value.

EXAMPLE

```
AlarmAcceptAll()
```

ALARMACCEPTEX(*SOURCE, METHOD, CODE*)

ARGUMENT	TYPE	DESCRIPTION
`source`	`int`	The source of the alarm.
`method`	`int`	The acceptance method.
`code`	`int`	The acceptance code.

DESCRIPTION

Accepts an alarm that has been signaled by a rich communications driver that is itself capable of generating alarms and events. This functionality is not used by any drivers that are currently included with Crimson.

FUNCTION TYPE

This function is active.

RETURN TYPE

This function does not return a value.

ALARMACCEPTTAG(*TAG, INDEX, EVENT*)

ARGUMENT	TYPE	DESCRIPTION
tag	**int**	The index of the tag for which the alarm is defined.
index	**int**	The relevant element of an array tag, or zero otherwise.
event	**int**	Either 1 or 2, depending on the alarm to be accepted.

DESCRIPTION

Accepts a alarm generated by a tag. The arguments indicate the tag number and the alarm number, and may optionally indicate an array element. When accepting alarm on tags that are not arrays, set the element number to zero.

FUNCTION TYPE

This function is active.

RETURN TYPE

This function does not return a value.

EXAMPLE

```
AcceptAlarmTag(10, 0, 1)
```

ASIN(*VALUE*)

ARGUMENT	TYPE	DESCRIPTION
value	**float**	The value to be processed.

DESCRIPTION

Returns the angle *theta* in radians such that `sin(`*theta*`)` is equal to *value*.

FUNCTION TYPE

This function is passive.

RETURN TYPE

`float`

EXAMPLE

```
theta := asin(1.0)
```

ASINR64(RESULT, TAG)

ARGUMENT	TYPE	DESCRIPTION
result	**int**	The result.
tag	**int**	The value to be processed.

DESCRIPTION

Calculates the arcsine of *tag* using 64-bit (double precision) floating point math and stores the result in *result*.

The input operand *tag* should be obtained from one of the 64-bit conversion functions provided or from a driver that can read double precision values. All arguments to this function must be integer arrays with lengths of 2. An in depth example is provided in the entry for AddR64.

FUNCTION TYPE

This function is active.

RETURN TYPE

```
void
```

EXAMPLE

```
asinR64(result[0], tag[0])
```

ASTEXT(*N*)

ARGUMENT	TYPE	DESCRIPTION
n	int / float	The value to be converted to text.

DESCRIPTION

Returns the numeric value, formatted as a string. The formatting performed is equivalent to that perform by the General numeric format. Note that numeric tags can be converted to strings by using their `AsText` property, by referring, for example, to `Tag1.AsText`.

FUNCTION TYPE

This function is passive.

RETURN TYPE

`cstring`

EXAMPLE

```
Text = AsText(Tag1 / Tag2)
```

ASTEXTR64(DATA)

ARGUMENT	TYPE	DESCRIPTION
data	**int**	The 64-bit floating point value to convert.

DESCRIPTION

Converts the value stored in *data* from a 64-bit floating point value into a string that is suitable for display. The tag *data* must be an integer array with an extent of at least 2. The value of *data* is typically obtained from one of the 64-bit floating point math functions provided. See the entry for AddR64 for an example of the intended use of this function.

FUNCTION TYPE

This function is passive.

RETURN TYPE

```
cstring
```

EXAMPLE

```
Result := AsTextR64(data[0])
```

ASTEXTR64WITHFORMAT(FORMAT, DATA)

ARGUMENT	TYPE	DESCRIPTION
format	**cstring**	A string containing the desired width.
data	**int**	The 64-bit floating point value to convert.

DESCRIPTION

Converts the value stored in *data* from a 64-bit floating point value into a string that is suitable for display. The tag *data* must be an integer array with an extent of at least 2. The value of *data* is typically obtained from one of the 64-bit floating point math functions provided. See the entry for AddR64 for an example of the intended use of this function.

FUNCTION TYPE

This function is passive.

RETURN TYPE

```
cstring
```

EXAMPLE

```
Result := AsTextR64WithFormat("17", data[0])
```

ATAN(*VALUE*)

ARGUMENT	TYPE	DESCRIPTION
value	**float**	The value to be processed.

DESCRIPTION

Returns the angle *theta* in radians such that tan(*theta*) is equal to *value*.

FUNCTION TYPE

This function is passive.

RETURN TYPE

float

EXAMPLE

```
theta := atan(1.0)
```

ATAN2(*A, B*)

ARGUMENT	TYPE	DESCRIPTION
a	**float**	The value of the side that is opposite the angle theta.
b	**float**	The value of the side that is adjacent to the angle theta

DESCRIPTION

This function is equivalent to `atan(a/b)`, except that it also considers the sign of *a* and *b*, and thereby ensures that the return value is in the appropriate quadrant. It is also capable of handling a zero value for *b*, thereby avoiding the infinity that would result if the single-argument form of `tan` were used instead.

FUNCTION TYPE

This function is passive.

RETURN TYPE

`float`

EXAMPLE

```
theta := atan2(1,1)
```

ATANR64(RESULT, TAG)

ARGUMENT	TYPE	DESCRIPTION
result	**int**	The result.
tag	**int**	The value to be processed.

DESCRIPTION

Calculates the arctangent of *tag* using 64-bit (double precision) floating point math and stores the result in *result*.

The input operand *tag* should be obtained from one of the 64-bit conversion functions provided or from a driver that can read double precision values. All arguments to this function must be integer arrays with lengths of 2. An in depth example is provided in the entry for AddR64.

FUNCTION TYPE

This function is active.

RETURN TYPE

```
void
```

EXAMPLE

```
atanR64(result[0], tag[0])
```

ATAN2R64(RESULT, A, B)

ARGUMENT	TYPE	DESCRIPTION
result	**int**	The result.
a	**int**	The value of the side that is opposite the angle theta.
b	**int**	The value of the side that is adjacent to the angle theta.

DESCRIPTION

The equivalent of *atan*(*a/b*) using 64-bit (double precision) floating point math and stores the result in *result*. This function considers the sign of *a* and *b* to calculate the value for the appropriate quadrant. It is the double precision equivalent of the atan2 function.

The input operands *a* and *b* should be obtained from one of the 64-bit conversion functions provided or from a driver that can read double precision values. All arguments to this function must be integer arrays with lengths of 2. An in depth example is provided in the entry for AddR64.

FUNCTION TYPE

This function is active.

RETURN TYPE

```
void
```

EXAMPLE

```
Atan2R64(result[0], a[0], b[0])
```

BEEP(*FREQ*, *PERIOD*)

ARGUMENT	TYPE	DESCRIPTION
freq	**int**	The required frequency in semitones.
period	**int**	The required period in milliseconds.

DESCRIPTION

Sounds the terminal's beeper for the indicated period at the indicated pitch. Passing a value of zero for *period* will turn off the beeper. Beep requests are not queued, so calling the function will immediately override any previous calls. For those of you with a musical bent, the *freq* argument is calibrated in semitones. On a more serious "note", the Beep function can be a useful debugging aid, as it provides an asynchronous method of signaling the handling of an event, or the execution of a program step.

FUNCTION TYPE

This function is active.

RETURN TYPE

This function does not return a value.

EXAMPLE

```
Beep(60, 100)
```

CANGOTONEXT()

ARGUMENT	TYPE	DESCRIPTION
none		

DESCRIPTION

Returns a true or false value indicating whether a call to `GotoNext()` will produce a page change. A value of false indicates that no further pages exist in the page history buffer.

FUNCTION TYPE

This function is passive.

RETURN TYPE

`int`

CANGOTOPREVIOUS()

ARGUMENT	TYPE	DESCRIPTION
none		

DESCRIPTION

Returns a true or false value indicating whether a call to `GotoPrevious()` will produce a page change. A value of false indicates that no further pages exist in the page history buffer.

FUNCTION TYPE

This function is passive.

RETURN TYPE

`int`

CLEAREVENTS()

ARGUMENT	TYPE	DESCRIPTION
none		

DESCRIPTION

Clears the list of events displayed in the event log.

FUNCTION TYPE

This function is active.

RETURN TYPE

This function does not return a value.

EXAMPLE

```
ClearEvents()
```

CLOSEFILE(*FILE*)

ARGUMENT	TYPE	DESCRIPTION
file	**int**	The file handle as returned by **OpenFile**.

DESCRIPTION

Closes a file previously opened in a call to `FileOpen()`.

FUNCTION TYPE

This function is active.

RETURN TYPE

This function does not return a value.

EXAMPLE

```
CloseFile(hFile)
```

COLBLEND(*DATA, MIN, MAX, COL1, COL2*)

ARGUMENT	TYPE	DESCRIPTION
data	**float**	The data value to be used to control the operation.
min	**float**	The minimum value of *data*.
max	**float**	The maximum value of *data*.
col1	**int**	The first color, selected if *data* is equal to *min*.
col2	**int**	The second color, selected if *data* is equal to *max*.

DESCRIPTION

Returns a color created by blending two other colors, with the proportion of each color being based upon the value of *data* relative to the limits specified by *min* and *max*. This function is useful when animating display primitives by changing their colors.

FUNCTION TYPE

This function is passive.

RETURN TYPE

`int`

COLFLASH(*FREQ, COL1, COL2*)

ARGUMENT	TYPE	DESCRIPTION
freq	**int**	The number of times per second to alternate.
col1	**int**	The first color.
col2	**int**	The second color.

DESCRIPTION

Returns an alternating color chosen from *col1* and *col2* that completes a cycle *freq* times per second. This function is useful when animating display primitives by changing their colors.

FUNCTION TYPE

This function is passive.

RETURN TYPE

`int`

COLGETBLUE(COL)

ARGUMENT	TYPE	DESCRIPTION
col	int	The color from which the component is to be selected.

DESCRIPTION

Returns the blue component of the indicated color value. The component is scaled to be in the range 0 to 255, even though Crimson works internally with 5-bit color components.

FUNCTION TYPE

This function is passive.

RETURN TYPE

int

COLGETGREEN(COL)

ARGUMENT	TYPE	DESCRIPTION
col	int	The color from which the component is to be selected.

DESCRIPTION

Returns the green component of the indicated color value. The component is scaled to be in the range 0 to 255, even though Crimson works internally with 5-bit color components.

FUNCTION TYPE

This function is passive.

RETURN TYPE

int

COLGETRED(COL)

ARGUMENT	TYPE	DESCRIPTION
col	int	The color from which the component is to be selected.

DESCRIPTION

Returns the red component of the indicated color value. The component is scaled to be in the range 0 to 255, even though Crimson works internally with 5-bit color components.

FUNCTION TYPE

This function is passive.

RETURN TYPE

int

COLGETRGB(R,G,B)

ARGUMENT	TYPE	DESCRIPTION
r	int	The red component.
g	int	The green component.
b	int	The blue component.

DESCRIPTION

Returns a color value constructed from the specified components. The components should be in the range 0 to 255, even though Crimson works internally with 5-bit color components.

FUNCTION TYPE

This function is passive.

RETURN TYPE

int

COLPICK2(*PICK, COL1, COL2*)

ARGUMENT	TYPE	DESCRIPTION
pick	**int**	The condition to be used to select the color.
col1	**int**	The first color, selected if *pick* is true.
col2	**int**	The second color, selected if *pick* is false.

DESCRIPTION

Returns one of the indicated colors, depending on the state of *pick*. Equivalent results can be achieved using the `?:` selection operator.

FUNCTION TYPE

This function is passive.

RETURN TYPE

`int`

COLPICK4(*DATA1, DATA2, COL1, COL2, COL3, COL4*)

ARGUMENT	TYPE	DESCRIPTION
data1	**int**	The first data value.
data2	**int**	The second data value.
col1	**int**	The value when both Data1 and Data2 are *false*.
col2	**int**	The value when Data1 is *true* and Data2 is *false*.
col3	**int**	The value when Data1 is *false* and Data2 is *true*.
col4	**int**	The value when both Data1 and Data2 are *true*.

DESCRIPTION

Returns one of fours values, based on the *true* or *false* status of two data items.

FUNCTION TYPE

This function is passive.

RETURN TYPE

int

COLSELFLASH(*ENABLE, FREQ, COL1, COL2, COL3*)

ARGUMENT	TYPE	DESCRIPTION
enable	**int**	A value that must be *true* to enable flashing.
freq	**int**	The frequency at which the flashing should occur.
col1	**int**	The value to be returned if flashing is disabled.
col2	**int**	The first flashing color.
col3	**int**	The second flashing color.

DESCRIPTION

If *enable* is *true*, returns an alternating color chosen from *col2* and *col3* that completes a cycle *freq* times per second. If *enable* is *false*, returns *col1* constantly. This function is useful when animating display primitives by changing their colors.

FUNCTION TYPE

This function is passive.

RETURN TYPE

int

COMMITANDRESET()

ARGUMENT	TYPE	DESCRIPTION
none		

DESCRIPTION

This function will force all retentive tags to be written on the internal flash memory and then will reset the unit. It is designed to be used in conjunction with function that change the configuration of the unit, and that then require a reset in order for the changes to take effect.

FUNCTION TYPE

This function is passive.

RETURN TYPE

This function does not return a value

EXAMPLE

```
CommitAndReset()
```

COMPACTFLASHEJECT()

ARGUMENT	TYPE	DESCRIPTION
none		

DESCRIPTION

Ceases all access of the CompactFlash card, allowing safe removal of the card.

FUNCTION TYPE

This function is active.

RETURN TYPE

This function does not return a value.

EXAMPLE

```
CompactFlashEject()
```

COMPACTFLASHSTATUS()

ARGUMENT	TYPE	DESCRIPTION
none		

DESCRIPTION

Returns the current status of the CompactFlash slot as an integer.

VALUE	STATE	DESCRIPTION
0	Empty	Either no card is installed or the card has been ejected via a call to the `CompactFlashEject` function.
1	Invalid	The card is damaged, incorrectly formatted or not formatted at all. Remember only FAT16 is supported.
2	Checking	The G3 is checking the status of the card. This state occurs when a card is first inserted into the G3.
3	Formatting	The G3 is formatting the card. This state occurs when a format operation is requested by the programming PC.
4	Locked	The operator interface is either writing to the card, or the card is mounted and Windows is accessing the card.
5	Mounted	A valid card is installed, but it is not locked by either the operator interface or Windows.

FUNCTION TYPE

This function is passive.

RETURN TYPE

```
int
```

EXAMPLE

```
d := CompactFlashStatus()
```

COMPU32(TAG1, TAG2)

ARGUMENT	TYPE	DESCRIPTION
tag1	**int**	The tag to be compared.
tag2	**int**	The tag to be compared to.

DESCRIPTION

Compares *tag1* to *tag2* in an unsigned context. Returns one of the following:

-1	tag1 is less than tag2.
0	tag1 is equal to tag2.
1	tag1 is greater than tag2.

FUNCTION TYPE

This function is passive.

RETURN TYPE

```
int
```

EXAMPLE

```
Result := CompU32(tag1, tag2)
```

CONTROLDEVICE(*DEVICE, ENABLE*)

ARGUMENT	TYPE	DESCRIPTION
device	**int**	The device to be enabled or disabled.
enable	**int**	Determines if device is enabled or disabled.

DESCRIPTION

Allows the database to disable or enable a specified communications device. The number to be placed in the *device* argument to identify the device can be viewed in the status bar of the Communications category when the device name is highlighted.

FUNCTION TYPE

This function is active.

RETURN TYPE

This function does not return a value.

EXAMPLE

```
ControlDevice(1, true)
```

COPY(*DEST*, *SRC*, *COUNT*)

ARGUMENT	TYPE	DESCRIPTION
dest	**int** / **float**	The first array element to be copied to.
src	**int** / **float**	The first array element to be copied from.
count	**int**	The number of elements to be processed.

DESCRIPTION

Copies *count* array elements from *src* onwards to *dest* onwards.

FUNCTION TYPE

This function is active.

RETURN TYPE

This function does not return a value.

EXAMPLE

```
Copy(Save[0], Work[0], 100)
```

COPYFILES(*SOURCE, TARGET, FLAGS*)

ARGUMENT	TYPE	DESCRIPTION
`source`	`cstring`	The path from which the files are to be copied.
`target`	`cstring`	The path to which the files are to be copied.
`flags`	`int`	The flags controlling the copying operation.

DESCRIPTION

Copies all the files in the *source* directory to the *target* directory.

The various bits in `flags` modify the copy operation…

BIT	WEIGHT	DESCRIPTION
0	1	If set, the operation will recurse into any subdirectories.
1	2	If set, existing files will be overwritten. If clear, existing files will be left untouched.
2	4	If set, all files will be copied. If clear, only files that do not exist at the destination or that have newer time stamps at the source will be copied.

The return value of the function will be *true* for success, or *false* for failure.

FUNCTION TYPE

This function is active.

RETURN TYPE

int

EXAMPLE

```
CopyFiles("C:\LOGS", "C:\BACKUP", 1)
```

COS(*THETA*)

ARGUMENT	TYPE	DESCRIPTION
theta	**float**	The angle, in radians, to be processed.

DESCRIPTION

Returns the cosine of the angle *theta*.

FUNCTION TYPE

This function is passive.

RETURN TYPE

```
float
```

EXAMPLE

```
xp := radius*cos(theta)
```

COSR64(RESULT, TAG)

ARGUMENT	TYPE	DESCRIPTION
result	**int**	The result.
tag	**int**	The angle, in radians, to be processed.

DESCRIPTION

Calculates the cosine of *tag* using 64-bit (double precision) floating point math and stores the result in *result*.

The input operand *tag* should be obtained from one of the 64-bit conversion functions provided or from a driver that can read double precision values. All arguments to this function must be integer arrays with lengths of 2. An in depth example is provided in the entry for AddR64.

FUNCTION TYPE

This function is active.

RETURN TYPE

```
void
```

EXAMPLE

```
cosR64(result[0], tag[0])
```

CREATEDIRECTORY(*NAME*)

ARGUMENT	TYPE	DESCRIPTION
name	**cstring**	The directory to be created.

DESCRIPTION

Creates a new directory on the CompactFlash card. Note that the filing system used on the card does not support long filenames, and that if backslashes are included in the pathname to separate path elements, they must be doubled-up per Crimson's rules for string constants as described in the chapter on Writing Expressions. To avoid this complication, forward slashes can be used in place of backslashes without the need for such doubling. The function returns a value of one if it succeeds, and a value of zero if it fails.

FUNCTION TYPE

This function is active.

RETURN TYPE

```
int
```

EXAMPLE

```
Result := CreateDirectory("/LOGS/LOG1")
```

CREATEFILE(*NAME*)

ARGUMENT	TYPE	DESCRIPTION
name	**cstring**	The file to be created.

DESCRIPTION

Creates an empty file on CompactFlash. Note that the filing system used on the card does not support long filenames, and that if backslashes are included in the pathname to separate path elements, they must be doubled-up per Crimson's rules for string constants as described in the chapter on Writing Expressions. To avoid this complication, forward slashes can be used in place of backslashes without the need for such doubling. The function returns a value of one if it succeeds, and a value of zero if it fails. Note that the file is not opened after it is created—a subsequent call to `OpenFile()` must be made to read or write data.

FUNCTION TYPE

This function is active.

RETURN TYPE

```
int
```

EXAMPLE

```
Success := CreateFile("/logs/custom/myfile.txt")
```

DATATOTEXT(*DATA, LIMIT*)

ARGUMENT	TYPE	DESCRIPTION
data	**int**	The first element in an array.
limit	**int**	The number of characters to process.

DESCRIPTION

Forms a string from an array (4 characters per numeric array element) until either the limit is reached or a null character is detected.

FUNCTION TYPE

This function is passive.

RETURN TYPE

```
cstring
```

EXAMPLE

```
string := DataToText(Data[0], 8)
```

DATE(*Y, M, D*)

ARGUMENT	TYPE	DESCRIPTION
y	**int**	The year to be encoded, in four-digit form.
m	**int**	The month to be encoded, from 1 to 12.
d	**int**	The date to be encoded, from 1 upwards.

DESCRIPTION

Returns a value representing the indicated date as the number of seconds elapsed since the datum point of 1st January 1997. This value can then be used with other time/date functions.

FUNCTION TYPE

This function is passive.

RETURN TYPE

```
int
```

EXAMPLE

```
t := Date(2000,12,31)
```

DECR64(RESULT, TAG)

ARGUMENT	TYPE	DESCRIPTION
result	**int**	The result.
tag	**int**	The value to be processed.

DESCRIPTION

Decrements the value of *tag* by one using 64-bit (double precision) floating point math and stores the result in *result*. This is the double precision equivalent of the -- operator.

The input operand *tag* should be obtained from one of the 64-bit conversion functions provided or from a driver that can read double precision values. All arguments to this function must be integer arrays with lengths of 2. An in depth example is provided in the entry for AddR64.

FUNCTION TYPE

This function is active.

RETURN TYPE

```
void
```

EXAMPLE

```
DecR64(result[0], tag[0])
```

DECTOTEXT(*DATA, SIGNED, BEFORE, AFTER, LEADING, GROUP*)

ARGUMENT	TYPE	DESCRIPTION
data	**int/float**	The numeric data to be formatted.
signed	**int**	0 – unsigned, 1 – soft sign, 2 – hard sign.
before	**int**	The number of digits to the left of the decimal point.
after	**int**	The number of digits to the right of the decimal point.
leading	**int**	0 – no leading zeros, 1 – leading zeros.
group	**int**	0 – no grouping, 1– group digits in threes.

DESCRIPTION

Formats the value in *data* as a decimal value according to the rest of the parameters. The function is typically used to generate advanced formatting option via programs, or to prepare strings to be sent via a raw port driver.

FUNCTION TYPE

This function is passive.

RETURN TYPE

```
cstring
```

EXAMPLE

```
Text := DecToText(var1, 2, 5, 2, 0, 1)
```

DEG2RAD(*THETA*)

ARGUMENT	TYPE	DESCRIPTION
theta	**float**	The angle to be processed.

DESCRIPTION

Returns *theta* converted from degrees to radians.

FUNCTION TYPE

This function is passive.

RETURN TYPE

```
float
```

EXAMPLE

```
Load := Weight * cos(Deg2Rad(Angle))
```

DELETEDIRECTORY(*NAME*)

ARGUMENT	TYPE	DESCRIPTION
name	**cstring**	The directory to be deleted.

DESCRIPTION

Remove a directory, its subdirectories and contents from the CompactFlash. Note that the filing system used on the card does not support long filenames, and that if backslashes are included in the pathname to separate path elements, they must be doubled-up per Crimson's rules for string constants as described in the chapter on Writing Expressions. To avoid this complication, forward slashes can be used in place of backslashes without the need for such doubling. The function returns a value of one if it succeeds, and a value of zero if it fails.

FUNCTION TYPE

This function is active.

RETURN TYPE

```
int
```

EXAMPLE

```
Success := DeleteDirectory("/logs/custom")
```

DELETEFILE(*FILE*)

ARGUMENT	TYPE	DESCRIPTION
file	**int**	The file handle as returned by OpenFile.

DESCRIPTION

Closes and then deletes a file located on the CompactFlash card.

FUNCTION TYPE

This function is active.

RETURN TYPE

```
int
```

EXAMPLE

```
Result := DeleteFile(hFile)
```

DEVCTRL(*DEVICE, FUNCTION, DATA*)

ARGUMENT	TYPE	DESCRIPTION
device	**int**	The index of the device to be controlled.
function	**int**	The required function to be executed.
data	**cstring**	Any parameter for the function.

DESCRIPTION

This function is used to perform a special operation on a communications device. The number to be placed in the *device* argument to identify the device can be viewed in the status bar of the Communications category when the device name is highlighted. The specific action to be performed is indicated by the *function* parameter, the values of which will depend upon the type of device being addresses. The *data* parameter may be used to pass addition information to the driver. Most drives do not support this function. Where supported, the operations are driver-specific, and are documented separately.

FUNCTION TYPE

This function is active.

RETURN TYPE

int

EXAMPLE

Refer to comms driver application notes for specific examples.

DISABLEDEVICE(*DEVICE*)

ARGUMENT	TYPE	DESCRIPTION
device	**int**	The device to be disabled.

DESCRIPTION

Disables communications for the specified device. The number to be placed in the *device* argument to identify the device can be viewed in the status bar of the Communications category when the device name is highlighted.

FUNCTION TYPE

The function is passive.

RETURN TYPE

This function does not return a value.

EXAMPLE

```
DisableDevice(1)
```

DISPOFF()

ARGUMENT	TYPE	DESCRIPTION
`none`	`float`	Turns backlight to display off.

DESCRIPTION

Turns backlight to display off.

FUNCTION TYPE

This function is active.

RETURN TYPE

This function does not return a value.

EXAMPLE

```
DispOff()
```

DISPON()

ARGUMENT	TYPE	DESCRIPTION
none		Turns backlight to display on..

DESCRIPTION

Turns backlight to display on.

FUNCTION TYPE

This function is active.

RETURN TYPE

This function does not return a value.

EXAMPLE

```
DispOn()
```

DIVR64(RESULT, TAG1, TAG2)

ARGUMENT	TYPE	DESCRIPTION
result	**int**	The result.
tag1	**int**	The dividend.
tag2	**int**	The divisor.

DESCRIPTION

Calculates the value of *tag1* divided by *tag2* using 64-bit (double precision) floating point math and stores the result in *result.* This is the double precision equivalent of *tag1 / tag2*.

The input operands *tag1* and *tag2* should be obtained from one of the 64-bit conversion functions provided or from a driver that can read double precision values. All arguments to this function must be integer arrays with lengths of 2. An in depth example is provided in the entry for AddR64.

FUNCTION TYPE

This function is active.

RETURN TYPE

```
void
```

EXAMPLE

```
DivR64(result[0], tag1[0], tag2[0])
```

DIVU32(TAG1, TAG2)

ARGUMENT	TYPE	DESCRIPTION
tag1	**int**	The dividend tag.
tag2	**int**	The divisor tag.

DESCRIPTION

Returns the value of *tag1* divided by *tag2* in an unsigned context.

FUNCTION TYPE

This function passive.

RETURN TYPE

```
int
```

EXAMPLE

```
Result := DivU32(tag1, tag2)
```

DRVCTRL(*PORT, FUNCTION, DATA OR VALUE???*)

ARGUMENT	TYPE	DESCRIPTION
port	**int**	The index of the driver to be controlled.
function	**int**	The required function to be executed.
data	**cstring**	Any parameter for the function.

DESCRIPTION

This function is used to perform a special operation on a communications driver. The number to be placed in the *port* argument to identify the driver is the port number to which the driver is bound. The specific action to be performed is indicated by the *function* parameter, the values of which will depend upon the driver itself. The *data* parameter may be used to pass addition information to the driver. Most drivers do not support this function. Where supported, the operations are driver-specific, and are documented separately.

FUNCTION TYPE

This function is active.

RETURN TYPE

int

EXAMPLE

Refer to comms driver application notes for specific examples.

EJECTDRIVE(*DRIVE*)

ARGUMENT	TYPE	DESCRIPTION
drive	**int**	The drive letter of the drive to be ejected.

DESCRIPTION

Ejects a removable drive attached to the system, allowing safe removal of the device.

Drive C refers to the CompactFlash card, while Drive D refers to the USB memory stick.

FUNCTION TYPE

This function is active.

RETURN TYPE

This function does not return a value.

EXAMPLE

```
EjectDrive('C')
```

EMPTYWRITEQUEUE (*DEV*)

ARGUMENT	TYPE	DESCRIPTION
dev	**int**	The device number

DESCRIPTION

Empties the writing queue for the device identified with the argument `dev`. This will remove any pendant writes to the device from the queue, therefore the removed information will not be transferred to the device. The device number can be identified in Crimson's status bar when a device is selected in Communication.

FUNCTION TYPE

This function is passive.

RETURN TYPE

This function does not return a value.

EXAMPLE

```
EmptyWriteQueue(1)
```

ENABLEDEVICE(*DEVICE*)

ARGUMENT	TYPE	DESCRIPTION
device	**int**	The device to be enabled.

DESCRIPTION

Enables communications for the specified device. The number to be placed in the *device* argument to identify the device can be viewed in the status bar of the Communications category when the device name is highlighted.

FUNCTION TYPE

This function is passive.

RETURN TYPE

This function does not return a value.

EXAMPLE

```
EnableDevice(1)
```

ENDBATCH()

ARGUMENT	TYPE	DESCRIPTION
none		

DESCRIPTION

Stops the current batch.

Note: Starting a new batch within less than 10 seconds of ending or starting the last one will produce undefined behavior. To go straight from one batch to another, call *NewBatch()* without an intervening call to *EndBatch()*.

FUNCTION TYPE

This function is passive.

RETURN TYPE

This function does not return a value

EXAMPLE

```
Result := EndBatch()
```

ENDMODAL(*CODE*)

ARGUMENT	TYPE	DESCRIPTION
code	**int**	The value to be returned to the caller of `ShowModal`.

DESCRIPTION

Modal popups displayed using the `ShowModal()` function are displayed immediately. The `ShowModal()` function will not return until an action on the popup page calls `EndModal()`, at which point the value passed the latter will be returned to the caller of the former.

FUNCTION TYPE

This function is active.

RETURN TYPE

This function does not return a value.

ENUMOPTIONCARD(*S*)

ARGUMENT	TYPE	DESCRIPTION
`s`	`int`	The option card slot, either 0 or 1.

DESCRIPTION

Returns the type of the option card configured for the indicated slot.

The following values may be returned…

VALUE	CARD TYPE
0	None
1	Serial
2	CAN
3	Profibus
4	FireWire
5	DeviceNet
6	CAT Link
7	Modem
8	MPI
9	Ethernet
10	USB Host

FUNCTION TYPE

This function is passive.

RETURN TYPE

int

EQUALR64(A, B)

ARGUMENT	TYPE	DESCRIPTION
a	int	The first value to compare.
b	int	The second value to compare.

DESCRIPTION

Compares the value of *a* to *b* using 64-bit (double precision) floating point math and returns 1 if *a* is equal to *b* and 0 otherwise. This is the double precision equivalent of a == b. Note that comparing floating point values for exact equality can be error prone due to rounding errors.

The input operands *a* and *b* should be obtained from one of the 64-bit conversion functions provided or from a driver that can read double precision values. All arguments to this function must be integer arrays with lengths of 2. An in depth example is provided in the entry for AddR64.

FUNCTION TYPE

This function is passive.

RETURN TYPE

```
int
```

EXAMPLE

```
EqualR64(a[0], b[0])
```

EXP(*VALUE*)

ARGUMENT	TYPE	DESCRIPTION
value	**float**	The value to be processed.

DESCRIPTION

Returns *e* (2.7183) raised to the power of *value*.

FUNCTION TYPE

This function is passive.

RETURN TYPE

```
float
```

EXAMPLE

```
Variable2 := exp(1.609)
```

EXP10(*VALUE*)

ARGUMENT	TYPE	DESCRIPTION
value	**float**	The value to be processed.

DESCRIPTION

Returns 10 raised to the power of *value*.

FUNCTION TYPE

This function is passive.

RETURN TYPE

```
float
```

EXAMPLE

```
Variable4 := exp10(0.699)
```

EXPR64(RESULT, TAG)

ARGUMENT	TYPE	DESCRIPTION
result	**int**	The result.
tag	**int**	The value to be processed.

DESCRIPTION

Calculates the value of Euler's constant, *e* (2.7183…) raised to the power of *tag* using 64-bit (double precision) floating point math and stores the result in *result*.

The input operand *tag* should be obtained from one of the 64-bit conversion functions provided or from a driver that can read double precision values. All arguments to this function must be integer arrays with lengths of 2. An in depth example is provided in the entry for AddR64.

FUNCTION TYPE

This function is active.

RETURN TYPE

```
void
```

EXAMPLE

```
expR64(result[0], tag[0])
```

FILESEEK(*FILE, POS*)

ARGUMENT	TYPE	DESCRIPTION
file	**int**	The file handle as returned by `OpenFile`.
pos	**int**	The position within the file.

DESCRIPTION

Moves the file pointer for the specified file to the indicated location.

FUNCTION TYPE

This function is active.

RETURN TYPE

int

FILETELL(*FILE*)

ARGUMENT	TYPE	DESCRIPTION
file	**int**	The file handle as returned by `OpenFile`.

DESCRIPTION

Returns the current value of the file pointer for the specified file.

FUNCTION TYPE

This function is passive.

RETURN TYPE

int

FILL(*ELEMENT, DATA, COUNT*)

ARGUMENT	TYPE	DESCRIPTION
element	**int** / **float**	The first array element to be processed.
data	**int** / **float**	The data value to be written.
count	**int**	The number of elements to be processed.

DESCRIPTION

Sets *count* array elements from *element* onwards to be equal to *data*.

FUNCTION TYPE

This function is active.

RETURN TYPE

This function does not return a value.

EXAMPLE

```
Fill(List[0], 0, 100)
```

FIND(*STRING,CHAR,SKIP*)

ARGUMENT	TYPE	DESCRIPTION
string	**cstring**	The string to be processed.
char	**int**	The character to be found.
skip	**int**	The number of times the character is skipped.

DESCRIPTION

Returns the position of ***char*** in ***string***, taking into account the number of ***skip*** occurrences specified. The first position counted is 0. Returns -1 if ***char*** is not found. In the example below, the position of ":", skipping the first occurrence is 7.

FUNCTION TYPE

This function is passive.

RETURN TYPE

int

EXAMPLE

```
Position := Find("one:two:three",':',1)
```

FINDFILEFIRST(*DIR*)

ARGUMENT	TYPE	DESCRIPTION
dir	**cstring**	Directory to be used in search.

DESCRIPTION

Returns the filename of name of the first file or directory located in the *dir* directory on the CompactFlash card. Returns an empty string if no files exist or if no card is present. This function can be used with the `FindFileNext` function to scan all files in a given directory.

FUNCTION TYPE

This function is active.

RETURN TYPE

```
cstring
```

EXAMPLE

```
Name := FindFileFirst("/LOGS/LOG1")
```

FINDFILENEXT()

ARGUMENT	TYPE	DESCRIPTION
none		

DESCRIPTION

Returns the filename of the next file or directory in the directory specified in a previous call to the `FindFileFirst` function. Returns and empty string if no more files exist. This function can be used with the `FindFileFirst` function to scan all files in a given directory.

FUNCTION TYPE

This function is active.

RETURN TYPE

```
cstring
```

EXAMPLE

```
Name := FindFileNext()
```

FINDTAGINDEX(*LABEL*)

ARGUMENT	TYPE	DESCRIPTION
label	**cstring**	The tag label (not tag name or mnemonic)

DESCRIPTION

Returns the index number of the tag specified by *label*.

FUNCTION TYPE

This function is active.

RETURN TYPE

```
int
```

EXAMPLE

```
Index = FindTagIndex("Power")
```

Returns the index number for the tag with label *Power*.

FLASH(*FREQ*)

ARGUMENT	TYPE	DESCRIPTION
freq	**Int**	The number of times per second to flash.

DESCRIPTION

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.

FUNCTION TYPE

This function is passive.

RETURN TYPE

```
int
```

FORCE(*DEST, DATA*)

ARGUMENT	TYPE	DESCRIPTION
`dest`	`int / float`	The tag to be changed.
`data`	`int / float`	The value to be written.

DESCRIPTION

This function sets the specified tag to the specified value. It differs from the more normally used assignment operator in that it (a) deletes any queued writes to this tag and replaces them with an immediate write of the specified value; and (b) forces a write to the remote comms device whether or not the data value has changed. It is thus used in situations where Crimson's normal write behavior is not required.

FUNCTION TYPE

This function is active.

RETURN TYPE

This function does not return a value.

FORCECOPY(*DEST, SRC, COUNT*)

ARGUMENT	TYPE	DESCRIPTION
`dest`	`int / float`	The first array element to be copied to.
`src`	`int / float`	The first array element to be copied from.
`count`	`int`	The number of elements to be processed.

DESCRIPTION

Copies *count* array elements from *src* onwards to *dest* onwards. The semantics used are the same as for the `Force()` function, thereby bypassing the write queue and forcing a write whether or not the original data has changed.

FUNCTION TYPE

This function is active.

RETURN TYPE

This function does not return a value.

FORCESQLSYNC()

ARGUMENT	TYPE	DESCRIPTION
None		

DESCRIPTION

Forces the SQL Sync service to run immediately and transmit log data to the configured SQL Server. Only works if the "Manual Sync" property of the SQL Sync service has been set to "Yes".

FUNCTION TYPE

This function is active.

RETURN TYPE

This function does not return a value.

EXAMPLE

```
ForceSQLSync()
```

FORMATCOMPACTFLASH()

ARGUMENT	TYPE	DESCRIPTION
none		

DESCRIPTION

Formats the CompactFlash card in the terminal, thereby deleting all data on the card. You should thus ensure that the user is given appropriate warnings before this function is invoked.

FUNCTION TYPE

This function is active.

RETURN TYPE

This function does not return a value.

EXAMPLE

```
FormatCompactFlash()
```

FORMATDRIVE(*DRIVE*)

ARGUMENT	TYPE	DESCRIPTION
drive	**int**	The letter of the drive to be formatted.

DESCRIPTION

Formats a removable drive attached to the system, deleting all data that it contains.

Drive C refers to the CompactFlash card, while Drive D refers to the USB memory stick.

FUNCTION TYPE

This function is active.

RETURN TYPE

This function does not return a value.

EXAMPLE

```
FormatDrive('C')
```

FTPGETFILE(*SERVER, LOC, REM, DELETE*)

ARGUMENT	TYPE	DESCRIPTION
server	**int**	The FTP connection number, always 0
loc	**cstring**	The local file name on the CompactFlash card
rem	**cstring**	The remote file name on the FTP server
delete	**int**	If true, the source will be deleted after the transfer, otherwise, it will remain on the source disk.

DESCRIPTION

This function will transfer the defined file from the FTP server to the operator interface CompactFlash card. It will return true if the transfer is successful, false otherwise. The source and destination file name can be different. The remote path is relative to the FTP server setting root path (See Synchronization Manager for details).

FUNCTION TYPE

This function is passive.

RETURN TYPE

```
int
```

EXAMPLE

```
Success = FtpGetFile(0, "/Recipes.csv", "/Recipes/Rec001.csv", 0)
```

In this example, the file Recipes.csv will be transferred from the FTP server to the CompactFlash Card. The original file will not be deleted from the PC server.

FTPPUTFILE(*SERVER, LOC, REM, DELETE*)

ARGUMENT	TYPE	DESCRIPTION
server	**int**	The FTP connection number, always 0
loc	**cstring**	The local file name on the CompactFlash card
rem	**cstring**	The remote file name on the FTP server
delete	**int**	If true, the source will be deleted after the transfer, otherwise, it will remain on the source disk.

DESCRIPTION

This function will transfer the defined file from the operator interface CompactFlash card to the FTP server. It will return true if the transfer is successful, false otherwise. The source and destination file name can be different. The remote path is relative to the FTP server setting root path (See Synchronization Manager for details).

FUNCTION TYPE

This function is passive.

RETURN TYPE

```
int
```

EXAMPLE

```
Success = FtpPutFile(0, "/LOGS/Report.txt", "/Reports/Report.txt", 1)
```

In this example, the file Report.txt will be sent to the FTP server and deleted from the CompactFlash Card upon success of the transfer.

GETALARMTAG(*INDEX*)

ARGUMENT	TYPE	DESCRIPTION
index	**int**	The tag index number

DESCRIPTION

This function returns a bit mask integer representing the tag alarms state for the tag identified with `index`. Bit 0 (ie. the bit with a value of 0x01) represents the Alarm 1 state and bit 1 (ie. the bit with a value of 0x02) the Alarm 2.

Note: The tag index can be found from the tag name using the `FindTagIndex()` function

FUNCTION TYPE

This function is passive.

RETURN TYPE

```
int
```

EXAMPLE

```
AlarmsInTag = GetAlarmTag(12)
```

In this example, the function returns the states of Alarm 1 and 2 for the tag with index 12.

GETAUTOCOPYSTATUSCODE()

ARGUMENT	TYPE	DESCRIPTION
none		

DESCRIPTION

Returns a value indicating the status of the synchronization operation that can optionally occur when a USB memory stick is inserted into the Crimson device. The possible values are their meanings are shown below…

VALUE	DESCRIPTION
0	Synchronization is not enabled.
1	The synchronization task is initializing.
2	The task is waiting for a memory stick to be inserted.
3	The task is copying the required files.
4	The task has completed, and is waiting for the stick to be removed.

FUNCTION TYPE

This function is passive.

RETURN TYPE

int

GETAUTOCOPYSTATUSTEXT()

ARGUMENT	TYPE	DESCRIPTION
none		

DESCRIPTION

Returns a string equivalent to the status code returned by `GetAutoCopyStatus()`.

FUNCTION TYPE

This function is passive.

RETURN TYPE

`Cstring`

GETBATCH()

ARGUMENT	TYPE	DESCRIPTION
none		

DESCRIPTION

Returns the name of the current batch.

FUNCTION TYPE

This function is passive.

RETURN TYPE

```
cstring
```

EXAMPLE

```
CurrentBatch := GetBatch()
```

GETCAMERADATA(*PORT, CAMERA, PARAM*)

ARGUMENT	TYPE	DESCRIPTION
port	**int**	The port number where the camera is connected
camera	**int**	The camera number on the port
param	**int**	The camera parameter to be read

DESCRIPTION

This function returns the value of the parameter number *param* for a Banner camera connected on the operator interface. The argument *camera* is the device number showing in Crimson 2.0 status bar when the camera is selected. More than one camera can be connected under the driver. The number to be placed in the *port* argument is the port number to which the driver is bound. Please see Banner documentation for parameter numbers and details.

FUNCTION TYPE

This function is active.

RETURN TYPE

```
int
```

EXAMPLE

```
Value = GetCameraData(4, 0, 1)
```

Returns parameter 1 on camera device number 0 connected on port 4.

GETCURRENTUSERNAME()

ARGUMENT	TYPE	DESCRIPTION
none		

DESCRIPTION

Returns the current user name, or an empty string is no user if logged on. Note that displaying the current user name may prejudice security in situations where user names are not commonly known. Care should thus be used in high-security applications.

FUNCTION TYPE

This function is passive.

RETURN TYPE

```
cstring
```

GETCURRENTUSERREALNAME()

ARGUMENT	TYPE	DESCRIPTION
none		

DESCRIPTION

Returns the real name of the current user, or an empty string if no user is logged on.

FUNCTION TYPE

This function is passive.

RETURN TYPE

cstring

GETCURRENTUSERRIGHTS()

ARGUMENT	TYPE	DESCRIPTION
none		

DESCRIPTION

Returns the user rights of the current user, as defined for the `HasAccess`() function.

FUNCTION TYPE

This function is passive.

RETURN TYPE

int

GETDATE (*TIME*) AND FAMILY

ARGUMENT	TYPE	DESCRIPTION
time	**int**	The time value to be decoded.

DESCRIPTION

Each member of this family of functions returns some component of a time/date value, as previously created by `GetNow`, `Time` or `Date`. The available functions are as follows…

FUNCTION	DESCRIPTION
GetDate	Returns the day-of-month portion of *time*.
GetDay	Returns the day-of-week portion of *time*.
GetDays	Returns the number of days in *time*.
GetHour	Returns the hours portion of *time*.
GetMin	Returns the minutes portion of *time*.
GetMonth	Returns the month portion of *time*.
GetSec	Returns the seconds portion of *time*.
GetWeek	Returns the week-of-year portion of *time*.
GetWeeks	Returns the number of weeks in *time*.
GetWeekYear	Returns the week year when using week numbers.
GetYear	Returns the year portion of *time*.

Note that `GetDays` and `GetWeeks` are typically used with the difference between two time values to calculate how long has elapsed in terms of days or weeks. Note also that the year returned by *GetWeekYear* is not always the same as that returned by *GetYear*, as the former may return a smaller value if the last week of a year extends beyond year-end.

FUNCTION TYPE

These functions are passive.

RETURN TYPE

```
int
```

EXAMPLE

```
d := GetDate(GetNow() - 12*60*60)
```

GETDEVICESTATUS(*DEVICE*)

ARGUMENT	TYPE	DESCRIPTION
`device`	`int`	The comms device to be queried.

DESCRIPTION

Returns the communications status of the specific comms device.

The bottom two bits encode the device's error state…

VALUE	DESCRIPTION
0	The device comms is initializing.
1	The device comms is operating correctly.
2	The device comms has one or more soft errors.
3	The device comms has encountered a fatal error.

The following hexadecimal values encode further information about the device…

VALUE	DESCRIPTION
0x0010	At least one error exists in the automatic comms blocks.
0x0020	At least one error exists in the gateway comms blocks.
0x0040	Communications to this device are suspended.
0x0100	Some level of response has been received from the device.
0x0200	Some form of error has occurred during communications.
0x1000	The primary write queue is nearly full.
0x2000	The secondary write queue is nearly full.

Note that the 0x0100 value does not imply that comms is working correctly, but merely that some sort of response has been received. It is useful for confirming wiring and so on. In a similar manner, the 0x0200 values does not imply that comms has failed, but indicates that all is not running as smoothly as it should. For example, Crimson's retry mechanism may allow recovery from errors such that comms appears to be operating, but this bit may still indicate that things are not proceeding on an error free basis.

FUNCTION TYPE

This function is passive.

RETURN TYPE

int

GETDISKFREEBYTES(*DRIVE*)

ARGUMENT	TYPE	DESCRIPTION
drive	**int**	The drive number, always 0.

DESCRIPTION

Returns the number of free memory bytes on the CompactFlash Card.

Note: This function requires time to calculate free memory space, as a long CompactFlash access is necessary. Do NOT call this function permanently with on tick, on update or in a formula. Call it upon an event such as *OnSelect* on the page you want to display the resulting value.

FUNCTION TYPE

This function is passive.

RETURN TYPE

```
int
```

EXAMPLE

```
FreeMemory = GetDiskFreeBytes(0)
```

GETDISKFREEPERCENT(*DRIVE*)

ARGUMENT	TYPE	DESCRIPTION
drive	**int**	The drive number, always 0.

DESCRIPTION

Returns the percentage of free memory space on the CompactFlash Card.

Note: This function requires time to calculate free memory space, as a long CompactFlash access is necessary. Do NOT call this function permanently with on tick, on update or in a formula. Call it upon an event such as *OnSelect* on the page you want to display the resulting value.

FUNCTION TYPE

This function is passive.

RETURN TYPE

```
int
```

EXAMPLE

```
FreeMemory = GetDiskFreePercent(0)
```

GETDISKSIZEBYTES(*DRIVE*)

ARGUMENT	TYPE	DESCRIPTION
drive	**int**	The drive number, always 0.

DESCRIPTION

Returns the size in bytes of the CompactFlash Card.

Note: This function requires time to calculate free memory space, as a long CompactFlash access is necessary. Do NOT call this function permanently with on tick, on update or in a formula. Call it upon an event such as *OnSelect* on the page you want to display the resulting value.

FUNCTION TYPE

This function is passive.

RETURN TYPE

```
int
```

EXAMPLE

```
CFSyze = GetDiskSyzeBytes(0)
```

GETDRIVESTATUS(*DRIVE*)

ARGUMENT	TYPE	DESCRIPTION
`drive`	`int`	The drive letter of the drive to be queried.

DESCRIPTION

Returns the current status of the specified drive as an integer.

VALUE	STATE	DESCRIPTION
0	Empty	Either no card is installed or the card has been ejected via a call to the `DriveEject` function.
1	Invalid	The card is damaged, incorrectly formatted or not formatted at all. Remember only FAT16 is supported.
2	Checking	The G3 is checking the status of the card. This state occurs when a card is first inserted into the G3.
3	Formatting	The G3 is formatting the card. This state occurs when a format operation is requested by the programming PC.
4	Locked	The operator interface is either writing to the card, or the card is mounted and Windows is accessing the card.
5	Mounted	A valid card is installed, but it is not locked by either the operator interface or Windows.

Drive C refers to the CompactFlash card, while Drive D refers to the USB memory stick.

FUNCTION TYPE

This function is passive.

RETURN TYPE

int

GETFILEBYTE(*FILE*)

ARGUMENT	TYPE	DESCRIPTION
file	**int**	File handle as returned by `OpenFile`.

DESCRIPTION

Reads a single byte from the indicated file. A value of -1 indicates the end of file.

FUNCTION TYPE

This function is active.

RETURN TYPE

int

GETFILEDATA(*FILE, DATA, LENGTH*)

ARGUMENT	TYPE	DESCRIPTION
file	**int**	The file handle as returned by `OpenFile`.
data	**int**	The first array element at which to store the data.
length	**int**	The number of elements to process.

DESCRIPTION

Reads *length* bytes from the specified file, and stores them in the indicated array elements.

The return value indicates the number of bytes successfully read, and may be less than *length*.

FUNCTION TYPE

This function is active.

RETURN TYPE

int

GETFORMATTEDTAG(*INDEX*)

ARGUMENT	TYPE	DESCRIPTION
index	**int**	Tag index number

DESCRIPTION

Returns a string representing the formatted value of the tag specified by *index*. The string returned follows the format programmed on the targeted tag. For example, a flag will show On or Off, a multi variable will show the text corresponding to the value. The index can be found from the tag label using the function `FindTagIndex()`.This function works with any type of tags.

FUNCTION TYPE

This function is active.

RETURN TYPE

```
cstring
```

EXAMPLE

```
Value = GetFormattedTag(10)
```

Returns the value of the tag with index 10 in a string.

```
Value = GetFormattedTag(FindTagIndex("Power"))
```

Returns the value from the tag with label *Power* in a string.

GETINTERFACESTATUS(*PORT*)

ARGUMENT	TYPE	DESCRIPTION
interface	**int**	The interface to be queried.

DESCRIPTION

Returns a string indicating the status of the specified TCP/IP interface. Refer to the earlier chapter on Advanced Communications for details of how to calculate the value to be placed in the *interface* parameter, and of how to interpret the returned value.

FUNCTION TYPE

This function is passive.

RETURN TYPE

```
cstring
```

EXAMPLE

```
EthernentStatus := GetInterfaceStatus(1)
```

GETINTTAG(*INDEX*)

ARGUMENT	TYPE	DESCRIPTION
index	**int**	The tag index number

DESCRIPTION

Returns the value of the integer tag specified by *index*. The index can be found from the tag label using the function `FindTagIndex()`. This function will only work if the targeted tag is an integer.

FUNCTION TYPE

This function is active.

RETURN TYPE

```
int
```

EXAMPLE

```
Value = GetIntTag(10)
```

Returns the value of the tag with index 10.

```
Value = GetIntTag(FindTagIndex("Power"))
```

Returns the value from the tag with label *Power*.

GETLANGUAGE()

ARGUMENT	TYPE	DESCRIPTION
none		

DESCRIPTION

Returns the currently selected language, as passed to the `SetLanguage()` function.

FUNCTION TYPE

This function is passive.

RETURN TYPE

int

GETLASTEVENTTEXT(*ALL*)

ARGUMENT	TYPE	DESCRIPTION
`all`	`int`	Indicates whether all alarm events should also be included in the definition of the last event.

DESCRIPTION

Returns the label of the last event captured by the event log.

FUNCTION TYPE

This function is passive.

RETURN TYPE

`cstring`

GETLASTEVENTTIME(ALL)

ARGUMENT	TYPE	DESCRIPTION
`all`	`int`	Indicates whether all alarm events should also be included in the definition of the last event.

DESCRIPTION

Returns the time at which the last event capture by the event log occurred. The value can be displayed in human-readable form using a field that has the Time and Date format type.

FUNCTION TYPE

This function is passive.

RETURN TYPE

```
int
```

GETLASTEVENTTIME(ALL)

ARGUMENT	TYPE	DESCRIPTION
`all`	`int`	Indicates whether all alarm events should also be included in the definition of the last event.

DESCRIPTION

Returns the time at which the last event capture by the event log occurred. The value can be displayed in human-readable form using a field that has the Time and Date format type.

FUNCTION TYPE

This function is passive.

RETURN TYPE

```
int
```

GETLASTEVENTTYPE(ALL)

ARGUMENT	TYPE	DESCRIPTION
`all`	`int`	Indicates whether all alarm events should also be included in the definition of the last event.

DESCRIPTION

Returns a string indicating the type of the last event captured by the event logging system.

FUNCTION TYPE

This function is passive.

RETURN TYPE

`cstring`

GETMODELNAME(*CODE*)

ARGUMENT	TYPE	DESCRIPTION
code	int	The name to return. Only 1 is supported at this time.

DESCRIPTION

Returns the name of the hardware platform on which Crimson is executing.

FUNCTION TYPE

This function is passive.

RETURN TYPE

cstring

GETMONTHDAYS(*Y, M*)

ARGUMENT	TYPE	DESCRIPTION
y	**int**	The year to be processed, in four-digit form.
m	**int**	The month to be processed, from 1 to 12.

DESCRIPTION

Returns the number of days in the indicated month, accounting for leap years etc.

FUNCTION TYPE

This function is passive.

RETURN TYPE

```
int
```

EXAMPLE

```
Days := GetMonthDays(2000, 3)
```

GETNETGATE(*PORT*)

ARGUMENT	TYPE	DESCRIPTION
port	**int**	The index of the Ethernet port. Must be zero.

DESCRIPTION

Returns the IP address of the port's default gateway as a dotted-decimal text string.

FUNCTION TYPE

The function is passive.

RETURN TYPE

```
cstring
```

EXAMPLE

```
gate := GetNetGate(0)
```

GETNETID(*PORT*)

ARGUMENT	TYPE	DESCRIPTION
port	**int**	The index of the Ethernet port. Must be zero.

DESCRIPTION

Reports an Ethernet port's MAC address as 17-character text string.

FUNCTION TYPE

This function is passive.

RETURN TYPE

```
cstring
```

EXAMPLE

```
MAC := GetNetId(1)
```

GETNETIP(*PORT*)

ARGUMENT	TYPE	DESCRIPTION
`port`	`int`	The index of the Ethernet port. Must be zero.

DESCRIPTION

Reports an Ethernet port's IP address as a dotted-decimal text string.

FUNCTION TYPE

This function is passive.

RETURN TYPE

```
cstring
```

EXAMPLE

```
IP := GetNetIp(1)
```

GETNETMASK(*PORT*)

ARGUMENT	TYPE	DESCRIPTION
port	**int**	The index of the Ethernet port. Must be zero.

DESCRIPTION

Reports an Ethernet port's IP address mask as a dotted-decimal text string.

FUNCTION TYPE

This function is passive.

RETURN TYPE

```
cstring
```

EXAMPLE

```
mask := GetNetMask(0)
```

GETNOW()

ARGUMENT	TYPE	DESCRIPTION
none		

DESCRIPTION

Returns the current time and date as the number of seconds elapsed since the datum point of 1st January 1997. This value can then be used with other time/date functions.

FUNCTION TYPE

This function is passive.

RETURN TYPE

```
int
```

EXAMPLE

```
t := GetNow()
```

GETNOWDATE()

ARGUMENT	TYPE	DESCRIPTION
none		

DESCRIPTION

Returns the number of seconds in the days that have passed since 1st of January 1997.

FUNCTION TYPE

This function is passive.

RETURN TYPE

```
int
```

*EXAMPLE

```
d: = GetNowDate()
```

GETNOWTIME()

ARGUMENT	TYPE	DESCRIPTION
none		

DESCRIPTION

Returns the time of day in terms of seconds.

FUNCTION TYPE

This function is passive.

RETURN TYPE

```
int
```

EXAMPLE

```
t := GetNowTime()
```

GETPORTCONFIG(*PORT, PARAM*)

ARGUMENT	TYPE	DESCRIPTION
port	**int**	The number of the port to be set
param	**int**	The port parameter to be set

DESCRIPTION

Returns the value of a parameter on port. The port number starts from the programming port with value 1. The table below shows the parameter number and associated return values.

PARAM NB	DESCRIPTION	POSSIBLE VALUES
1	Baud Rate	The actual baud rate, e.g. 115200
2	Data Bits	7, 8 or 9
3	Stop Bits	1 or 2
4	Parity	0 (none), 1 (odd) or 2 (even)
5	Physical Mode	0 (RS232), 1 (422 Master), 2 (422 Slave), 3 (485)

FUNCTION TYPE

This function is active.

RETURN TYPE

```
int
```

EXAMPLE

```
Config = GetPortConfig(2, 4)
```

In this example, *Config* will take the value of the current parity setting on the RS232 communication port.

GETREALTAG(*INDEX*)

ARGUMENT	TYPE	DESCRIPTION
index	**int**	Tag index number

DESCRIPTION

Returns the value of the real tag specified by *index*. The index can be found from the tag label using the function `FindTagIndex()`. This function will only work if the targeted tag is a real (floating point).

FUNCTION TYPE

This function is active.

RETURN TYPE

```
float
```

EXAMPLE

```
Value = GetRealTag(10)
```

Returns the floating-point value of the tag with index 10.

```
Value = GetRealTag(FindTagIndex("Power"))
```

Returns the floating-point value from the tag with label *Power*.

GETRESTARTCODE(*N*)

ARGUMENT	TYPE	DESCRIPTION
n	int	The entry in the restart table, from 0 to 6 inclusive.

DESCRIPTION

Returns the Guru Meditation Code corresponding to the specified restart.

FUNCTION TYPE

This function is passive.

RETURN TYPE

```
cstring
```

GETRESTARTINFO(*N*)

ARGUMENT	TYPE	DESCRIPTION
n	**int**	The entry in the restart table, from 0 to 6 inclusive.

DESCRIPTION

Returns an extended description of the specified restart, complete with time and date stamp.

FUNCTION TYPE

This function is passive.

RETURN TYPE

`cstring`

GETRESTARTTEXT(*N*)

ARGUMENT	TYPE	DESCRIPTION
n	**int**	The entry in the restart table, from 0 to 6 inclusive.

DESCRIPTION

Returns an extended description of the specified restart, without a time and date stamp.

FUNCTION TYPE

This function is passive.

RETURN TYPE

`cstring`

GETRESTARTTIME(*N*)

ARGUMENT	TYPE	DESCRIPTION
`n`	`int`	The entry in the restart table, from 0 to 6 inclusive.

DESCRIPTION

Returns the time at which the specified restart occurred. Not defined for all situations.

FUNCTION TYPE

This function is passive.

RETURN TYPE

int

GETSTRINGTAG(*INDEX*)

ARGUMENT	TYPE	DESCRIPTION
index	**int**	Tag index number

DESCRIPTION

Returns the value of the string tag specified by *index*. The index can be found from the tag label using the function `FindTagIndex()`.This function will only work if the targeted tag is a String.

FUNCTION TYPE

This function is active.

RETURN TYPE

`cstring`

EXAMPLE

```
Value = GetStringTag(10)
```

Returns the string value of the tag with index 10.

```
Value = GetStringTag(FindTagIndex("Name"))
```

Returns the string value from the tag with label *Name*.

GETTAGLABEL(*INDEX*)

ARGUMENT	TYPE	DESCRIPTION
index	**int**	Tag index number

DESCRIPTION

Returns the label of the tag (not the mnemonic or tag name) specified by *index*.

FUNCTION TYPE

This function is active.

RETURN TYPE

```
cstring
```

EXAMPLE

```
Label = GetTagLabel(10)
```

Returns the label of the tag with index 10.

GETUPDOWNDATA(*DATA, LIMIT*)

ARGUMENT	TYPE	DESCRIPTION
data	**int**	A steadily increasing source value.
limit	**int**	The number of values to generate.

DESCRIPTION

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. If the `GetUpDownStep` function is called with the same arguments, it will return a value indicating the direction of change of the data returned by `GetUpDownData`.

FUNCTION TYPE

This function is passive.

RETURN TYPE

```
int
```

EXAMPLE

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

GETUPDOWNSTEP(*DATA, LIMIT*)

ARGUMENT	TYPE	DESCRIPTION
data	**int**	A steadily increasing source value.
limit	**int**	The number of values to generate.

DESCRIPTION

See `GetUpDownData` for a description of this function.

FUNCTION TYPE

This function is passive.

RETURN TYPE

```
int
```

EXAMPLE

```
Delta := GetUpDownStep(DispCount, 100)
```

GETVERSIONINFO(*CODE*)

ARGUMENT	TYPE	DESCRIPTION
`code`	`int`	The item to be returned.

DESCRIPTION

Returns information about the various version numbers…

CODE	DESCRIPTION
1	Returns the boot loader version.
2	Returns the build of the runtime software.
3	Returns the build of configuration software used to prepare the current database.

FUNCTION TYPE

This function is passive.

RETURN TYPE

int

GOTONEXT()

ARGUMENT	TYPE	DESCRIPTION
none		

DESCRIPTION

Causes the panel to move forwards again in the page history buffer, reversing the result of a previous call to `GotoPrevious()`. The portion of the history buffer accessible via this function will be cleared if the `GotoPage()` function is called.

FUNCTION TYPE

This function is active.

RETURN TYPE

This function does not return a value.

EXAMPLE

```
GotoNext()
```

GOTOPAGE(*NAME*)

ARGUMENT	TYPE	DESCRIPTION
name	Display Page	The page to be displayed.

DESCRIPTION

Selects page *name* to be shown on the terminal's display.

FUNCTION TYPE

This function is active.

RETURN TYPE

This function does not return a value.

EXAMPLE

```
GotoPage(Page1)
```

GOTOPREVIOUS()

ARGUMENT	TYPE	DESCRIPTION
none		

DESCRIPTION

Causes the panel to return to the previous page shown on the terminal's display.

FUNCTION TYPE

This function is active.

RETURN TYPE

This function does not return a value.

EXAMPLE

```
GotoPrevious()
```

GREATERR64(A, B)

ARGUMENT	TYPE	DESCRIPTION
a	int	The value to be compared.
b	int	The value to compare to.

DESCRIPTION

Compares the value of *a* to *b* using 64-bit (double precision) floating point math and returns 1 if *a* is greater than *b* and 0 otherwise. This is the double precision equivalent of a > b.

The input operands *a* and *b* should be obtained from one of the 64-bit conversion functions provided or from a driver that can read double precision values. All arguments to this function must be integer arrays with lengths of 2. An in depth example is provided in the entry for AddR64.

FUNCTION TYPE

This function is passive.

RETURN TYPE

```
int
```

EXAMPLE

```
Result := GreaterR64(a[0], b[0])
```

HASACCESS (*RIGHTS*)

ARGUMENT	TYPE	DESCRIPTION
rights	**int**	The required access rights.

DESCRIPTION

Returns a value of **true** or **false** depending on whether the current user has access rights defined by the *rights* parameter. This parameter comprises a bit-mask representing the various user-defined rights, with bit 0 (ie. the bit with a value of 0x01) representing User Right 1, bit 1 (ie. the bit with a value of 0x02) representing User Right 2 and so on. The function is typically used in programs that perform a number of actions that might be subject to security, and that might otherwise not occur.

FUNCTION TYPE

This function is passive.

RETURN TYPE

```
int
```

EXAMPLE

```
if( HasAccess(1)) {
       Data1 := 0;
       Data2 := 0;
       Data3 := 0;
       }
```

HASALLACCESS(*RIGHTS*)

ARGUMENT	TYPE	DESCRIPTION
`rights`	`int`	The required access rights.

DESCRIPTION

Returns a value of *true* or *false* depending on whether the current user has all the access rights defined by the `rights` parameter. This parameter comprises a bit-mask representing the various user-defined rights, with bit 0 (ie. the bit with a value of 0x01) representing User Right 1, bit 1 (ie. the bit with a value of 0x02) representing User Right 2 and so on. The function is typically used in programs that perform a number of actions that might be subject to security, and that might otherwise not occur.

FUNCTION TYPE

This function is passive.

RETURN TYPE

int

HIDEALLPOPUPS()

ARGUMENT	TYPE	DESCRIPTION
none		

DESCRIPTION

Hides any popups, including nested popups, shown by `ShowPopup()` or `ShowNested()`.

FUNCTION TYPE

This function is active.

RETURN TYPE

This function does not return a value.

HIDEPOPUP()

ARGUMENT	TYPE	DESCRIPTION
none		

DESCRIPTION

Hides the popup that was previously shown using `ShowPopup`.

FUNCTION TYPE

This function is active.

RETURN TYPE

This function does not return a value.

EXAMPLE

```
HidePopup()
```

INCR64(RESULT, TAG)

ARGUMENT	TYPE	DESCRIPTION
result	**int**	The result.
tag	**int**	The value to be processed.

DESCRIPTION

Increments the value of *tag* by one using 64-bit (double precision) floating point math and stores the result in *result*. This is the double precision equivalent of the ++ operator.

The input operand *tag* should be obtained from one of the 64-bit conversion functions provided or from a driver that can read double precision values. All arguments to this function must be integer arrays with lengths of 2. An in depth example is provided in the entry for AddR64.

FUNCTION TYPE

This function is active.

RETURN TYPE

```
void
```

EXAMPLE

```
IncR64(result[0], tag[0])
```

INTTOR64(RESULT, N)

ARGUMENT	TYPE	DESCRIPTION
result	**int**	The result.
n	**int**	The value to be converted.

DESCRIPTION

Converts the value stored in *n* from an integer to a 64-bit (double precision) number and stores the result as an array of length 2 in *result.* The tag *result* should therefore be an integer array with an extent of at least 2. After execution of this function, the value stored in *result* is suitable for use in other 64-bit math functions. See the entry for AddR64 for an example of the intended use of this function.

FUNCTION TYPE

This function is active.

RETURN TYPE

```
void
```

EXAMPLE

```
IntToR64(result[0], n)
```

INTTOTEXT(*DATA*, *RADIX*, *COUNT*)

ARGUMENT	TYPE	DESCRIPTION
data	**int**	The value to be processed.
radix	**int**	The number base to be used.
count	**int**	The number of digits to generate.

DESCRIPTION

Returns the string obtained by formatting *data* in base *radix*, generating *count* digits. The value is assumed to be unsigned, so if a signed value is required, use `Sgn` to decide whether to prefix a negative sign, and then use `Abs` to pass the absolute value to `IntToText`.

FUNCTION TYPE

This function is passive.

RETURN TYPE

`cstring`

EXAMPLE

```
PortPrint(1, IntToText(Value, 10, 4))
```

ISBATCHNAMEVALID(*NAME*)

ARGUMENT	TYPE	DESCRIPTION
`name`	`cstring`	The batch name to be tested.

DESCRIPTION

Returns *true* if the specified batch name contains valid characters, and does not already exist.

FUNCTION TYPE

This function is active.

RETURN TYPE

int

ISBATTERYLOW()

ARGUMENT	TYPE	DESCRIPTION
none		

DESCRIPTION

Returns *true* if the unit's internal battery is low.

FUNCTION TYPE

This function is passive.

RETURN TYPE

int

ISDEVICEONLINE(*DEVICE*)

ARGUMENT	TYPE	DESCRIPTION
device	**int**	Reports if device is online.

DESCRIPTION

Reports if device is online or not. As device is marked as offline if a repeated sequence of communications error have occurred. When a device is in the offline state, it will be polled periodically to see if has returned online.

FUNCTION TYPE

This function is passive.

RETURN TYPE

```
int
```

EXAMPLE

```
Okay := IsDeviceOnline(1)
```

ISLOGGINGACTIVE()

ARGUMENT	TYPE	DESCRIPTION
none		

DESCRIPTION

Returns true or false indicating whether data logging is active in the current database. A value of true indicates that a log has been defined, and that the log contains at least one data tag.

FUNCTION TYPE

This function is passive.

RETURN TYPE

```
int
```

ISPORTREMOTE(*PORT*)

ARGUMENT	TYPE	DESCRIPTION
`port`	`int`	The communications port to be queried.

DESCRIPTION

Returns *true* if the specified port has been taken over via port sharing.

FUNCTION TYPE

This function is passive.

RETURN TYPE

int

ISWRITEQUEUEEMPTY(*DEV*)

ARGUMENT	TYPE	DESCRIPTION
dev	**int**	The device number to get the queue state from

DESCRIPTION

Returns the state of the writes queue for the device identified with the argument dev. The function will return true if the queue is empty, false otherwise. The device number can be identified in Crimson's status bar when a device is selected in Communication.

FUNCTION TYPE

This function is passive.

RETURN TYPE

```
int
```

EXAMPLE

```
QueueEmpty = IsWriteQueueEmpty(1)
```

In this example, the function returns the write queue stat for device 1.

KILLDIRECTORY(*NAME*)

ARGUMENT	TYPE	DESCRIPTION
`name`	`cstring`	The directory to be deleted.

DESCRIPTION

Deletes the specified directory and any subdirectories or files that it contains.

FUNCTION TYPE

This function is active.

RETURN TYPE

int

LEFT(*STRING*, *COUNT*)

ARGUMENT	TYPE	DESCRIPTION
string	**cstring**	The string to be processed.
count	**int**	The number of characters to return.

DESCRIPTION

Returns the first *count* characters from *string*.

FUNCTION TYPE

This function is passive.

RETURN TYPE

cstring

EXAMPLE

```
AreaCode := Left(Phone, 3)
```

LEN(*STRING*)

ARGUMENT	TYPE	DESCRIPTION
string	**cstring**	The string to be processed.

DESCRIPTION

Returns the number of characters in *string*.

FUNCTION TYPE

This function is passive.

RETURN TYPE

int

EXAMPLE

```
Size := Len(Input)
```

LESSR64(A, B)

ARGUMENT	TYPE	DESCRIPTION
a	**int**	The value to be compared.
b	**int**	The value to compare to.

DESCRIPTION

Compares the value of *a* to *b* using 64-bit (double precision) floating point math and returns 1 if *a* is less than *b* and 0 otherwise. This is the double precision equivalent of a < b.

The input operands *a* and *b* should be obtained from one of the 64-bit conversion functions provided or from a driver that can read double precision values. All arguments to this function must be integer arrays with lengths of 2. An in depth example is provided in the entry for AddR64.

FUNCTION TYPE

This function is passive.

RETURN TYPE

```
int
```

EXAMPLE

```
LessR64(a[0], b[0])
```

LOADCAMERASETUP(*PORT, CAMERA, INDEX, FILE*)

ARGUMENT	TYPE	DESCRIPTION
port	**int**	The port number where the camera is connected
camera	**int**	The camera device number
index	**int**	The inspection file number in the camera
file	**cstring**	The path and filename for the inspection file on the operator interface CompactFlash card

DESCRIPTION

This function loads the inspection file from the operator interface CompactFlash card to the camera memory. The number to be placed in the *port* argument is the port number to which the driver is bound. The argument *camera* is the device number showing in Crimson 2.0 status bar when the camera is selected. More than one camera can be connected under a single driver. The *index* represents the inspection file number within the camera where the file will be loaded in. The *file* is the path and filename for the source inspection file on the CompactFlash card. This function will return true if the transfer is successful, false otherwise. Note that this function is best called in a user program that runs in the background so the G3 has enough time to access the CompactFlash card.

FUNCTION TYPE

This function is active.

RETURN TYPE

```
int
```

EXAMPLE

```
Success = LoadCameraSetup(4, 0, 1, "\\in0.isp")
```

Loads the file named "in0.isp" in inspection file number 1 in camera device number 0 connected on port 4.

LOADSECURITYDATABASE(*MODE, FILE*)

ARGUMENT	TYPE	DESCRIPTION
`mode`	`int`	The file format to be used.
`file`	`cstring`	The file to hold the database.

DESCRIPTION

Loads the database's security database from the specified file. A *mode* value of 0 is used to save and subsequently load only the password associated with each user. A *mode* value of 1 is used to save and load the entire user list, complete with user names, real names and passwords. In each case, the file is encrypted and will not contain clear-text passwords.

The return value is *true* for success, and *false* for failure.

FUNCTION TYPE

This function is active.

RETURN TYPE

int

LOG(*VALUE*)

ARGUMENT	TYPE	DESCRIPTION
value	**float**	The value to be processed.

DESCRIPTION

Returns the natural log of *value*.

FUNCTION TYPE

This function is passive.

RETURN TYPE

```
float
```

EXAMPLE

```
Variable1 := log(5.0)
```

LOG10(*VALUE*)

ARGUMENT	TYPE	DESCRIPTION
value	**float**	The value to be processed.

DESCRIPTION

Returns the base-10 log of *value*.

FUNCTION TYPE

This function is passive.

RETURN TYPE

```
float
```

EXAMPLE

```
Variable3 := log10(5.0)
```

LOGBATCHCOMMENT(*SET, TEXT*)

ARGUMENT	TYPE	DESCRIPTION
`set`	`int`	The batch set number.
`text`	`cstring`	The comment to be logged.

DESCRIPTION

Logs a comment to all batches associated with the specified batch set.

FUNCTION TYPE

This function is active.

RETURN TYPE

This function does not return a value.

LOGBATCHHEADER(*SET, TEXT*)

ARGUMENT	TYPE	DESCRIPTION
`set`	`int`	The batch set number.
`text`	`cstring`	The header to be logged.

DESCRIPTION

Logs a header comment to all batches associated with the specified batch set. The call should be made immediately after creating a new batch. The comments will always be placed ahead of any other data in the file.

FUNCTION TYPE

This function is active.

RETURN TYPE

This function does not return a value.

LOGCOMMENT(LOG, TEXT)

ARGUMENT	TYPE	DESCRIPTION
log	**int**	The index of the log to be accessed.
text	**cstring**	The textual comment to be added to the log.

DESCRIPTION

Adds a comment to a data log. The data log must be configured to support comments via the appropriate property. Comments can be used to provide batch or other details at the start of a log, or to allow the operator to mark a point of interest during the logging process.

FUNCTION TYPE

This function is active.

RETURN TYPE

```
int
```

EXAMPLE

```
LogComment(1, "Start of Shift")
```

LOGHEADER(*LOG, TEXT*)

ARGUMENT	TYPE	DESCRIPTION
`log`	`int`	The index or name of the log.
`text`	`cstring`	The comment to be logged.

DESCRIPTION

Records a comment in the specified log file.

Comments must be enabled for the log in question.

FUNCTION TYPE

This function is active.

RETURN TYPE

This function does not return a value.

LOGR64(RESULT, TAG)

ARGUMENT	TYPE	DESCRIPTION
result	int	The result.
tag	int	The value to be processed.

DESCRIPTION

Calculates the natural logarithm of *tag* using 64-bit (double precision) floating point math and stores the result in *result*.

The input operand *tag* should be obtained from one of the 64-bit conversion functions provided or from a driver that can read double precision values. All arguments to this function must be integer arrays with lengths of 2. An in depth example is provided in the entry for AddR64.

FUNCTION TYPE

This function is active.

RETURN TYPE

```
void
```

EXAMPLE

```
logR64(result[0], tag[0])
```

LOGSAVE()

ARGUMENT	TYPE	DESCRIPTION
none		

DESCRIPTION

Forces the data logger to save on the CompactFlash Card.

Note: This function should NOT be called permanently or regularly. It is intended only for punctual use. An overuse of this function may result in CompactFlash card damage and loss of data.

FUNCTION TYPE

This function is passive.

RETURN TYPE

This function does not return a value

EXAMPLE

```
LogSave()
```

MAKEFLOAT(*VALUE*)

ARGUMENT	TYPE	DESCRIPTION
value	**int**	The value to be converted.

DESCRIPTION

Reinterprets the integer argument as a floating-point value. This function does not perform a type conversion, but instead takes the bit pattern stored in the argument, and assumes that rather than representing an integer, it actually represents a floating-point value. It can be used to manipulate data from a remote device that might actually have a different data type from that expected by the communications driver.

FUNCTION TYPE

This function is passive.

RETURN TYPE

```
float
```

EXAMPLE

```
fp := MakeFloat(n)
```

MAKEINT(*VALUE*)

ARGUMENT	TYPE	DESCRIPTION
value	**float**	The value to be converted.

DESCRIPTION

Reinterprets the floating-point argument as an integer. This function does not perform a type conversion, but instead takes the bit pattern stored in the argument, and assumes that rather than representing a floating-point value, it actually represents an integer. It can be used to manipulate data from a remote device that might actually have a different data type from that expected by the communications driver.

FUNCTION TYPE

This function is passive.

RETURN TYPE

```
int
```

EXAMPLE

```
n := MakeInt(fp)
```

MAX(*A*, *B*)

ARGUMENT	TYPE	DESCRIPTION
a	**int / float**	The first value to be compared.
b	**int / float**	The second value to be compared.

DESCRIPTION

Returns the larger of the two arguments.

FUNCTION TYPE

This function is passive.

RETURN TYPE

`int` or `float`, depending on the type of the arguments.

EXAMPLE

```
Larger := Max(Tank1, Tank2)
```

MAXR64(RESULT, TAG1, TAG2)

ARGUMENT	TYPE	DESCRIPTION
result	**int**	The result.
tag1	**int**	The first value to be compared.
tag2	**int**	The second value to be compared.

DESCRIPTION

Calculates the larger value of *tag1* and *tag2* using 64-bit (double precision) floating point math and stores the result in *result*.

The input operands *tag1* and *tag2* should be obtained from one of the 64-bit conversion functions provided or from a driver that can read double precision values. All arguments to this function must be integer arrays with lengths of 2. An in depth example is provided in the entry for AddR64.

FUNCTION TYPE

This function is active

RETURN TYPE

```
void
```

EXAMPLE

```
MaxR64(result[0], tag1[0], tag2[0])
```

MAXU32(TAG1, TAG2)

ARGUMENT	TYPE	DESCRIPTION
tag1	**int**	The first value to be compared.
tag2	**int**	The second value to be compared.

DESCRIPTION

Returns the larger value of *tag1* and *tag2* in an unsigned context.

FUNCTION TYPE

This function is passive.

RETURN TYPE

```
int
```

EXAMPLE

```
Larger := MaxU32(tag1, tag2)
```

MEAN(*ELEMENT*, *COUNT*)

ARGUMENT	TYPE	DESCRIPTION
element	**int** / **float**	The first array element to be processed.
count	**int**	The number of elements to be processed.

DESCRIPTION

Returns the mean of the *count* array elements from *element* onwards.

FUNCTION TYPE

This function is passive.

RETURN TYPE

```
float
```

EXAMPLE

```
Average := Mean(Data[0], 10)
```

MID(*STRING*, *POS*, *COUNT*)

ARGUMENT	TYPE	DESCRIPTION
string	**cstring**	The string to be processed.
pos	**int**	The position at which to start.
count	**int**	The number of characters to return.

DESCRIPTION

Returns *count* characters from position *pos* within *string*, where 0 is the first position.

FUNCTION TYPE

This function is passive.

RETURN TYPE

```
cstring
```

EXAMPLE

```
Exchange := Mid(Phone, 3, 3)
```

MIN(*A*, *B*)

ARGUMENT	TYPE	DESCRIPTION
a	**int / float**	The first value to be compared.
b	**int / float**	The second value to be compared.

DESCRIPTION

Returns the smaller of the two arguments.

FUNCTION TYPE

This function is passive.

RETURN TYPE

`int` or `float`, depending on the type of the arguments.

EXAMPLE

```
Smaller := Min(Tank1, Tank2)
```

MINR64(RESULT, TAG1, TAG2)

ARGUMENT	TYPE	DESCRIPTION
result	**int**	The result.
tag1	**int**	The first value to compare.
tag2	**int**	The second value to compare.

DESCRIPTION

Calculates the smaller value of *tag1* and *tag2* using 64-bit (double precision) floating point math and stores the result in *result*.

The input operands *tag1* and *tag2* should be obtained from one of the 64-bit conversion functions provided or from a driver that can read double precision values. All arguments to this function must be integer arrays with lengths of 2. An in depth example is provided in the entry for AddR64.

FUNCTION TYPE

This function is active.

RETURN TYPE

```
void
```

EXAMPLE

```
MinR64(result[0], tag1[0], tag2[0])
```

MINU32(TAG1, TAG2)

ARGUMENT	TYPE	DESCRIPTION
tag1	**int**	The first value to be compared.
tag2	**int**	The second value to be compared.

DESCRIPTION

Returns the smaller value of *tag1* and *tag2* in an unsigned context.

FUNCTION TYPE

This function is (active/passive?).

RETURN TYPE

```
int
```

EXAMPLE

```
Smaller := MinU32(tag1, tag2)
```

MINUSR64(RESULT, TAG)

ARGUMENT	TYPE	DESCRIPTION
result	**int**	The result.
tag	**int**	The value to be processed.

DESCRIPTION

Inverts the sign of *tag* using 64-bit (double precision) floating point math and stores the result in *result*. This function will cause positive numbers to become negative and negative numbers to become positive.

The input operand *tag* should be obtained from either one of the 64-bit conversion functions provided or from a driver that can read double precision values. All arguments to this function must be integer arrays with lengths of 2. An in depth example is provided in the entry for AddR64.

FUNCTION TYPE

This function is active.

RETURN TYPE

```
void
```

EXAMPLE

```
MinusR64(result[0], tag[0])
```

MODU32(TAG1, TAG2)

ARGUMENT	TYPE	DESCRIPTION
tag1	**int**	The dividend.
tag2	**int**	The divisor.

DESCRIPTION

Returns the value of *tag1* modulo *tag2* in an unsigned context. This is the unsigned equivalent of *tag1* % *tag2*, or the remainder of tag1 divided by tag2.

FUNCTION TYPE

This function is passive.

RETURN TYPE

```
int
```

EXAMPLE

```
Result := ModU32(tag1, tag2)
```

MOVEFILES(*SOURCE, TARGET, FLAGS*)

ARGUMENT	TYPE	DESCRIPTION
`source`	`cstring`	The path from which the files are to be moved.
`target`	`cstring`	The path to which the files are to be moved.
`flags`	`int`	The flags controlling the move operation.

DESCRIPTION

Moves all the files in the *source* directory to the *target* directory.

The various bits in `flags` modify the move operation…

BIT	WEIGHT	DESCRIPTION
0	1	If set, the operation will recurse into any subdirectories.
1	2	If set, existing files will be overwritten. If clear, existing files will be left untouched.

The return value of the function will be *true* for success, or *false* for failure.

FUNCTION TYPE

This function is active.

RETURN TYPE

int

MU1U32(TAG1, TAG2)

ARGUMENT	TYPE	DESCRIPTION
tag1	**int**	The multiplicand tag.
tag2	**int**	The multiplier tag.

DESCRIPTION

Returns the value of *tag1* times *tag2* in an unsigned context.

FUNCTION TYPE

This function is passive.

RETURN TYPE

```
int
```

EXAMPLE

```
Result := MulU32(tag1, tag2)
```

MULDIV(*A, B, C*)

ARGUMENT	TYPE	DESCRIPTION
a	**int**	The first value.
b	**int**	The second value.
c	**int**	The third value.

DESCRIPTION

Returns `a*b/c`. The intermediate math is done with 64-bit integers to avoid overflows.

FUNCTION TYPE

This function is passive.

RETURN TYPE

`int`

EXAMPLE

```
d := MulDiv(a, b, c)
```

MULR64(RESULT, TAG1, TAG2)

ARGUMENT	TYPE	DESCRIPTION
result	**int**	The result.
tag1	**int**	The multiplicand.
tag2	**int**	The multiplier.

DESCRIPTION

Calculates the value of *tag1* times *tag2* using 64-bit (double precision) floating point math and stores the result in *result.* This is the double precision equivalent of *tag1 * tag2.*

The input operands *tag1* and *tag2* should be obtained from one of the 64-bit conversion functions provided or from a driver that can read double precision values. All arguments to this function must be integer arrays with lengths of 2. An in depth example is provided in the entry for AddR64.

FUNCTION TYPE

This function is active.

RETURN TYPE

```
void
```

EXAMPLE

```
MulR64(result[0], tag1[0], tag2[0])
```

MUTESIREN()

ARGUMENT	TYPE	DESCRIPTION
none		

DESCRIPTION

Turns off the operator panel's internal siren.

FUNCTION TYPE

This function is active.

RETURN TYPE

This function does not return a value.

EXAMPLE

```
MuteSiren()
```

NEWBATCH(*NAME*)

ARGUMENT	TYPE	DESCRIPTION
name	**cstring**	The name of the batch.

DESCRIPTION

Starts a batch called *name*. The name must be no more than 8 characters in length and made up of characters that are valid FAT16 filename. Restarting a batch already on the CF card will append the data. If a new batch exceeds the maximum number of batches to be kept, the oldest batch (i.e. The one last changed) will be deleted. If name is empty, the function is equivalent to `EndBatch()`.

Note: Batch status is retained during a power cycle. Starting a new batch within less than 10 seconds of ending or starting the last one will produce undefined behavior. To go straight from one batch to another, call `NewBatch()` without an intervening call to `EndBatch()`.

FUNCTION TYPE

This function is passive.

RETURN TYPE

This function does not return a value

EXAMPLE

```
NewBatch("ProdA")
```

NOP()

ARGUMENT	TYPE	DESCRIPTION
none		

DESCRIPTION

This function does nothing.

FUNCTION TYPE

This function is active.

RETURN TYPE

This function does not return a value.

EXAMPLE

```
Nop()
```

NOTEQUALR64(A, B)

ARGUMENT	TYPE	DESCRIPTION
a	int	The first value to be compared.
b	int	The second value to be compared.

DESCRIPTION

Compares the value of *a* to *b* using 64-bit (double precision) floating point math and returns 1 if *a* is not equal to *b* and 0 otherwise. This is the double precision equivalent of a != b. Note that comparing floating point values for equality can be error prone due to rounding errors.

The input operands *a* and *b* should be obtained from one of the 64-bit conversion functions provided or from a driver that can read double precision values. All arguments to this function must be integer arrays with lengths of 2. An in depth example is provided in the entry for AddR64.

FUNCTION TYPE

This function is passive.

RETURN TYPE

```
int
```

EXAMPLE

```
NotEqualR64(a[0], b[0])
```

OPENFILE(*NAME, MODE*)

ARGUMENT	TYPE	DESCRIPTION
name	**cstring**	The file to be opened.
mode	**int**	The mode in which the file is to be opened… 0 = Read Only 1 = Read/Write at Start of File 2 = Read/Write at End of File

DESCRIPTION

Returns a handle to the file *name* located on the CompactFlash card. This function is restricted to a maximum of four open files at any given time. The CompactFlash card cannot be unmounted while a file is open. Note that the filing system used on the card does not support long filenames, and that if backslashes are included in the pathname to separate path elements, they must be doubled-up per Crimson's rules for string constants as described in the chapter on Writing Expressions. To avoid this complication, forward slashes can be used in place of backslashes without the need for such doubling. Note also that this function will not create a file that does not exist. To do this, call `CreateFile()` before calling this function.

FUNCTION TYPE

This function is active.

RETURN TYPE

```
int
```

EXAMPLE

```
hFile := OpenFile("/LOGS/LOG1/01010101.csv", 0)
```

PI()

ARGUMENT	TYPE	DESCRIPTION
none		

DESCRIPTION

Returns *pi* as a floating-point number.

FUNCTION TYPE

This function is passive.

RETURN TYPE

```
float
```

EXAMPLE

```
Scale = Pi()/180
```

PLAYRTTTL(*TUNE*)

ARGUMENT	TYPE	DESCRIPTION
tune	**cstring**	The tune to be played in RTTTL representation.

DESCRIPTION

Plays a tune using the terminal's internal beeper. The *tune* argument should contain the tune to be played in RTTTL format—the format used by a number of cell phones for custom ring tones. Sample tunes can be obtained from many sites on the World Wide Web.

FUNCTION TYPE

This function is active.

RETURN TYPE

This function does not return a value.

EXAMPLE

```
PlayRTTTL("TooSexy:d=4,o=5,b=40:16f,16g,16f,16g,16f.,16f,16g,16f,16g,16g#
.,16g#,16g,16g#,16g,16f.,16f,16g,16f,16g,16f.,16f,16g,16f,16g,16f.,16f,16
g,16f,16g,16g#.,16g#,16g,16g#,16g,16f.,16f,16g,16f,16g,32f.")
```

POPDEV(*ELEMENT*, *COUNT*)

ARGUMENT	TYPE	DESCRIPTION
element	**int** / **float**	The first array element to be processed.
count	**int**	The number of elements to be processed.

DESCRIPTION

Returns the standard deviation of the *count* array elements from *element* onwards, assuming the data points to represent the whole of the population under study. If you need to find the standard deviation of a sample, use the StdDev function instead.

FUNCTION TYPE

This function is passive.

RETURN TYPE

```
float
```

EXAMPLE

```
Dev := PopDev(Data[0], 10)
```

PORTCLOSE(*PORT*)

ARGUMENT	TYPE	DESCRIPTION
`port`	`int`	Closes the specified port.

DESCRIPTION

This function is used in conjunction with the active or passive TCP raw port drivers to close the selected port by gracefully closing the connection that is attached to the associated socket.

FUNCTION TYPE

This function is active.

RETURN TYPE

This function does not return a value.

EXAMPLE

```
PortClose(6)
```

PORTGETCTS(*PORT*)

ARGUMENT	TYPE	DESCRIPTION
port	int	The raw port to get the CTS state from

DESCRIPTION

Returns the CTS.state of the port indicated by port. The port must be configured to use a raw driver and be one of the serial ports.

Note: The communication port number can be identified in Crimson's status bar when the port is selected.

FUNCTION TYPE

This function is active.

RETURN TYPE

```
int
```

EXAMPLE

```
CtsState = PortgetCTS(2)
```

In this example, the function returns the CTS state of the RS232 communication port in the variable `CtsState`.

PORTINPUT(*PORT*, *START*, *END*, *TIMEOUT*, *LENGTH*)

ARGUMENT	TYPE	DESCRIPTION
port	**int**	The raw port to be read.
start	**int**	The start character to match, if any.
end	**int**	The end character to match, if any.
timeout	**int**	The inter-character timeout in milliseconds, if any.
length	**int**	The maximum number of characters to read, if any.

DESCRIPTION

Reads a string of characters from the *port* indicated by port, using the various other parameters to control the input process. If *start* is non-zero, the process begins by waiting until the character indicated by this parameter is received. If *start* is zero, the receive process begins immediately. The process then continues until one of the following conditions has been met…

- *end* is non-zero and a character matching *end* is received.
- *timeout* is non-zero, and that period passes without a character being received.
- *length* is non-zero, and that many characters have been received.

The function then returns the characters received, not including the *start* or *end* byte. This function is used together with Raw Port drivers to implement custom protocols using Crimson's programming language. It replaces the RYOP functionality found in Edict.

FUNCTION TYPE

This function is active.

RETURN TYPE

```
cstring
```

EXAMPLE

```
Frame := PortInput(1, '*', 13, 100, 200)
```

PORTPRINT(*PORT, STRING*)

ARGUMENT	TYPE	DESCRIPTION
port	**int**	The raw port to be written to.
string	**cstring**	The text string to be transmitted.

DESCRIPTION

Transmits the text contained in *string* to the port indicated by *port*. The port must be configured to use a raw driver, such as the raw serial port driver, or either of the raw TCP/IP drivers. The data will be transmitted, and the function will return. The port driver will handle handshaking and control of transmitter enable lines as required.

FUNCTION TYPE

This function is active.

RETURN TYPE

This function does not return a value.

EXAMPLE

```
PortPrint(1, "ABCD")
```

PORTREAD(*PORT*, *PERIOD*)

ARGUMENT	TYPE	DESCRIPTION
port	**int**	The raw port to be read.
period	**int**	The time to wait in milliseconds.

DESCRIPTION

Attempts to read a character from the port indicated by *port*. The port must be configured to use a raw driver, such as the raw serial port driver, or either of the raw TCP/IP drivers. If no data is available within the indicated time period, a value of –1 will be returned. Setting *period* to zero will result in any queued data being returned, but will prevent Crimson from waiting for data to arrive if none is available.

FUNCTION TYPE

This function is active.

RETURN TYPE

```
int
```

EXAMPLE

```
Data := PortRead(1, 100)
```

PORTSETRTS(*PORT, STATE*)

ARGUMENT	TYPE	DESCRIPTION
port	**int**	The raw port to control
state	**int**	The state of the RTS, true (1) or false (0)

DESCRIPTION

Sets the RTS.of the port indicated by port with the setting in state. The port must be configured to use a raw driver and be on of the serial ports. The state argument can take values 0 or 1 only.

Note: The communication port number can be identified in Crimson's status bar when the port is selected.

FUNCTION TYPE

This function is active.

RETURN TYPE

This function does not return a value.

EXAMPLE

```
PortSetRTS(2, 1)
```

In this example, the function sets the RTS of the RS232 communication port to true.

PORTWRITE(*PORT, DATA*)

ARGUMENT	TYPE	DESCRIPTION
port	**int**	The raw port to be written to.
data	**int**	The byte to be transmitted.

DESCRIPTION

Transmits the byte indicated by *data* on the port indicated by *port*. The port must be configured to use a raw driver, such as the raw serial port driver, or either of the raw TCP/IP drivers. The character will be transmitted, and the function will return. The port driver will handle handshaking and control of transmitter enable lines as required.

FUNCTION TYPE

This function is active.

RETURN TYPE

This function does not return a value.

EXAMPLE

```
PortWrite(1, 'A')
```

POSTKEY(*CODE, TRANSITION*)

ARGUMENT	TYPE	DESCRIPTION
code	int	Key code.
transition	int	Transition code.

DESCRIPTION

Adds a physical key operation to the input queue.

CODE	KEY
0x80	Soft Key 1
0x81	Soft Key 2
0x82	Soft Key 3
0x83	Soft Key 4
0x84	Soft Key 5
0x85	Soft Key 6
0x86	Soft Key 7
0x90	Function Key 1
0x91	Function Key 2
0x92	Function Key 3
0x93	Function Key 4
0x94	Function Key 5

CODE	KEY
0x95	Function Key 6
0x96	Function Key 7
0x97	Function Key 8
0xA0	ALARMS
0xA1	MUTE
0x1B	EXIT
0xA2	MENU
0xA3	RAISE
0xA4	LOWER
0x09	NEXT
0x08	PREV
0x0D	ENTER

TRANSITION	OPERATION
0	Post key down and then key up.
1	Post key down only.
2	Post key up only.
3	Post key repeat only.

FUNCTION TYPE

This function is active.

RETURN TYPE

```
void
```

EXAMPLE

```
PostKey(0x80, 0)
```

POWER(*VALUE*, *POWER*)

ARGUMENT	TYPE	DESCRIPTION
value	**int** / **float**	The value to be processed.
power	**int** / **float**	The power to which *value* is to be raised.

DESCRIPTION

Returns *value* raised to the *power*-th power.

FUNCTION TYPE

This function is passive.

RETURN TYPE

`int` or `float`, depending on the type of the *value* argument.

EXAMPLE

```
Volume := Power(Length, 3)
```

POWR64(RESULT, VALUE, POWER)

ARGUMENT	TYPE	DESCRIPTION
result	**int**	The result.
value	**int**	The value to be processed.
power	**int**	The power to which *value* will be raised.

DESCRIPTION

Calculates the value of *value* raised to *power*-th power using 64-bit (double precision) floating point math and stores the result in *result.*

The input operands *value* and *power* should be obtained from one of the 64-bit conversion functions provided or from a driver that can read double precision values. All arguments to this function must be integer arrays with lengths of 2. An in depth example is provided in the entry for AddR64.

FUNCTION TYPE

This function is active.

RETURN TYPE

```
void
```

EXAMPLE

```
PowR64(result[0], tag1[0], tag2[0])
```

PRINTSCREENTOFILE(PATH, NAME, RES)

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.

DESCRIPTION

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 zero to create an 8 bits-per-pixel bitmap, while a value of one will create a 16 bits-per-pixel bitmap. The latter value will produce much large files, all the more so before these files are not capable of supporting RLE8 compression. The return value indicates whether the function succeeded.

FUNCTION TYPE

This function is active.

RETURN TYPE

```
int
```

PUTFILEBYTE(*FILE, DATA*)

ARGUMENT	TYPE	DESCRIPTION
file	**int**	The file handle as returned by `OpenFile`.
data	**int**	The data value to be written.

DESCRIPTION

Writes a single byte to the specified file. Returns 1 for success and -1 for failure.

FUNCTION TYPE

This function is active.

RETURN TYPE

int

PUTFILEDATA(*FILE, DATA, LENGTH*)

ARGUMENT	TYPE	DESCRIPTION
file	**int**	The file handle as returned by `OpenFile`.
data	**int**	The first array element to be written.
length	**int**	The number of elements to be processed.

DESCRIPTION

Write the specific number of bytes to the file, taking one byte from each array element.

The return value is the number of bytes written, and may be less than *length*.

FUNCTION TYPE

This function is active.

RETURN TYPE

int

R64TOINT(X)

ARGUMENT	TYPE	DESCRIPTION
x	**int**	The value to be converted.

DESCRIPTION

Converts the 64-bit (double precision) floating point value stored in *x* as an array with extent 2 to a signed integer and returns the result. Typically, the array *x* will contain a 64-bit floating point value obtained as a result from one of the 64-bit math functions provided. Note that if the number represented by the array *x* must be able to be represented by 32-bit integer for this conversion to be successful. See the entry for AddR64 for an example of the use of 64-bit math functions.

FUNCTION TYPE

This function is passive.

RETURN TYPE

```
int
```

EXAMPLE

```
Result := R64ToInt(x[0])
```

R64TOREAL(X)

ARGUMENT	TYPE	DESCRIPTION
x	**int**	The value to be converted.

DESCRIPTION

Converts the 64-bit (double precision) floating point value stored as an array with extent 2 in *x* to a 32-bit floating point number (a *float* in Crimson 3) and returns the result. Typically, the array *x* will contain a 64-bit floating point value obtained as a result from one of the 64-bit math functions provided. Note that if the number represented by the array *x* must be able to be represented a 32-bit floating point number for this conversion to be successful. See the entry for AddR64 for an example of the use of 64-bit math functions.

FUNCTION TYPE

This function is passive.

RETURN TYPE

```
float
```

EXAMPLE

```
Result := R64ToReal(x[0])
```

RAD2DEG(*THETA*)

ARGUMENT	TYPE	DESCRIPTION
theta	**float**	The angle to be processed.

DESCRIPTION

Returns *theta* converted from radians to degrees.

FUNCTION TYPE

This function is passive.

RETURN TYPE

```
float
```

EXAMPLE

```
Right := Rad2Deg(Pi()/2)
```

RANDOM(*RANGE*)

ARGUMENT	TYPE	DESCRIPTION
range	**int**	The range of random values to produce.

DESCRIPTION

Returns a pseudo-random value between 0 and *range*-1.

FUNCTION TYPE

This function is passive.

RETURN TYPE

```
int
```

EXAMPLE

```
Noise := Random(100)
```

READDATA(*DATA, COUNT*)

ARGUMENT	TYPE	DESCRIPTION
data	**any**	The first array element to be read.
count	**int**	The number of elements to be read.

DESCRIPTION

Requests that *count* elements from array element *data* onwards to read on the next comms scan. This function is used with arrays that have been mapped to external data, and which have their read policy set to *Read Manually*. The function returns immediately, and does not wait for the data to be read.

FUNCTION TYPE

This function is active.

RETURN TYPE

This function does not return a value.

EXAMPLE

```
ReadData(array1[8], 10)
```

READFILE(*FILE, CHARS*)

ARGUMENT	TYPE	DESCRIPTION
file	**int**	The file handle as required by OpenFile.
chars	**int**	The number of characters to be read.

DESCRIPTION

Reads a string up to 512 characters in length from the specified file. This function does not look for a line feed and carriage return therefore allowing line read of more than 510 characters (`ReadFileLine()` limit).

If a file as multiple lines, the string returned by `ReadFile()` will be as many lines as required to reach the number of characters to be read. Line feed and carriage return will be part of the returned string.

FUNCTION TYPE

This function is active.

RETURN TYPE

`string`

EXAMPLE

```
Text := ReadFile(hFile, 80)
```

READFILELINE(*FILE*)

ARGUMENT	TYPE	DESCRIPTION
file	**int**	The file handle as returned by **OpenFile**.

DESCRIPTION

Returns a single line of text from file.

FUNCTION TYPE

This function is active.

RETURN TYPE

```
cstring
```

EXAMPLE

```
Text := ReadFileLine(hFile)
```

REALTOR64(RESULT, N)

ARGUMENT	TYPE	DESCRIPTION
result	**int**	The result.
n	**float**	The value to be converted.

DESCRIPTION

Converts the value stored in *n* from a real number (a *float* in Crimson 3) to a 64-bit (double precision) number and stores the result as an array of length 2 in *result.* The tag *result* should therefore be an integer array with an extent of at least 2. After execution of this function, the value stored in *result* is suitable for use in other 64-bit math functions. See the entry for AddR64 for an example of the intended use of this function.

FUNCTION TYPE

This function is active.

RETURN TYPE

```
void
```

EXAMPLE

```
RealToR64(result[0], n)
```

RENAMEFILE(*HANDLE, NAME*)

ARGUMENT	TYPE	DESCRIPTION
handle	**int**	The file handle.
name	**cstring**	The new file name.

DESCRIPTION

Returns a non-zero value upon a successful rename file operation. The file handle is the returned value of the `Openfile()` function. After the rename operation, the file stays open and should be closed if no further operations are required. The file name is maximum 8 characters long, excluding the extension, which is 3 characters long maximum.

FUNCTION TYPE

This function is active.

RETURN TYPE

```
int
```

EXAMPLE

```
Result := RenameFile(File , "NewName.txt")
```

RESOLVEDNS(*NAME*)

ARGUMENT	TYPE	DESCRIPTION
name	**cstring**	The DNS name to be resolved.

DESCRIPTION

Returns the IP address of the specified DNS name.

FUNCTION TYPE

This function is passive.

RETURN TYPE

int

EXAMPLE

```
ip = ResolveDNS("www.redlion.net")
```

RIGHT(*STRING*, *COUNT*)

ARGUMENT	TYPE	DESCRIPTION
string	**cstring**	The string to be processed.
count	**int**	The number of characters to return.

DESCRIPTION

Returns the last *count* characters from *string*.

FUNCTION TYPE

This function is passive.

RETURN TYPE

```
cstring
```

EXAMPLE

```
Local := Right(Phone, 7)
```

RSHU32(TAG1, TAG2)

ARGUMENT	TYPE	DESCRIPTION
tag1	**int**	The value to be shifted.
tag2	**int**	The amount to shift by.

DESCRIPTION

Returns the value *tag1* shifted *tag2* bits to the right in an unsigned context. This is the unsigned equivalent to *tag1 >> tag2*.

FUNCTION TYPE

This function is passive.

RETURN TYPE

```
int
```

EXAMPLE

```
Shifted := RShU32(tag1, tag2)
```

RXCAN(PORT, DATA, ID)

ARGUMENT	TYPE	DESCRIPTION
port	**int**	The raw 29-bit CAN port
data	**Int**	The first array element to hold received data.
ID	**int**	29-bit CAN Identifier

DESCRIPTION

Retrieves received CAN messages that have been initialized with RxCANInit(). The first four bytes of the received message will be packed (big endian) in the indicated array element while remaining bytes (if any) will be stored (big endian) in the next consecutive element of the array. Returns a value of 1 upon success.

FUNCTION TYPE

This function is active.

RETURN TYPE

```
int
```

EXAMPLE

```
RxCAN(8, Data, 0x12345678)
```

RXCANINIT(PORT, ID, DLC)

ARGUMENT	TYPE	DESCRIPTION
port	**int**	The raw 29-bit CAN port
ID	**int**	29-bit CAN Identifier
DLC	**int**	Data Length Count in bytes (1 – 8 bytes are supported)

DESCRIPTION

Initializes CAN messages (Red Lion CAN Option Card required). Initialization function returns a value of 1 upon success or a value of 0 indicating failure. Calls of RxCANInit() should be made only after the system has started and each 29-bit identifier should only be initialized one time via RxCANInit().

FUNCTION TYPE

This function is active.

RETURN TYPE

```
int
```

EXAMPLE

```
RxCANInit(8, 0x12345678, 8)
```

SAVECAMERASETUP(*PORT, CAMERA, INDEX, FILE*)

ARGUMENT	TYPE	DESCRIPTION
port	**int**	The port number where the camera is connected
camera	**int**	The camera device number
index	**int**	The inspection file number in the camera
file	**cstring**	The path and filename for the inspection file on the operator interface CompactFlash card

DESCRIPTION

This function saves the inspection file uploaded from the camera on the operator interface CompactFlash card. The number to be placed in the *port* argument is the port number to which the driver is bound. The argument *camera* is the device number showing in Crimson 2.0 status bar when the camera is selected. More than one camera can be connected under a single driver. The *index* represents the inspection file number within the camera. The *file* is the path and filename where the inspection file should be saved on CompactFlash card. This function will return true if the transfer is successful, false otherwise.

*Note: This function should be called in a user program that runs in the background so the G3 has enough time to access the CompactFlash card.

FUNCTION TYPE

This function is active.

RETURN TYPE

```
int
```

EXAMPLE

```
Success = SaveCameraSetup(4, 0, 1, "\\in0.isp")
```

Saves the inspection file number 1 from camera device number 0 connected on port 4 under the name "in0.isp".

SAVECONFIGFILE(*FILE*)

ARGUMENT	TYPE	DESCRIPTION
`file`	`cstring`	The image file to which to write.

DESCRIPTION

Save the current boot loader, firmware and database image to a CDI file for subsequent transfer to another device. Note that image files created in this manner will only contain the firmware for the exact hardware model on which they were created, and may not operate with similar but non-identical devices. The return value is *true* for success, and *false* for failure.

FUNCTION TYPE

This function is active.

RETURN TYPE

int

EXAMPLE

```
SaveConfigFile("image.cdi")
```

SAVESECURITYDATABASE(*MODE, FILE*)

ARGUMENT	TYPE	DESCRIPTION
`mode`	`int`	The file format to be used.
`file`	`cstring`	The file to hold the database.

DESCRIPTION

Saves the database's security database from the specified file. A *mode* value of 0 is used to save and subsequently load only the password associated with each user. A *mode* value of 1 is used to save and load the entire user list, complete with user names, real names and passwords. In each case, the file is encrypted and will not contain clear-text passwords.

The return value is *true* for success, and *false* for failure.

RETURN TYPE

int

SCALE(*DATA, R1, R2, E1, E2*)

ARGUMENT	TYPE	DESCRIPTION
data	**int**	The value to be scaled.
r1	**int**	The minimum raw value stored in *data*..
r2	**int**	The maximum raw value stored in *data*..
e1	**int**	The engineering value corresponding to *r1*.
e2	**int**	The engineering value corresponding to *r2*.

DESCRIPTION

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*. The internal math is implemented using 64-bit integers, thereby avoiding the overflows that might result if you attempted to scale very large values using Crimson's own math operators.

FUNCTION TYPE

This function is passive.

RETURN TYPE

```
int
```

EXAMPLE

```
Data := Scale([D100], 0, 4095, 0, 99999)
```

SENDFILE(*RCPT, FILE*)

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.

DESCRIPTION

Sends an email from the operator interface with the file specified attached. The function returns immediately, having first added the required email to the system's mail queue. The message will be sent using the appropriate mail transport as configured in the database.

FUNCTION TYPE

This function is passive.

RETURN TYPE

This function does not return a value

EXAMPLE

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

SENDFILEEX(*RCPT, FILE, SUBJECT, FLAG*)

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.
subject	**cstring**	The subject of the email.
flag	**int**	Not used. Should be set to zero.

DESCRIPTION

Sends an email from the operator interface with the file specified attached, and with the specified subject line. The function returns immediately, having first added the required email to the system's mail queue. The message will be sent using the appropriate mail transport as configured in the database.

FUNCTION TYPE

This function is passive.

RETURN TYPE

This function does not return a value

EXAMPLE

```
SendFileEx(0, "/LOGS/LOG1/260706.csv", "Test Email", 0)
```

SENDMAIL(*RCPT, SUBJECT, BODY*)

ARGUMENT	TYPE	DESCRIPTION
rcpt	**int**	The recipient's index in the database's address book.
subject	**cstring**	The required subject line for the email.
body	**cstring**	The required body text of the email.

DESCRIPTION

Sends an email from the operator interface. The function returns immediately, having first added the required email to the system's mail queue. The message will be sent using the appropriate mail transport as configured in the database.

Note: The first recipient is 0.

FUNCTION TYPE

This function is active.

RETURN TYPE

This function does not return a value.

EXAMPLE

```
SendMail(1, "Test Subject Line", "Test Body Text")
```

SET(*TAG, VALUE*)

ARGUMENT	TYPE	DESCRIPTION
tag	**int/float**	The tag to be changed.
value	**int/float**	The value to be assigned.

DESCRIPTION

This function sets the specified tag to the specified value. It differs from the more normally used assignment operator in that it deletes any queued writes to this tag and replaces them with an immediate write of the specified value. It is thus used in situations where Crimson's normal write behavior is not required.

FUNCTION TYPE

This function is active.

RETURN TYPE

This function does not return a value.

EXAMPLE

```
Set(Tag1, 100)
```

SETINTTAG(*INDEX, VALUE*)

ARGUMENT	TYPE	DESCRIPTION
index	**int**	Tag index number
value	**int**	The value to be assigned

DESCRIPTION

This function sets the tag specified by *index* to the specified value. The index can be found from the tag label using the function `FindTagIndex()`.This function will only work if the target tag is an integer.

FUNCTION TYPE

This function is active.

RETURN TYPE

This function does not return a value.

EXAMPLE

```
SetIntTag(5,1234)
```

Set the tag of index 5 with value 1234.

SETLANGUAGE(*CODE*)

ARGUMENT	TYPE	DESCRIPTION
code	**int**	The language to be selected.

DESCRIPTION

Set the terminal's current language to that indicated by *code*.

FUNCTION TYPE

This function is active.

RETURN TYPE

This function does not return a value.

EXAMPLE

```
SetLanguage(1)
```

SETNOW(*TIME*)

ARGUMENT	TYPE	DESCRIPTION
time	**int**	The new time to be set.

DESCRIPTION

Sets the current time via an integer that represents the number of seconds that have elapsed since 1st January 1997. The integer is typically generated via the other time/date functions.

FUNCTION TYPE

This function is active.

RETURN TYPE

This function does not return a value.

EXAMPLE

```
SetNow(252288000)
```

SETPORTCONFIG(*PORT, PARAM, VALUE*)

ARGUMENT	TYPE	DESCRIPTION
port	**int**	The number of the port to be set
param	**int**	The port parameter to be set
value	**int**	The value of the parameter

DESCRIPTION

Sets the serial port parameter to value. The port number starts from the programming port with value 1. The table below shows the parameter number and associated possible values.

PARAM	DESCRIPTION	POSSIBLE VALUES
1	Baud Rate	The actual baud rate, e.g. 115200
2	Data Bits	7, 8 or 9
3	Stop Bits	1 or 2
4	Parity	0 (none), 1 (odd) or 2 (even)
5	Physical Mode	1 (RS232), 2 (422 Master), 3 (422 Slave), 4 (485)

This function will only work when called before the device startup. The On Load field provided in the User Interface on the pages tree root is used for this purpose. See example below for more details. The function `CommitAndReset()` is used to force the device to cycle power in order for the `SetPortConfig()` function to set the new port parameters.

FUNCTION TYPE

This function is active.

RETURN TYPE

`cstring`

SETREALTAG(*INDEX, VALUE*)

ARGUMENT	TYPE	DESCRIPTION
index	**int**	The tag index number
value	**float**	The value to be assigned

DESCRIPTION

This function sets the tag specified by *index* to the specified value. The index can be found from the tag label using the function `FindTagIndex()`.This function will only work if the target tag is a real (floating point).

FUNCTION TYPE

This function is active.

RETURN TYPE

This function does not return a value.

EXAMPLE

```
SetRealTag(5, 12.55)
```

Set the real tag of index 5 with value 12.55.

SETSTRINGTAG(*INDEX, DATA*)

ARGUMENT	TYPE	DESCRIPTION
index	**int**	Tag index number.
data	**cstring**	The value to be assigned.

DESCRIPTION

This function sets the tag specified by *index* to the specified value. The index can be found from the tag label using the function `FindTagIndex()`.This function will only work if the target tag is a string.

FUNCTION TYPE

This function is active.

RETURN TYPE

This function does not return a value.

SGN(*VALUE*)

ARGUMENT	TYPE	DESCRIPTION
value	**int / float**	The value to be processed.

DESCRIPTION

Returns –1 if *value* is less than zero, +1 if it is greater than zero, or 0 if it is equal to zero.

FUNCTION TYPE

This function is passive.

RETURN TYPE

`int` or `float`, depending on the type of the *value* argument.

EXAMPLE

```
State := Sgn(Level)+1
```

SHOWMENU(*NAME*)

ARGUMENT	TYPE	DESCRIPTION
name	Display Page	The display page to show as popup menu.

DESCRIPTION

Displays the page specified as a popup menu. This function is only available with on units fitted with touch-screens. Popup menus are shown on top of whatever is already on the screen, and are aligned with the left-hand side of the display.

FUNCTION TYPE

This function is active.

RETURN TYPE

This function does not return a value.

EXAMPLE

```
ShowMenu(Page2)
```

SHOWMODAL(*NAME*)

ARGUMENT	TYPE	DESCRIPTION
name	Display Page	The page to be displayed as a modal popup.

DESCRIPTION

Shows page *name* as a popup on the terminal's display. The popup will be centered on the display, and shown on top of the existing page and any existing popups. The popup will not be removed and the function will not return until a call is made to `EndModal()`, at which point that value passed to that function will be returned by `ShowModal()`.

Modal popups are used to implement user interface features such as yes-or-no confirmation popups from within a program. For example, you may wish to have the user confirm that a given file should indeed be deleted by your proceed with the delete operation. Modal popups make this easier, and involve the need to create complex state machines.

FUNCTION TYPE

This function is active.

RETURN TYPE

int

EXAMPLE

```
if( ShowModal(ConfirmDelete) == 1 ) {
       DeleteFile(OpenFile("file.dat", 1));
       }
```

SHOWNESTED(*NAME*)

ARGUMENT	TYPE	DESCRIPTION
name	Display Page	The page to be displayed as a popup.

DESCRIPTION

Shows page *name* as a popup on the terminal's display. The popup will be centered on the display, and shown on top of the existing page and any existing popups. The popup can be removed by calling either the `HidePopup()` or `HideAllPopups()` functions. It will also be removed from the display if a new page is selected by invoking the `GotoPage()` function, or by a suitably defined keyboard action.

FUNCTION TYPE

This function is active.

RETURN TYPE

This function does not return a value.

SHOWPOPUP(*NAME*)

ARGUMENT	TYPE	DESCRIPTION
name	**Display Page**	The page to be displayed as a popup.

DESCRIPTION

Shows page *name* as a popup on the terminal's display. The popup will be centered on the display, and shown on top of the existing page. The popup can be removed by calling the `HidePopup()` function. It will also be removed from the display if a new page is selected by invoking the `GotoPage()` function, or by a suitably defined keyboard action.

FUNCTION TYPE

This function is active.

RETURN TYPE

This function does not return a value.

EXAMPLE

```
ShowPopup(Popup1)
```

SIN(*THETA*)

ARGUMENT	TYPE	DESCRIPTION
theta	**float**	The angle, in radians, to be processed.

DESCRIPTION

Returns the sine of the angle *theta*.

FUNCTION TYPE

This function is passive.

RETURN TYPE

```
float
```

EXAMPLE

```
yp := radius*sin(theta)
```

SINR64(RESULT, TAG)

ARGUMENT	TYPE	DESCRIPTION
result	**int**	The result.
tag	**int**	The angle, in radians, to be processed.

DESCRIPTION

Calculates the sine of *tag* using 64-bit (double precision) floating point math and stores the result in *result*.

The input operand *tag* should be obtained from one of the 64-bit conversion functions provided or from a driver that can read double precision values. All arguments to this function must be integer arrays with lengths of 2. An in depth example is provided in the entry for AddR64.

FUNCTION TYPE

This function is active.

RETURN TYPE

```
void
```

EXAMPLE

```
sinR64(result[0], tag[0])
```

SIRENON()

ARGUMENT	TYPE	DESCRIPTION
none		

DESCRIPTION

Turns on the operator panel's internal siren.

FUNCTION TYPE

This function is active.

RETURN TYPE

This function does not return a value.

EXAMPLE

```
SirenOn()
```

SLEEP(*PERIOD*)

ARGUMENT	TYPE	DESCRIPTION
`period`	`int`	The period for which to sleep, in milliseconds.

DESCRIPTION

Sleeps the current task for the indicated number of milliseconds. This function is normally used within programs that run in the background, or that implement custom communications using Raw Port drivers. Calling it in response to triggers or key presses is not recommended.

FUNCTION TYPE

This function is active.

RETURN TYPE

This function does not return a value.

EXAMPLE

```
Sleep(100)
```

SQRT(*VALUE*)

ARGUMENT	TYPE	DESCRIPTION
value	**int** / **float**	The value to be processed.

DESCRIPTION

Returns the square root of *value*.

FUNCTION TYPE

This function is passive.

RETURN TYPE

int or float, depending on the type of the *value* argument.

EXAMPLE

```
Flow := Const * Sqrt(Input)
```

SQRTR64(RESULT, TAG)

ARGUMENT	TYPE	DESCRIPTION
result	**int**	The result.
tag	**int**	The value to be processed.

DESCRIPTION

Calculates the square root of *tag* using 64-bit (double precision) floating point math and stores the result in *result*.

The input operand *tag* should be obtained from one of the 64-bit conversion functions provided or from a driver that can read double precision values. All arguments to this function must be integer arrays with lengths of 2. An in depth example is provided in the entry for AddR64.

FUNCTION TYPE

This function is active.

RETURN TYPE

```
void
```

EXAMPLE

```
SqrtR64(result[0], tag[0])
```

STDDEV(*ELEMENT, COUNT*)

ARGUMENT	TYPE	DESCRIPTION
element	**int** / **float**	The first array element to be processed.
count	**int**	The number of elements to be processed.

DESCRIPTION

Returns the standard deviation of the *count* array elements from *element* onwards, assuming the data points to represent a sample of the population under study. If you need to find the standard deviation of the whole population, use the PopDev function instead.

FUNCTION TYPE

This function is passive.

RETURN TYPE

```
float
```

EXAMPLE

```
Dev := StdDev(Data[0], 10)
```

STOPSYSTEM()

ARGUMENT	TYPE	DESCRIPTION
none		

DESCRIPTION

Stops the operator interface to allow a user to update the database. This function is typically used when serial programming is required with respect to a unit whose programming port has been allocated for communications. Calling this function shuts down all communications, and thereby allows the port to function as a programming port once more.

FUNCTION TYPE

This function is active.

RETURN TYPE

This function does not return a value.

EXAMPLE

```
StopSystem()
```

STRIP(*TEXT, TARGET*)

ARGUMENT	TYPE	DESCRIPTION
text	**cstring**	The string to be processed.
target	**int**	The character to be removed.

DESCRIPTION

Removes all occurrences of a given character from a text string.

FUNCTION TYPE

This function is passive.

RETURN TYPE

```
cstring
```

EXAMPLE

```
Text := Strip("Mississippi", 's')
```

Text now contains "**Miiippi**".

SUBR64(RESULT, TAG1, TAG2)

ARGUMENT	TYPE	DESCRIPTION
result	**int**	The result.
tag1	**int**	The minuend value.
tag2	**int**	The subtrahend value.

DESCRIPTION

Calculates the value of *tag1* minus *tag2* using 64-bit (double precision) floating point math and stores the result in *result.*

The input operands *tag1* and *tag2* should be obtained from one of the 64-bit conversion functions provided or from a driver that can read double precision values. All arguments to this function must be integer arrays with lengths of 2. An in depth example is provided in the entry for AddR64.

FUNCTION TYPE

This function is active.

RETURN TYPE

```
void
```

EXAMPLE

```
SubR64(result[0], tag1[0], tag2[0])
```

SUBU32(TAG1, TAG2)

ARGUMENT	TYPE	DESCRIPTION
tag1	**int**	The minuend tag.
tag2	**int**	The subtrahend tag.

DESCRIPTION

Returns the value of *tag1* minus *tag2* in an unsigned context.

FUNCTION TYPE

This function is passive.

RETURN TYPE

```
int
```

EXAMPLE

```
Result := SubU32(tag1, tag2)
```

SUM(*ELEMENT*, *COUNT*)

ARGUMENT	TYPE	DESCRIPTION
element	**int** / **float**	The first array element to be processed.
count	**int**	The number of elements to be processed.

DESCRIPTION

Returns the sum of the *count* array elements from *element* onwards.

FUNCTION TYPE

This function is passive.

RETURN TYPE

`int` or `float`, depending on the type of the *value* argument.

EXAMPLE

```
Total := Sum(Data[0], 10)
```

TAN(*THETA*)

ARGUMENT	TYPE	DESCRIPTION
theta	**float**	The angle, in radians, to be processed.

DESCRIPTION

Returns the tangent of the angle *theta*.

FUNCTION TYPE

This function is passive.

RETURN TYPE

```
float
```

EXAMPLE

```
yp := xp * tan(theta)
```

TANR64(RESULT, TAG)

ARGUMENT	TYPE	DESCRIPTION
result	**int**	The result
tag	**int**	The angle, in radians, to be processed.

DESCRIPTION

Calculates the tangent of *tag* using 64-bit (double precision) floating point math and stores the result in *result*.

The input operand *tag* should be obtained from one of the 64-bit conversion functions provided or from a driver that can read double precision values. All arguments to this function must be integer arrays with lengths of 2. An in depth example is provided in the entry for AddR64.

FUNCTION TYPE

This function is active.

RETURN TYPE

```
void
```

EXAMPLE

```
tanR64(result[0], tag[0])
```

TESTACCESS(*RIGHTS*, *PROMPT*)

ARGUMENT	TYPE	DESCRIPTION
rights	**int**	The required access rights.
prompt	**cstring**	The prompt to be used in the log-on popup.

DESCRIPTION

Returns a value of *true* or *false* depending on whether the current user has access rights defined by the *rights* parameter. This parameter comprises a bit-mask representing the various user-defined rights, with bit 0 (ie. the bit with a value of 0x01) representing User Right 1, bit 1 (ie. the bit with a value of 0x02) representing User Right 2 and so on. If no user is currently logged on, the system will display a popup to ask for user credentials, using the *prompt* argument to indicate why the popup is being displayed. The function is typically used in programs that perform a number of actions that might be subject to security, and that might otherwise be interrupted by a log-on popup. By executing this function before the actions are performed, you can provide a better indication to the user as to why a log-on is required, and you can avoid a security failure part way through a series of operations.

FUNCTION TYPE

This function is passive.

RETURN TYPE

```
int
```

EXAMPLE

```
if( TestAccess(1, "Clear all data?") ) {
       Data1 := 0;
       Data2 := 0;
       Data3 := 0;
       }
```

TEXTTOADDR(*ADDR*)

ARGUMENT	TYPE	DESCRIPTION
addr	**cstring**	The addressed in dotted-decimal form.

DESCRIPTION

Converts a dotted-decimal string into a 32-bit IP address.

FUNCTION TYPE

This function is passive.

RETURN TYPE

```
int
```

EXAMPLE

```
ip := TextToAddr("192.168.0.1")
```

TEXTTOFLOAT(*STRING*)

ARGUMENT	TYPE	DESCRIPTION
string	**cstring**	The string to be processed.

DESCRIPTION

Returns the value of *string*, treating it as a floating-point number. This function is often used together with `Mid` to extract values from strings received from raw serial ports. It can also be used to convert other string values into floating-point numbers.

FUNCTION TYPE

This function is passive.

RETURN TYPE

```
float
```

EXAMPLE

```
Data := TextToFloat("3.142")
```

TEXTTOINT(*STRING*, *RADIX*)

ARGUMENT	TYPE	DESCRIPTION
string	**cstring**	The string to be processed.
radix	**int**	The number base to be used.

DESCRIPTION

Returns the value of *string*, treating it as a number of base *radix*. This function is often used together with Mid to extract values from strings received from raw serial ports. It can also be used to convert other string values into integers.

FUNCTION TYPE

This function is passive.

RETURN TYPE

int

EXAMPLE

```
Data := TextToInt("1234", 10)
```

TEXTTOR64(INPUT, OUTPUT)

ARGUMENT	TYPE	DESCRIPTION
input	**cstring**	The text to be converted.
output	**int**	An array with extent 2 that contains the 64-bit result.

DESCRIPTION

Interprets the value stored in the string *input* as a 64-bit (double precision) floating point number and stores the result as an array of length 2 in *output.* The tag *output* should therefore be an integer array with an extent of at least 2. After execution of this function, the value stored in *result* is suitable for use in other 64-bit math functions. See the entry for AddR64 for an example of the intended use of this function.

FUNCTION TYPE

This function is active.

RETURN TYPE

```
void
```

EXAMPLE

```
TextToR64(input, output[0])
```

TIME(*H, M, S*)

ARGUMENT	TYPE	DESCRIPTION
h	**int**	The hour to be encoded, from 0 to 23.
m	**int**	The minute to be encoded, from 0 to 59.
s	**int**	The second to be encoded, from 0 to 59.

DESCRIPTION

Returns a value representing the indicated time as the number of seconds elapsed since midnight. This value can then be used with other time/date functions. It can also be added to the value produced by `Date` to produce a value that references a particular time and date.

FUNCTION TYPE

This function is passive.

RETURN TYPE

```
int
```

EXAMPLE

```
t := Date(2000,12,31) + Time(12,30,0)
```

TXCAN(PORT, DATA, ID)

ARGUMENT	TYPE	DESCRIPTION
port	**int**	The raw 29-bit CAN port
data	**Int**	The first array element holding data to transmit.
ID	**int**	29-bit CAN Identifier

DESCRIPTION

Sends CAN messages that have been initialized with TxCANInit(). The first four bytes of the message to transmit should be set (big endian) in the indicated array element while the remaining 4 bytes if any should be set (big endian) in the next consecutive element of the array. Returns a value of 1 upon success.

FUNCTION TYPE

This function is active.

RETURN TYPE

```
int
```

EXAMPLE

```
TxCAN(8, Data, 0x12345677)
```

TXCANINIT(PORT, ID, DLC)

ARGUMENT	TYPE	DESCRIPTION
port	**int**	The raw 29-bit CAN port
ID	**int**	29-bit CAN Identifier
DLC	**int**	Data Length Count in bytes (1 – 8 bytes are supported)

DESCRIPTION

Initializes CAN messages (Red Lion CAN Option Card required). Initialization function returns a value of 1 upon success or a value of 0 indicating failure. Calls of TxCANInit() should be made only after the system has started and each 29-bit identifier should only be initialized one time via TxCANInit().

FUNCTION TYPE

This function is active.

RETURN TYPE

```
int
```

EXAMPLE

```
TxCANInit(8, 0x12345677, 8)
```

USECAMERASETUP(*PORT, CAMERA, INDEX*)

ARGUMENT	TYPE	DESCRIPTION
port	**int**	The port number where the camera is connected
camera	**int**	The camera device number
index	**int**	The inspection file number in the camera

DESCRIPTION

This function selects the inspection file to be used by the camera. The number to be placed in the *port* argument is the port number to which the driver is bound. The argument *camera* is the device number showing in Crimson 2.0 status bar when the camera is selected. More than one camera can be connected under a single driver. The *index* represents the inspection file number within the camera. This function will return true if the successful, false otherwise.

*Note: This function should be called in a user program that runs in the background to let the camera enough time to change the file.

FUNCTION TYPE

This function is active.

RETURN TYPE

```
int
```

EXAMPLE

```
Success = UseCameraSetup(4, 0, 1)
```

Selects inspection file number 1 on camera device number 0 connected on port 4.

USERLOGOFF()

ARGUMENT	TYPE	DESCRIPTION
none		

DESCRIPTION

Causes the current user to be logged-off the system. Any future actions that require security access rights will result in the display of the log-on popup to allow the entry of credentials.

FUNCTION TYPE

This function is active.

RETURN TYPE

This function does not return a value.

EXAMPLE

```
UserLogOff()
```

USERLOGON()

ARGUMENT	TYPE	DESCRIPTION
none		

DESCRIPTION

Forces the display of the log-on popup to allow the entry of user credentials. You do not normally have to use this function, as Crimson will prompt for credentials when any action that requires security clearance is performed.

FUNCTION TYPE

This function is active.

RETURN TYPE

This function does not return a value.

EXAMPLE

```
UserLogOn()
```

WAITDATA(*DATA*, *COUNT*, *TIME*)

ARGUMENT	TYPE	DESCRIPTION
data	**any**	The first array element to be read.
count	**int**	The number of elements to be read.
time	**int**	The timeout period in milliseconds.

DESCRIPTION

Requests that *count* elements from array element *data* onwards to read on the next comms scan. This function is used with arrays that have been mapped to external data, and which have their read policy set to *Read Manually*. Unlike `ReadData()`, the function waits for up to the time specified by the *time* parameter in order to allow the data to be read. The return value is one if the read completed within that period, or zero otherwise.

FUNCTION TYPE

This function is active.

RETURN TYPE

```
int
```

EXAMPLE

```
status := WaitData(array1[8], 10, 1000)
```

WRITEALL()

ARGUMENT	TYPE	DESCRIPTION
none		

DESCRIPTION

Forces all mapped tags that are not ready-only to be written to their remote devices.

FUNCTION TYPE

This function is active.

RETURN TYPE

This function does not return a value.

WRITEFILE(*FILE, TEXT*)

ARGUMENT	TYPE	DESCRIPTION
file	**int**	The file handle as required by OpenFile.
Text	**cstring**	The text to be written to file.

DESCRIPTION

Writes a string up to 512 characters in length to the specified file and returns the number of bytes successfully written. This function does not automatically include a Line feed and carriage return at the end. For easier programming, refer to `WriteFileLine()`.

FUNCTION TYPE

This function is active.

RETURN TYPE

```
int
```

EXAMPLE

```
count := WriteFile(hFile, "Writing text to file.")
```

WRITEFILELINE(*FILE, TEXT*)

ARGUMENT	TYPE	DESCRIPTION
file	**int**	File handle as required by OpenFile.
text	**cstring**	Text to be written to file.

DESCRIPTION

Writes a string to the specified file and returns the number of bytes successfully written, including the carriage return and linefeed characters that will be appended to each line.

FUNCTION TYPE

This function is active.

RETURN TYPE

```
int
```

EXAMPLE

```
count := WriteFileLine(hFile, "Writing text to file.")
```

SYSTEM VARIABLE REFERENCE

The following pages describe the various system variables that exist within Crimson. These system variables can be invoked within actions or expressions as described in the user manual.

HOW ARE SYSTEM VARIABLES USED

System variables are used either to reflect the state of the system, or to modify the behavior of the system in some way. The former type of variable will be read-only, while the latter type can have a value assigned to it.

ACTIVEALARMS

DESCRIPTION

Returns a count of the currently active alarms.

VARIABLE TYPE

int

ACCESS TYPE

Read-Only.

COMMSERROR

DESCRIPTION

Returns a bit-mask indicating whether or not each communications device is offline. A value of 1 in a given bit position indicates that the corresponding device is experiencing comms errors. Bit 0 (ie. the bit with a value of 1) corresponds to the first communication device.

VARIABLE TYPE

int

ACCESS TYPE

Read-Only.

DISPBRIGHTNESS

DESCRIPTION

Returns a number indicating the brightness of the display from 0 to 100, with zero being off.

VARIABLE TYPE

int

ACCESS TYPE

Read / Write.

DISPCONTRAST

DESCRIPTION

Returns a number indicating the amount of display contrast from 0 to 100.

VARIABLE TYPE

int

ACCESS TYPE

Read / Write.

DISPCOUNT

DESCRIPTION

Returns a number indicating the number of display updates since last reset.

VARIABLE TYPE

int

ACCESS TYPE

Read-Only.

DISPUPDATES

DESCRIPTION

Returns a number indicating how fast the display updates.

VARIABLE TYPE

int

ACCESS TYPE

Read-Only.

ISPRESSED

DESCRIPTION

Return true if the current primitive is being pressed via the touchscreen or web server, and false otherwise. The variable is only valid within the expression or actions that are within the primitive's configuration, or within foreground programs called from those places. Referring to it in other situation will produce an undefined value.

VARIABLE TYPE

int

ACCESS TYPE

Read Only

ISSIRENON

DESCRIPTION

Returns true if the panel's sounder is on or false otherwise.

VARIABLE TYPE

int

ACCESS TYPE

Read-Only.

PI

DESCRIPTION

Returns *pi* as a floating-point number.

VARIABLE TYPE

float

ACCESS TYPE

Read-Only.

TIMENOW

DESCRIPTION

Returns the current time and date as the number of seconds elapsed since the datum point of 1st January 1997. This value can then be used with other time/date functions. Writing to this variable will set the real-time clock to the appropriate time.

VARIABLE TYPE

int

ACCESS TYPE

Read / Write

TIMEZONE

DESCRIPTION

Returns the Time Zone in hours from –12 to +12. Using the Link Send Time command in Crimson will set the unit time and time zone to the computer's values. Changing the Time Zone afterwards will increment or decrement the unit time. Note: TimeZone can only be viewed or changed if the Time Manager is enabled.

VARIABLE TYPE

int

ACCESS TYPE

Read / Write.

TIMEZONEMINS

DESCRIPTION

Returns the Time Zone in minutes from –720 to +720. Using the Link Send Time command in Crimson will set the unit time and time zone to the computer's values. Changing the Time Zone afterwards will increment or decrement the unit time. Note: TimeZoneMins can only be viewed or changed if the Time Manager is enabled.

VARIABLE TYPE

int

ACCESS TYPE

Read / Write.

UNACCEPTEDALARMS

DESCRIPTION

Returns the number of unaccepted alarms in the system.

VARIABLE TYPE

int

ACCESS TYPE

Read

USEDST

DESCRIPTION

Returns the unit daylight saving time state. This variable will add an hour to the unit time if set to true. Note: UseDST can only be viewed or changed if the Time Manager is enabled.

VARIABLE TYPE

flag

ACCESS TYPE

Read / Write.