RED LION®

Crimson® 3.2

Reference Guide | May 2024

LP1157 | Revision B

Corporate Headquarters
Red Lion Controls, Inc.
1750 5th Avenue
York, PA 17403

CONTACT INFORMATION:

AMERICAS

Inside US: +1 (877) 432-9908
Outside US: +1 (717) 767-6511
Hours: 8 am-6 pm Eastern Standard Time
(UTC/GMT -5 hours)

ASIA-PACIFIC

Shanghai, P.R. China: +86 21-6113-3688 x767
Hours: 9 am-6 pm China Standard Time
(UTC/GMT +8 hours)

EUROPE

Netherlands: +31 33-4723-225
France: +33 (0) 1 84 88 75 25
Germany: +49 (0) 1 89 5795-9421
UK: +44 (0) 20 3868 0909
Hours: 9 am-5 pm Central European Time
(UTC/GMT +1 hour)

Website: www.redlion.net
Support: support.redlion.net

Table of Contents

Preface **1**
Disclaimer 1
Trademark Acknowledgments 1
Document History and Related Publications 1
Additional Product Information 1
Chapter 1 Introduction **3**
Supported Devices 3
System Requirements 3
Checking for Updates 3
Getting Assistance 3
Technical Support 3
Online Forums 3
Chapter 2 Standard Functions **5**
Abs(*value*) 6
AbsR64(*result, tag*) 7
acos(*value*) 8
acosR64(*result, tag*) 9
AddR64(*result, tag1, tag2*) 10
AddU32(*tag1, tag2*) 11
AlarmAccept(*alarm*) 12
AlarmAcceptAll(void) 13
AlarmAcceptEx(*source*, *method*, *code*) 14
AlarmAcceptTag(*tag*, *index*, *event*) 15
AreTunnelsActive(void) 16
AreTunnelsBlocked(void) 17
asin(*value*) 18
asinR64(*result, tag*) 19
AsText(*num*) 20
AsTextL64(*data, radix, count*) 21
AsTextR64(*data*) 22
AsTextR64WithFormat(*format, data*) 23
atan(*value*) 24
atan2(*arg1*, *arg2*) 25
atanR64(*result, tag*) 26
atan2R64(*result, arg1, arg2*) 27
Beep(*freq*, *period*) 28
BlockTunnels(*enable*) 29
CanGotoNext(void) 30
CanGotoPrevious(void) 31

ClearEvents(void) 32
CloseFile(*file*) 33
ColBlend(*data*, *min*, *max*, *col1*, *col2*) 34
ColFlash(*freq*, *col1*, *col2*) 35
ColGetBlue(*col*) 36
ColGetGreen(*col*) 37
ColGetRed(*col*) 38
ColGetRGB(*r*, *g*, *b*) 39
ColPick2(*pick*, *col1*, *col2*) 40
ColPick4(*data1*, *data2*, *col1*, *col2*, *col3*, *col4*) 41
ColSelFlash(*enable*, *freq*, *col1*, *col2*, *col3*) 42
CommitAndReset(void) 43
CommitPersonality(*reset*) 44
CompactFlashEject(void) 45
CompactFlashStatus(void) 46
CompU32(*tag1*, *tag2*) 47
ControlDevice(*device*, *enable*) 48
Copy(*dest*, *src*, *count*) 49
CopyFiles(*source*, *target*, *flags*) 50
cos(*theta*) 51
cosR64(*result*, *tag*) 52
CreateDirectory(*name*) 53
CreateFile(*name*) 54
DataToText(*data*, *limit*) 55
Date(*y*, *m*, *d*) 56
DebugDumpLocals(void) 57
DebugPrint(*text*) 58
DebugStackTrace(void) 59
DecR64(*result*, *tag*) 60
DecToText(*data*, *signed*, *before*, *after*, *leading*, *group*) 61
Deg2Rad(*theta*) 62
DeleteDirectory(*name*) 63
DeleteFile(*file*) 64
DevCtrl(*device*, *function*, *data*) 65
DisableDevice(*device*) 66
DispOff(void) 67
DispOn(void) 68
DivR64(*result*, *tag1*, *tag2*) 69
DivU32(*tag1*, *tag2*) 70
DrvCtrl(*port*, *function*, *data*) 71
EjectDrive(*drive*) 72

EmptyWriteQueue(*dev*) 73
EnableBatteryCheck(*disable*) 74
EnableDevice(*device*) 75
EndBatch(void) 76
EndModal(*code*) 77
EnumOptionCard(*s*) 78
EqualR64(*arg1*, *arg2*) 79
exp(*value*) 80
exp10(*value*) 81
exp10R64(*result*, *tag*) 82
expR64(*result*, *tag*) 83
FileSeek(*file*, *pos*) 84
FileTell(*file*) 85
Fill(*element*, *data*, *count*) 86
Find(*string*, *char*, *skip*) 87
FindFileFirst(*dir*) 88
FindFileNext(void) 89
FindTagIndex(*label*) 90
Flash(*freq*) 91
Force(*dest*, *data*) 92
ForceCopy(*dest*, *src*, *count*) 93
ForceSQLSync(void) 94
FormatCompactFlash(void) 95
FormatDrive(*drive*) 96
FtpGetFile(*server*, *loc*, *rem*, *delete*) 97
FtpPutFile(*server*, *loc*, *rem*, *delete*) 98
GetAlarmTag(*index*) 99
GetAutoCopyStatusCode(void) 100
GetAutoCopyStatusText(void) 101
GetBatch(void) 102
GetCameraData(*port*, *camera*, *param*) 103
GetCurrentUserName(void) 104
GetCurrentUserRealName(void) 105
GetCurrentUserRights(void) 106
GetDate (*time*) and Family 107
GetDeviceStatus(*device*) 108
GetDiskFreeBytes(*drive*) 109
GetDiskFreePercent(*drive*) 110
GetDiskSizeBytes(*drive*) 111
GetDriveStatus(*drive*) 112
GetFileByte(*file*) 113

GetFileData(*file*, *data*, *length*) 114
GetFormattedTag(*index*) 115
GetInterfaceStatus(*port*) 116
GetIntTag(*index*) 117
GetLanguage(void) 118
GetLastEventText(*all*) 119
GetLastEventTime(*all*) 120
GetLastEventType(*all*) 121
GetLastSQLSyncStatus(void) 122
GetLastSQLSyncTime(*Request*) 123
GetLicenseState(*type*) 124
GetLocationPropery(*index, property*) 125
GetModelName(*code*) 126
GetModemProperty(*index, property*) 127
GetMonthDays(*y*, *m*) 128
GetNetGate(*port*) 129
GetNetId(*port*) 130
GetNetIp(*port*) 131
GetNetMask(*port*) 132
GetNow(void) 133
GetNowDate(void) 134
GetNowTime(void) 135
GetPersonalityFloat(*name*) 136
GetPersonalityInt (name) 137
GetPersonalityIp(*name*) 138
GetPersonalityString(*name*) 139
GetPortConfig(*port*, *param*) 140
GetQueryStatus(q*uery*) 141
GetQueryTime(q*uery*) 142
GetRealTag(*index*) 143
GetRestartCode(*index*) 144
GetRestartInfo(*index*) 145
GetRestartText(*index*) 146
GetRestartTime(*index*) 147
GetSQLConnectionStatus(void) 148
GetStringTag(*index*) 149
GetTagLabel(*index*) 150
GetSystemIo(*prop*) 151
GetTcpSocketPeer(*port, interface*) 152
GetUpDownData(*data*, *limit*) 153
GetUpDownStep(*data*, *limit*) 154

GetVersionInfo(*code*) 155
GetWebParamHex(*param*) 156
GetWebParamInt(*param*) 157
GetWebParamStr(*param*) 158
GetWifiProperty(*index, property*) 159
GotoNext() 160
GotoPage(*name*) 161
GotoPrevious() 162
GreaterEqR64(*arg1*, *arg2*) 163
GreaterR64(*arg1*, *arg2*) 164
HasAccess(*rights*) 165
HasAllAccess(*rights*) 166
HideAllPopups(void) 167
HidePopup(void) 168
IntR64(*result*, *tag*) 169
IntToR64(*result*, *tag*) 170
IntToText(*data*, *radix*, *count*) 171
IsBatchNameValid(*name*) 172
IsBatteryLow(void) 173
IsDeviceOnline(*device*) 174
IsLoggingActive(void) 175
IsPortRemote(*port*) 176
IsTcpSocketActive(*port, interface*) 177
IsSQLSyncRunning(void) 178
IsWriteQueueEmpty(*dev*) 179
KillDirectory(*name*) 180
Left(*string*, *count*) 181
Len(*string*) 182
LessEqR64(*arg1*, *arg2*) 183
LessR64(*arg1*, *arg2*) 184
LoadCameraSetup(*port*, *camera*, *index*, *file*) 185
LoadSecurityDatabase(*mode*, *file*) 186
log(*value*) 187
log10(*value*) 188
log10R64(*result, tag*) 189
LogBatchComment(*set*, *text*) 190
LogBatchHeader(*set*, *text*) 191
LogComment(log, text) 192
LogHeader(*log*, *text*) 193
logR64(*result*, *tag*) 194
LogSave(void) 195

MakeFloat(*value*) 196
MakeInt(*value*) 197
Max(*arg1*, arg2) 198
MaxR64(result, tag1, tag2) 199
MaxU32(*tag1*, *tag2*) 200
Mean(*element*, *count*) 201
Mid(*string*, *pos*, *count*) 202
Min(*arg1*, arg2) 203
MinR64(*result*, *tag1*, *tag2*) 204
MinU32(*tag1*, *tag2*) 205
MinusR64(*result*, *tag*) 206
ModU32(*tag1*, *tag2*) 207
MountCompactFlash(*enable*) 208
MoveFiles(*source*, *target*, *flags*) 209
MulDiv(*arg1*, arg2, arg3) 210
MulR64(*result*, *tag1*, *tag2*) 211
MulU32(*tag1*, *tag2*) 212
MuteSiren(void) 213
NetworkPing(*address*, *timeout*) 214
NewBatch(*name*) 215
Nop(void) 216
NotEqualR64(*arg1*, arg2) 217
OpenFile(*name*, *mode*) 218
Pi(void) 219
PlayRTTTL(*tune*) 220
PopDev(*element*, *count*) 221
PortClose(*port*) 222
PortGetCTS(*port*) 223
PortInput(*port*, *start*, *end*, *timeout*, *length*) 224
PortPrint(*port*, *string*) 225
PortPrintEx(*port*, *string*) 226
PortRead(*port*, *period*) 227
PortSendData(port, data, count) 228
PortSetRTS(*port*, *state*) 229
PortWrite(*port*, *data*) 230
PostKey(*code*, *transition*) 231
Power(*value*, *power*) 232
PowR64(*result*, *value*, *power*) 233
PrintScreenToFile(*path*, *name*, *res*) 234
PutFileByte(*file*, *data*) 235
PutFileData(*file*, *data*, *length*) 236

R64ToInt(*value*) ... 237
R64ToReal(*value*) ... 238
Rad2Deg(*theta*) ... 239
Random(*range*) ... 240
ReadData(*data*, *count*) ... 241
ReadFile(*file*, *chars*) ... 242
ReadFileLine(*file*) ... 243
RealToR64(*result*, *n*) ... 244
RenameFile(*file*, *name*) ... 245
ResolveDNS(*name*) ... 246
Right(*string*, *count*) ... 247
RShU32(*arg1*, *arg2*) ... 248
RunAllQueries(void) ... 249
RunQuery(*query*) ... 250
RxCAN(*port*, *data*, *id*) ... 251
RxCANInit(*port*, *id*, *dlc*) ... 252
SaveCameraSetup(*port*, *camera*, *index*, *file*) ... 253
SaveConfigFile(*file*) ... 254
SaveSecurityDatabase(*mode*, *file*) ... 255
Scale(*data*, *raw1*, *raw2*, *eng1*, *eng2*) ... 256
SendFile(*rcpt*, *file*) ... 257
SendFileEx(*rcpt*, *file*, *subject*, *flag*) ... 258
SendFileTo(*rcpt*, *file*) ... 259
SendFileToAck(*rcpt, file, ack*) ... 260
SendMail(*rcpt*, *subject*, *body*) ... 261
SendMailTo(*rcpt, subject, message*) ... 262
SendMailToAck(*rcpt, subject, message, ack*) ... 263
Set(*tag*, *value*) ... 264
SetIconLed(*id*, *state*) ... 265
SetIntTag(*index*, *value*) ... 266
SetSystemIo(*prop, value*) ... 267
SetLanguage(*code*) ... 268
SetNow(*time*) ... 269
SetPersonalityFloat(*name, data*) ... 270
SetPersonalityInt(*name, data*) ... 271
SetPersonalityIp(*name, data*) ... 272
SetPersonalityString(*name, data*) ... 273
SetRealTag(*index*, *value*) ... 274
SetStringTag(*index*, *data*) ... 275
Sgn(*value*) ... 276
ShowMenu(*name*) ... 277

ShowModal(*name*) 278
ShowNested(*name*) 279
ShowPopup(*name*) 280
sin(*theta*) 281
sinR64(*result, tag*) 282
SirenOn(void) 283
Sleep(*period*) 284
Sqrt(*value*) 285
SqrtR64(*result, tag*) 286
StdDev(*element, count*) 287
StopRTTTL(void) 288
StopSystem(void) 289
Strip(*text, target*) 290
SubR64(*result, tag1, tag2*) 291
SubU32(*tag1, tag2*) 292
Sum(*element, count*) 293
tan(*theta*) 294
tanR64(*result, tag*) 295
TestAccess(*rights, prompt*) 296
TextToAddr(*address*) 297
TextToFloat(*string*) 298
TextToInt(*string, radix*) 299
TextToL64(*input, output, radix*) 300
TextToR64(*input, output*) 301
Time(*h, m, s*) 302
TxCAN(*port, data, id*) 303
TxCANInit(*port, id, dlc*) 304
UseCameraSetup(*port, camera, index*) 305
UserLogOff(void) 306
UserLogOn(void) 307
WaitData(*data, count, time*) 308
WriteAll(void) 309
WriteFile(*file, text*) 310
WriteFileLine(*file, text*) 311
Chapter 3 System Variables 313
ActiveAlarms 314
CommsError 315
DispBrightness 316
DispContrast 317
DispCount 318
DispUpdates 319

IconBrightness 320
IsPressed 321
IsSirenOn 322
Pi 323
TimeNow 324
TimeZone 325
TimeZoneMins 326
Unaccepted Alarms 327
UnacceptedAndAutoAlarms 328
UseDST 329

Preface

Disclaimer

While every effort has been made to ensure that this document is complete and accurate at the time of release, the information that it contains is subject to change. Red Lion Controls, Inc. is not responsible for any additions to or alterations of the original document. Industrial networks vary widely in their configurations, topologies, and traffic conditions. This document is intended as a general guide only. It has not been tested for all possible applications, and it may not be complete or accurate for some situations.

This guide is intended to be used by personnel responsible for configuring and commissioning Crimson® devices for use in visualization, monitoring and control applications. Users of this document are urged to heed warnings and cautions used throughout the document.

Trademark Acknowledgments

Red Lion Controls, Inc. acknowledges and recognizes ownership of the following trademarked terms used in this document.

- EtherNet/IP™and CIP™ are trademarks of ODVA.
- Microsoft®, Windows®, Windows NT®, and Windows Vista™ are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

All other marks are the property of their respective owners.

Document History and Related Publications

The hard copy and electronic media versions of this document are revised only at major releases and therefore, may not always contain the latest product information. Tech Notes and/or product addendums will be provided as needed between major releases to describe any new information or document changes.

The latest online version of this document can be accessed through the Red Lion website at https://www.redlion.net/red-lion-software/crimson/crimson-32.

Additional Product Information

Additional product information can be obtained by contacting your local sales representative or Red Lion through the contact numbers and/or support website address listed on the inside of the front cover.

Chapter 1 Introduction

Crimson® 3.2 is the latest version of Red Lion's widely-acclaimed Crimson device configuration software. This Reference Guide augments the Crimson 3.2 Software Guide by detailing the Standard Functions and System Variables available within Crimson 3.2. These features help Crimson 3.2 users design powerful and attractive Crimson device solutions more easily and efficiently.

Supported Devices

Crimson 3.2 is designed to work with Red Lion's FlexEdge® products, Graphite® HMIs and Edge Controller, CR3000 and CR1000 HMIs and the DA10 and DA30 Data Station devices. Databases can be imported from any devices that are supported by Crimson 3.1 or Crimson 3.0, but earlier versions of Crimson will still be required if you want to configure those products. Red Lion plans to add support for further products to Crimson 3.2 over time, so please check back regularly for updates.

System Requirements

Crimson 3.2 is designed to run on any version of Microsoft Windows, from Windows 7 onwards. Memory requirements are modest and any system that meets the minimum system requirements for its operating system will be able to run Crimson 3.2. About 1GB of free disk space will be needed for installation, and you should ideally have a display with sufficient resolution to allow the editing of display pages without having to scroll.

Checking for Updates

You may manually download the latest Crimson 3.2 version from the Red Lion website by visiting the Downloads page within the Support section.

Getting Assistance

If you experience a problem or need assistance, the following resources are available.

Technical Support

Technical assistance is available on the web at: support.redlion.net
You may also call:
Inside US: +1 (877) 432-9908
Outside US: +1 (717) 767-6511

Online Forums

A number of online forums exist to support users of PLCs and HMIs. Red Lion recommends the Q&A forum at http://www.plctalk.net/qanda/. The discussion board is populated by many experts who are willing to help, and Red Lion's own technical support staff monitors this forum for questions relating to our products.

Chapter 2 Standard Functions

This chapter describes the standard functions that are provided in Crimson® 3.2. These functions can be invoked within programs, actions, or expressions, as described in the Crimson 3.2 Software Guide. 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

```
float 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

This function does not return a value.

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

```
float 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

This function does not return a value.

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

This function does not return a value.

Example

This example shows how to calculate π + 2 using 64-bit math.
`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", this being 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 sum of *tag1* and *tag2* in an unsigned context.

Function Type

This function is *passive*.

Return Type

```
int
```

Example

```
int 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(void)

ARGUMENT	TYPE	DESCRIPTION
`none`		

Description

Accepts all active alarms.

Function Type

This function is *active*.

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® 3.2.

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 an alarm generated by the 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

```
AlarmAcceptTag(10, 0, 1);
```

AreTunnelsActive(void)

ARGUMENT	TYPE	DESCRIPTION
`none`		

Description

Indicates whether networking tunnels are active. Currently only supported for FlexEdge® devices.

Function Type

This function is *passive*.

Return Type

```
int
```

Example

```
int State = AreTunnelsActive();
```

AreTunnelsBlocked(void)

ARGUMENT	TYPE	DESCRIPTION
`none`		

Description

Indicates whether networking tunnels are blocked. Currently only supported for FlexEdge® devices.

Function Type

This function is *passive*.

Return Type

```
int
```

Example

```
int State = AreTunnelsBlocked();
```

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

```
float 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 the `AddR64()` function.

Function Type

This function is *active*.

Return Type

This function does not return a value.

Example

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

AsText(*num*)

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

Description

Returns the numeric value *num*, formatted as a string equivalent to that performed by the General numeric format.

Note that numeric tags can be converted to strings by using their `AsText` property, for example `Tag1.AsText`.

Function Type

This function is *passive*.

Return Type

```
cstring
```

Example

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

AsTextL64(*data, radix, count*)

ARGUMENT	TYPE	DESCRIPTION
data	int	The 64-bit integer value to convert.
radix	Int	The number base used.
count	int	The number of digits to generate.

Description

Converts the value stored in *data* from a 64-bit integer 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 *radix* argument defines the number base - use 8 for octal, 10 for decimal and 16 for hexadecimal. The *count* argument defines the resulting text width.

Function Type

This function is *passive*.

Return Type

```
cstring
```

Example

```
cstring Text = AsTextL64(data[0], 10, 20);
```

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 *data* must be an integer array data tag 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

```
cstring Text = AsTextR64(data[0]);
```

AsTextR64WithFormat(*format, data*)

ARGUMENT	TYPE	DESCRIPTION
`format`	`cstring`	A string containing the desired width, precision and flags.
`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 per the *format* specified. The *format* should be encoded as "width.precision.flags", where the width defines the maximum number of characters representing the numerical portion of the resulting string and the precision defines the number of numerical characters to the right of the decimal point in the resulting string. Flags available are 1 to show leading zeros and 2 to hide trailing 0. 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

```
cstring Text = AsTextR64WithFormat("17.8.3", 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

```
float theta = atan(1.0);
```

atan2(*arg1*, *arg2*)

ARGUMENT	TYPE	DESCRIPTION
`arg1`	`float`	The value of the side that is opposite the angle theta.
`arg2`	`float`	The value of the side that is adjacent to the angle theta.

Description

The equivalent of `atan(a/b)`, except that it also considers the sign of *arg1* and *arg2*, and thereby ensures that the return value is in the appropriate quadrant. It is also capable of handling a zero value for *arg2*, 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

```
float 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

This function does not return a value.

Example

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

atan2R64(*result, arg1, arg2*)

ARGUMENT	TYPE	DESCRIPTION
`result`	`int`	The result.
`arg1`	`int`	The value of the side that is opposite the angle theta.
`arg2`	`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 *arg1* and *arg2* to calculate the value for the appropriate quadrant. It is the double precision equivalent of the `atan2()` function. The input operands *arg1* and *arg2* 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

This function does not return a value.

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 Crimson® device'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);
```

BlockTunnels(*enable*)

ARGUMENT	TYPE	DESCRIPTION
`enable`	`int`	Enable blocking of networking tunnels.

Description

Blocks networking tunnels according to the *enable* value. Currently only supported for the FlexEdge® products.

Function Type

This function is *active*.

Return Type

This function does not return a value.

Example

```
BlockTunnels(true);
```

CanGotoNext(void)

ARGUMENT	TYPE	DESCRIPTION
`none`		

Description

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

Function Type

This function is *passive*.

Return Type

```
int
```

Example

```
if( CanGotoNext() ) {

    GotoNext();
}
```

CanGotoPrevious(void)

ARGUMENT	TYPE	DESCRIPTION
`none`		

Description

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

Function Type

This function is *passive*.

Return Type

```
int
```

Example

```
if( CanGotoPrevious() ) {

    GotoPrevious ();
}
```

ClearEvents(void)

ARGUMENT	TYPE	DESCRIPTION
`none`		

Description

Clears the event log of stored events.

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 by a call to `OpenFile()`.

Function Type

This function is *active*.

Return Type

This function does not return a value.

Example

```
int hFile = OpenFile(“MyFile.txt”, 0);

if( hFile ) {

   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`

Example

```
int Color = ColBlend(Tag1,Tag1.Min,Tag1.Max,0x7C00,0x001F);
```

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
```

Example

```
int Color = ColFlash(10,0x01E0,0x001F);
```

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
```

Example

```
int Color = ColGetBlue(0x618C);
```

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
```

Example

```
int Color = ColGetGreen(0x618C);
```

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
```

Example

```
int Color = ColGetRed(0x618C);
```

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`

Example

```
int Color = ColGetRGB(150, 150, 150);
```

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 col1 or col2, depending on the state of pick. Equivalent results can be achieved using the ?: selection operator.

Function Type

This function is *passive*.

Return Type

```
int
```

Example

```
int Color = ColPick2(Tag1,0x01E0,0x001F);
```

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 *true*.
`col2`	`int`	The value when Data1 is *false* and Data2 is *true*.
`col3`	`int`	The value when Data1 is *true* and Data2 is *false*.
`col4`	`int`	The value when both Data1 and Data2 are *false*.

Description

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

Function Type

This function is *passive*.

Return Type

`int`

Example

```
int Color = ColPick4(Tag1,Tag2,0x03E0,0x001F,0x01E0,0x000F);
```

ColSelFlash(*enable*, *freq*, *col1*, *col2*, *col3*)

ARGUMENT	TYPE	DESCRIPTION
`enable`	`int`	The 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
```

Example

```
int Color = ColSelFlash(Tag2,Tag1,0x3C00,0x01E0,0x001F);
```

CommitAndReset(void)

ARGUMENT	TYPE	DESCRIPTION
`none`		

Description

Forces 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 functions that change the configuration of the unit, and that then require a reset for the changes to take effect.

Function Type

This function is *active*.

Return Type

This function does not return a value.

Example

```
CommitAndReset();
```

CommitPersonality(*reset*)

ARGUMENT	TYPE	DESCRIPTION
`reset`	`int`	Determines unit behavior.

Description

Commits new Key Values in the Device Personality to internal memory. The device will reset immediately afterwards if the argument *reset* is non-zero.

Function Type

This function is *active*.

Return Type

This function does not return a value.

Example

```
CommitPersonality(0);
```

CompactFlashEject(void)

ARGUMENT	TYPE	DESCRIPTION
`none`		

Description

Ceases all access of the memory 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(void)

ARGUMENT	TYPE	DESCRIPTION
`none`		

Description

Returns the status of the memory card slot as an integer. FlexEdge® Products - Returns the status of the internal memory.

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.
2	Checking	The HMI is checking the status of the card. This state occurs when a card is first inserted into the HMI.
3	Formatting	The HMI is formatting the card. This state occurs when a format operation is requested by the programming PC.
4	Locked	The Crimson® device 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 Crimson device or Windows.

Function Type

This function is *passive*.

Return Type

```
int
```

Example

```
int 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

```
int 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

Disables or enables the communications device specified by the *device* argument, according to the *enable* argument. The device number is displayed in Crimson®'s status bar of the Communications category when the device is selected.

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, as follows:

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

```
int Result = 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

```
float 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

This function does not return a value.

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 memory card. The Crimson® 3.2 filing system now supports both FAT16 and FAT32. If the memory card has been formatted to use FAT32, then long filenames may be used. If the memory card has been formatted using FAT16, then long filenames are not supported. Note 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 within the Crimson 3.2 Software Guide. 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.

When using this function with a FlexEdge® device the directory can be created on the internal memory, Memory card, or Memory stick. By default, the function will create the directory on the internal memory. If you wish to create the directory elsewhere you must specify the driver letter.

Function Type

This function is *active*.

Return Type

```
int
```

Example

```
int Result = CreateDirectory("/LOGS/LOG1");
int Result = CreateDirectory("D:/LOGS/LOG1");  // Memory Stick  *FlexEdge Only*
int Result = CreateDirectory("E:/LOGS/LOG1");  // SD Card  *FlexEdge Only*
```

CreateFile(*name*)

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

Description

Creates an empty file on the memory card. The Crimson® 3.2 filing system now supports both FAT16 and FAT32. If the memory card has been formatted to use FAT32, then long filenames may be used. If the memory card has been formatted using FAT16, then long filenames are not supported. Note 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 within the Crimson 3.2 Software Guide. 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.

When using this function with a FlexEdge® device the file can be created on the internal memory, Memory card, or Memory stick. By default, the function will create the file on the internal memory. If you wish to create the file elsewhere you must specify the driver letter.

Function Type

This function is *active*.

Return Type

```
int
```

Example

```
  int Result = CreateFile("/logs/custom/myfile.txt");
  int Result = CreateFile("D:/logs/custom/myfile.txt"); // Memory Stick *FlexEdge
Only*
  int Result = CreateFile("E:/logs/custom/myfile.txt"); // Memory Card *FlexEdge
Only*
```

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, extracting 4 characters from each 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

```
cstring Text = 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

```
int t = Date(2000,12,31);
```

DebugDumpLocals(void)

ARGUMENT	TYPE	DESCRIPTION
none		

Description

Outputs a list of local variables and their current value to the configured Debug Console.

Function Type

This function is *active*.

Return Type

This function does not return a value.

Example

```
DebugDumpLocals()

Results in the following output:

025.080 : USER     :   Locals:
025.080 : USER     :     i = 112

Format is as follows:

Seconds.milliseconds : TASK : Locals
```

DebugPrint(*text*)

ARGUMENT	TYPE	DESCRIPTION
`text`	`Cstring`	Text to output to the debug console.

Description

Outputs the desired text to the configured Debug Console.

Function Type

This function is *active*.

Return Type

This function does not return a value.

Example

```
cstring Text = "Local variable i is a value of ";

Text += IntToText(i, 10, 4);

DebugPrint(Text);


Results in the following output:

228.100 : USER     :   Local variable i is a value of 0003

Format is as follows:

Seconds.milliseconds : TASK : Text
```

DebugStackTrace(void)

ARGUMENT	TYPE	DESCRIPTION
none		

Description

Outputs text representing the area of code execution to the configured Debug Console.

Function Type

This function is *active*.

Return Type

This function does not return a value.

Example

```
DebugStackTrace();

Results in the following output:

141.075 : USER     :   Stack:
141.075 : USER     :     Program1

Format is as follows:

Seconds.milliseconds : TASK : Stack
```

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

This function does not return a value.

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 per the rest of the parameters. The function is typically used to generate advanced formatting options via programs, or to prepare strings to be sent via a raw port driver. Refer to the Numeric format section in the Crimson® 3.2 Software Guide.

Function Type

This function is *passive*.

Return Type

```
cstring
```

Example

```
cstring 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

```
float Load = Weight * cos(Deg2Rad(Angle));
```

DeleteDirectory(*name*)

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

Description

Removes an empty directory (i.e. one that contains no files and/or subdirectories) from the memory card. The Crimson® 3.2 filing system now supports both FAT16 and FAT32. If the memory card has been formatted to use FAT32, then long filenames may be used. If the memory card has been formatted using FAT16, then long filenames are not supported. Note 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 within the Crimson 3.2 Software Guide. To avoid this complication, forward slashes can be used in place of back slashes without the need for such doubling. The function returns a value of one if it succeeds, and a value of zero if it fails.

When using this function with a FlexEdge® device the directory can be deleted from the internal memory, Memory card, or Memory stick. By default, the function will delete the directory on the internal memory. If you wish to delete a directory elsewhere you must specify the driver letter.

Function Type

This function is *active*.

Return Type

```
int
```

Example

```
int Result = DeleteDirectory("/logs/custom");
int Result = DeleteDirectory("D:/logs/custom");  // Memory Stick  *FlexEdge Only*
int Result = DeleteDirectory("E:/logs/custom");  // SD Card  *FlexEdge Only*
```

DeleteFile(*file*)

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

Description

Closes and then deletes a *file* located on the memory card. The file must first be opened in a writeable state.

Function Type

This function is *active*.

Return Type

`int`

Example

```
int hFile = OpenFile("/LOGS/LOG1/01010101.csv", 1);

if( hFile ) {
   int Result = DeleteFile(hFile);

   CloseFile(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

Performs a special operation on the communications device specified by the *device* argument. The specific action to be performed is indicated by the *function* parameter, the values of which will depend upon the type of device being addressed. The *data* parameter may be used to pass additional information to the driver. Most drivers do not support this function. Where supported, the operations are driver-specific, and are documented separately. The device number is displayed in Crimson®'s status bar of the Communications category when the device is selected.

Function Type

This function is *active*.

Return Type

`int`

Example

Refer to the communications driver application notes for specific examples, available online at: http://www.redlion.net/red-lion-software/crimson/crimson-32.

DisableDevice(*device*)

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

Description

Disables communications for the specified *device*. The device number is displayed in Crimson®'s status bar of the Communications category when the device is selected.

Function Type

The function is *passive*.

Return Type

This function does not return a value.

Example

```
DisableDevice(1);
```

DispOff(void)

ARGUMENT	TYPE	DESCRIPTION
`none`		

Description

Turns the display backlight off.

Function Type

This function is *active*.

Return Type

This function does not return a value.

Example

```
DispOff();
```

DispOn(void)

ARGUMENT	TYPE	DESCRIPTION
`none`		

Description

Turns the display backlight 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 under `AddR64()`.

Function Type

This function is *active*.

Return Type

This function does not return a value.

Example

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

DivU32(*tag1, tag2*)

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

Description

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

Function Type

This function is *passive*.

Return Type

```
int
```

Example

```
int Result = DivU32(tag1, tag2);
```

DrvCtrl(*port, function, data*)

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

Performs 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 additional information to the driver. Most drivers do not support this function. Where supported, the operations are driver-specific, and are documented separately. The port number is displayed in Crimson®'s status bar of the Communications category when the port is selected.

Function Type

This function is *active*.

Return Type

`int`

Example

Refer to the communications driver application notes for specific examples, available online at: http://www.redlion.net/red-lion-software/crimson/crimson-32.

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 memory card, while Drive D refers to the USB memory stick.
FlexEdge® Products – Drive ‘C’ refers to the internal memory, while Drive ‘D’ refers to the Memory card and Drive ‘E’ 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 pending 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 Communications tree.

Function Type

This function is *passive*.

Return Type

This function does not return a value.

Example

```
EmptyWriteQueue(1);
```

EnableBatteryCheck(*disable*)

ARGUMENT	TYPE	DESCRIPTION
`disable`	`int`	Set to 1 to disable the check.

Description

Enable or disable the battery check that is performed after the system starts up. Set *disable* to 0 to enable the battery check. Set *disable* to 1 to disable the battery check. If the battery check is enabled and the battery is low, the system will show a warning screen to inform the user. The user must either wait 60 seconds or follow the on-screen instructions to proceed past the warning. If the battery check is disabled, the battery low warning screen will never be displayed, regardless of the battery's status.

Function Type

This function is *active*.

Return Type

This function does not return a value.

Example

```
EnableBatteryCheck(0)
```

Note: This function is deprecated in Crimson® 3.2.

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 is highlighted.

Function Type

This function is *passive*.

Return Type

This function does not return a value.

Example

```
EnableDevice(1);
```

EndBatch(void)

ARGUMENT	TYPE	DESCRIPTION
`none`		

Description

Stops the current batch. Note that 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

```
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 by 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.

Example

```
EndModal(1);
```

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`

Note: This function is deprecated in Crimson® 3.2.

EqualR64(*arg1*, *arg2*)

ARGUMENT	TYPE	DESCRIPTION
`arg1`	`int`	The first value to compare.
`arg2`	`int`	The second value to compare.

Description

Compares the value of *arg1* to *arg2* using 64-bit double precision floating point math and returns 1 if *arg1* is equal to *arg2*, 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 *arg1* and *arg2* 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

```
int Result = 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

```
float 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

```
float Variable4 = exp10(0.699);
```

exp10R64(*result, tag*)

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

Description

Calculates the value of 10 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

This function does not return a value.

Example

```
Exp10R64(result[0], tag1[0]);
```

expR64(*result, tag*)

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

Description

Calculates the value of *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

This function does not return a value.

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 location indicated by argument *pos*.

Function Type

This function is *active*.

Return Type

```
int
```

Example

```
int Result = FileSeek(“MyFile.txt”, 100);
```

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
```

Example

```
int Pos = FileTell("MyFile.txt");
```

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*, ignoring the first *skip* occurrences specified. The first position counted is 0. Returns -1 if *char* is not found.

In the example below, the position of the colon, skipping the first occurrence, is 7.

Function Type

This function is *passive*.

Return Type

```
int
```

Example

```
int Pos = 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 memory card. Returns an empty string if no files exist or if no memory card is present. This function can be used with the `FindFileNext()` function to scan all files in each directory.

When using this function with a FlexEdge® device the internal memory, Memory card, or Memory stick can be referenced. By default, the function will reference the internal memory. If you wish to reference a different drive, you must specify the drive letter.

Function Type

This function is *active*.

Return Type

`cstring`

Example

```
cstring Name = FindFileFirst("/LOGS/LOG1/");
cstring Name = FindFileFirst("D:/LOGS/LOG1/");  // Memory Stick  *FlexEdge Only*
cstring Name = FindFileFirst("E:/LOGS/LOG1/");  // SD Card  *FlexEdge Only*
```

FindFileNext(void)

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, or an empty string if no more files exist. This function can be used with the `FindFileFirst()` function to scan all files in each directory.

Function Type

This function is *active*.

Return Type

`cstring`

Example

```
FindFileFirst();
cstring Name = FindFileNext();
```

FindTagIndex(*label*)

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

Description

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

Function Type

This function is *active*.

Return Type

`int`

Example

```
Int Index = FindTagIndex(“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
```

Example

```
int Flash = Flash(10);
```

Force(*dest, data*)

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

Description

Sets the data tag specified by *dest* to the value specified by *data*. 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 irrespective of whether the data value has changed. It is used in situations where Crimson®'s normal data tag write-behavior is not required.

Function Type

This function is *active*.

Return Type

This function does not return a value.

Example

```
Force(Group1.Tag1, 3.14);
```

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 irrespective of whether the original data has changed. It is used in situations where Crimson®'s normal data tag write-behavior is not required.

Function Type

This function is *active*.

Return Type

This function does not return a value.

Example

```
ForceCopy(Group1.Tag1[3], Group1.Tag2[13], 10);
```

ForceSQLSync(void)

ARGUMENT	TYPE	DESCRIPTION
`none`		

Description

Forces the SQL Sync service to run immediately and transmit log data to the configured SQL Server, only when 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(void)

ARGUMENT	TYPE	DESCRIPTION
`none`		

Description

Formats the memory card in the Crimson® device, thereby deleting all data on the card. You should 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 memory card, while Drive ‘D’ refers to the USB memory stick.
FlexEdge® Products – Drive ‘C’ refers to the internal memory, while Drive ‘D’ refers to the USB memory stick and Drive ‘E’ refers to the SD card.

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 memory card.
`rem`	`cstring`	The remote file name on the FTP server.
`delete`	`int`	If true, the source will be deleted after the transfer. If false, it will remain on the source disk.

Description

Transfers the defined file from the FTP server to the Crimson® device memory card. It will return *true* if the transfer is successful and *false* otherwise. The source and destination file names can be different. The remote path is relative to the FTP server setting root path. See the Synchronization Manager for more details.

Function Type

This function is *active*.

Return Type

```
int
```

Example

```
int Result = FtpGetFile(0, “/Recipes.csv”, “/Recipes/Rec001.csv”, 0);
```

FtpPutFile(*server, loc, rem, delete*)

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

Description

Transfers the defined file from the Crimson® device memory card to the FTP server. It will return *true* if the transfer is successful, and *false* otherwise. The source and destination file names can be different. The remote path is relative to the FTP server setting root path. See the Synchronization Manager for more details.

Function Type

This function is *active*.

Return Type

```
int
```

Example

```
int Result = FtpPutFile(0, "/LOGS/Report.txt", "/Reports/Report.txt", 1);
```

GetAlarmTag(*index*)

ARGUMENT	TYPE	DESCRIPTION
`index`	`int`	The tag index number.

Description

Returns an integer bit mask representing the tag alarms' state for the tag identified with *index*. Bit 0 (i.e. the bit with a value of 0x01) represents the state of Alarm 1 and bit 1 (i.e. the bit with a value of 0x02) represents the state of Alarm 2. The tag index can be found from the tag name using the `FindTagIndex()` function, or by looking up the tag in the configuration software.

Function Type

This function is *passive*.

Return Type

```
int
```

Example

```
int AlarmsInTag = GetAlarmTag(12);
```

GetAutoCopyStatusCode(void)

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 and their meanings follow:

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
```

Example

```
int Code = GetAutoCopyStatusCode();
```

GetAutoCopyStatusText(void)

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
```

Example

```
cstring Text = GetAutoCopyStatusText();
```

GetBatch(void)

ARGUMENT	TYPE	DESCRIPTION
`none`		

Description

Returns the name of the current batch.

Function Type

This function is *passive*.

Return Type

```
cstring
```

Example

```
cstring Name = 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

Returns the value of the parameter number *param* for a Banner camera connected on the Crimson® device. The argument *camera* is the device number displayed in the Crimson 3.2 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

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

GetCurrentUserName(void)

ARGUMENT	TYPE	DESCRIPTION
`none`		

Description

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

Function Type

This function is *passive*.

Return Type

```
cstring
```

Example

```
cstring User = GetCurrentUserName();
```

GetCurrentUserRealName(void)

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
```

Example

```
cstring Real = GetCurrentUserRealName();
```

GetCurrentUserRights(void)

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
```

Example

```
int Rights = GetCurrentUserRights();
```

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

```
int 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, as follows:

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
```

Example

```
int Status = GetDeviceStatus(1);
```

GetDiskFreeBytes(*drive*)

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

Description

Returns the number of free memory kilobytes on the memory card. Note that it takes a considerable amount of effort to calculate the available space as many read operations must be performed. Do not, therefore, use this function in an expression that is called too often, by placing it on a display page or running it in response to the tick event. Rather, call it in response to a page's `OnSelect` event and store the value in a tag for later display.

FlexEdge® Products – Returns the number of free memory kilobytes on the drive specified. Drive 'C' refers to the internal memory, while Drive 'D' refers to the USB memory stick and Drive 'E' refers to the Memory card.

Function Type

This function is *passive*.

Return Type

```
int
```

Example

```
int FreeMemory = GetDiskFreeBytes('c');
```

GetDiskFreePercent(*drive*)

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

Description

Returns the percentage of free memory space on the memory card. Note that it takes a considerable amount of effort to calculate the available space as many read operations must be performed. Do not, therefore, use this function in an expression that is called too often, by placing it on a display page or running it in response to the tick event. Rather, call it in response to a page's `OnSelect` event and store the value in a tag for later display.

FlexEdge® Products – Returns the percentage of free memory on the drive specified. Drive 'C' refers to the internal memory, while Drive 'D' refers to the USB memory stick and Drive 'E' refers to the Memory card.

Function Type

This function is *passive*.

Return Type

```
int
```

Example

```
int FreeMemory = GetDiskFreePercent('c');
```

GetDiskSizeBytes(*drive*)

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

Description

Returns the size in kilobytes of the memory card. Note that it takes a considerable amount of effort to calculate the available space as many read operations must be performed. Do not, therefore, use this function in an expression that is called too often, by placing it on a display page or running it in response to the tick event. Rather, call it in response to a page's `OnSelect` event and store the value in a tag for later display.

FlexEdge® Products – Returns the size in kilobytes on the drive specified. Drive 'C' refers to the internal memory, while Drive 'D' refers to the USB memory stick and Drive 'E' refers to the Memory card.

Function Type

This function is *passive*.

Return Type

```
int
```

Example

```
int DiskSize = GetDiskSizeBytes('c');
```

GetDriveStatus(*drive*)

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

Description

Returns the status of the specified drive as an integer, as follows:

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.
2	Checking	The HMI is checking the status of the card. This state occurs when a card is first inserted into the HMI.
3	Formatting	The HMI is formatting the card. This state occurs when a format operation is requested by the programming PC.
4	Locked	The Crimson® device 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 Crimson device or Windows.

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

FlexEdge® Products - Drive ‘C’ refers to the internal memory, while Drive ‘D’ refers to the USB memory stick and Drive ‘E’ refers to the Memory card.

Function Type

This function is *passive*.

Return Type

```
int
```

Example

```
int Status = GetDriveStatus(‘c’);
```

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
```

Example

```
int hFile = OpenFile(“MyFile.txt”);

if( hFile ) {

    int n = GetFileByte(hFile);

    if( n == -1 ) {

          return;
    }
}
```

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 *data* 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`

Example

```
int hFile = OpenFile("MyFile.txt");

if( hFile ) {

    int n = GetFileData(hFile, Tag1[0], 10);
}
```

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. The index can be found from the tag label using the function `FindTagIndex()` or by looking it up in the Crimson® configuration tool. This function works with any type of data tag.

Function Type

This function is *passive*.

Return Type

```
cstring
```

Example

```
cstring Text = GetFormattedTag(10);
```

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.

Function Type

This function is *passive*.

Return Type

```
cstring
```

Example

```
cstring Status = 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 Crimson® configuration tool, or from the tag label using the function `FindTagIndex()`. This function will work only if the tag is an integer.

Function Type

This function is *active*.

Return Type

```
int
```

Example

```
int Value = GetIntTag(10);
```

GetLanguage(void)

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
```

Example

```
int Lang = GetLanguage();
```

GetLastEventText(*all*)

ARGUMENT	TYPE	DESCRIPTION
`all`	`int`	Set to true to also include alarm events.

Description

Returns the label of the last event captured by the event log. If the *all* parameter is set to *true*, the definition of event includes those automatically generated by the alarm system, rather than just those generated by the Event Logger.

Function Type

This function is *passive*.

Return Type

```
cstring
```

Example

```
cstring Text = GetLastEventText(true);
```

GetLastEventTime(*all*)

ARGUMENT	TYPE	DESCRIPTION
`All`	`int`	Set to true to also include alarm events.

Description

Returns the time at which the last event capture by the event logger occurred. The value can be displayed in a human-readable form using a field that has the Time and Date format type. If the *all* parameter is set to true, the definition of event includes events automatically generated by the alarm system, rather than just those generated by the Event Logger.

Function Type

This function is *passive*.

Return Type

```
int
```

Example

```
int Time = GetLastEventTime(true);
```

GetLastEventType(*all*)

ARGUMENT	TYPE	DESCRIPTION
`all`	`int`	Set to true to also include alarm events.

Description

Returns a string indicating the type of the last event captured by the event logging system. If the all parameter is set to true, the definition of event includes events automatically generated by the alarm system, rather than just those generated by the Event Logger.

Function Type

This function is *passive*.

Return Type

```
Cstring
```

Example

```
cstring Type = GetLastEventType(true);
```

GetLastSQLSyncStatus(void)

ARGUMENT	TYPE	DESCRIPTION
`none`		

Description

Returns the status from the last time that the SQL Sync Service attempted to synchronize data logs with a SQL server.

VALUE	STATE	DESCRIPTION
0	Pending	The status of the SQL Sync service is in an indeterminate state. This is because the service has yet to run, the service has not yet completed synchronizing with the SQL server, or the service is disabled.
1	Success	The service successfully synchronized with the SQL server.
2	Failure	The service failed to synchronize with the SQL server.

Function Type

This function is *passive*.

Return Type

```
int
```

Example

```
int Status = GetLastSQLSyncStatus();
```

GetLastSQLSyncTime(*Request*)

ARGUMENT	TYPE	DESCRIPTION
`Request`	`int`	The specific time to retrieve.

Description

Returns the last time that the SQL Sync Service synchronized with a SQL server since the system started up. The returned value is suitable for formatting using the Crimson® time manipulation functions. Until the service attempts to synchronize, all three request types return 0, which represents January 1, 1997.

VALUE	REQUEST TYPE	DESCRIPTION
0	Last Start Time	Get the last time that the SQL Sync Service began synchronizing with a SQL server.
1	Last Success Time	Get the last time that the service successfully synchronized with a SQL server.
2	Last Failure Time	Get the last time that the service failed to synchronize with a SQL server.

Function Type

This function is *passive*.

Return Type

```
int
```

Examples

```
int LastStartTime = GetLastSQLSyncTime(0);
int LastSuccessTime = GetLastSQLSyncTime(1);
int LastFailTime = GetLastSQLSyncTime(2);
```

GetLicenseState(*type*)

ARGUMENT	TYPE	DESCRIPTION
`type`	`int`	Index of licenses installed on device.

Description

Returns a string describing the current state of a Crimson® license module installed on the device.

Function Type

This function is *active*.

Return Type

```
cstring
```

Example

```
cstring State = GetLicenseState(0x1012);
```

GetLocationPropery(*index, property*)

ARGUMENT	TYPE	DESCRIPTION
`index`	`int`	The offset to the modem
`property`	`cstring`	The property to retrieve

Description

Retrieves a location property from the given modem using the property name as a string. The property name is not case sensitive. The possible options for the property name are given in the table below.

PROPERTY	DESCRIPTION
`fix`	GPS fix state
`seq`	The sequence number
`lat`	Latitude
`long`	Longitude
`alt`	Altitude

Function Type

The function is *passive*.

Return Type

```
float
```

Example

```
float Latitude = GetLocationProperty(0, “lat”);
```

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
```

Example

```
cstring Text = GetModelName(1);
```

GetModemProperty(*index, property*)

ARGUMENT	TYPE	DESCRIPTION
`index`	`int`	The offset to the modem
`property`	`cstring`	The property to retrieve

Description

Retrieves a property from the given modem using the property name as a string. The property name is not case sensitive. The possible options for the property name are given in the table below.

PROPERTY	DESCRIPTION
`online`	Returns TRUE if the modem is online, else FALSE
`register`	Returns TRUE if registered, else FALSE
`roam`	Returns TRUE if the roaming, else FALSE
`slot`	The SIM slot
`service`	The service of the modem
`carrier`	The carrier of the connected network
`iccid`	The ICCID from the SIM card
`imsi`	The IMSI from the SIM card
`addr`	The IP address of the modem
`imei`	The IMEI of the modem
`state`	The state of the network connection
`network`	The name of the network
`model`	The model of the modem
`version`	The version of the modem firmware
`signal`	The signal strength, where a number closer to zero indicates a stronger connection
`time`	The time since the connection was established

Function Type

The function is *passive*.

Return Type

```
cstring
```

Example

```
cstring Signal = GetModemProperty(0, “signal”);
```

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

```
int 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

```
cstring Gate = GetNetGate(0);
```

GetNetId(*port*)

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

Description

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

INDEX	DESCRIPTION
0	`Returns address of first or only port configured.`
1	`Returns address of port 1.`
2	`Returns address of port 2.`

Function Type

This function is *passive*.

Return Type

```
cstring
```

Example

```
cstring MAC = GetNetId(1);
```

GetNetIp(*port*)

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

Description

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

Function Type

This function is *passive*.

Return Type

```
cstring
```

Example

```
cstring Addr = 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

```
cstring Mask = GetNetMask(0);
```

GetNow(void)

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

```
int t = GetNow();
```

GetNowDate(void)

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

```
int d = GetNowDate();
```

GetNowTime(void)

ARGUMENT	TYPE	DESCRIPTION
`none`		

Description

Returns the time of day in terms of seconds.

Function Type

This function is *passive*.

Return Type

```
int
```

Example

```
int t = GetNowTime();
```

GetPersonalityFloat(*name*)

ARGUMENT	TYPE	DESCRIPTION
`name`	`cstring`	The Key Name of the Device Personality.

Description

Gets the specified Device Personality Key Name as a real number.

Function Type

This function is *active*.

Return Type

```
float
```

Example

```
float Value = GetPersonalityFloat("tags.daily.average");
```

GetPersonalityInt (name)

ARGUMENT	TYPE	DESCRIPTION
`name`	`cstring`	The Key Name of the Device Personality.

Description

Gets the specified Device Personality Key Name as an integer.

Function Type

This function is *active*.

Return Type

```
int
```

Example

```
int Value = GetPersonalityInt("net.faces.eth1.sendmss");
```

GetPersonalityIp(*name*)

ARGUMENT	TYPE	DESCRIPTION
`name`	`cstring`	The Key Name of the Device Personality.

Description

Gets the specified Device Personality Key Name as an IP address.

Function Type

This function is *active*.

Return Type

```
int
```

Example

```
int Value = GetPersonalityIp("net.faces.eth1.address");
```

GetPersonalityString(*name*)

ARGUMENT	TYPE	DESCRIPTION
`name`	`cstring`	The Key Name of the Device Personality.

Description

Gets the specified Device Personality Key Name as a string.

Function Type

This function is *active*.

Return Type

```
cstring
```

Example

```
cstring Value = GetPersonalityString("services.smtp.user");
```

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 the serial port identified by *port*. The port number starts from the programming port with value 1. The table below shows the various *param* settings and associated return values. The port number is displayed in Crimson®'s status bar of the Communications category when the port is selected.

VALUE	PARAMETER	DESCRIPTION OF RETURN VALUE
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	0None, 1Odd, 2Even.
5	Physical Mode	0RS-232, 1RS-422 Master, 2RS-422 Slave, 3RS-485.

Function Type

This function is *passive*.

Return Type

```
int
```

Example

```
int Port2Parity = GetPortConfig(2, 4);
```

GetQueryStatus(*query*)

ARGUMENT	TYPE	DESCRIPTION
`query`	`string`	The name of the query, as found on the SQL Manager tree.

Description

Retrieves the last time that the given query attempted.

VALUE	STATE	DESCRIPTION
0	`Pending`	The status of the query is in an undeterminate state. Either the query has yet to execute or it is actively executing.
1	`Success`	The query successfully executed and retrieved data for all columns.
2	`Partial Data`	The query retrieved some data, but not for all the columns. Verify that the configured column type matches the actual type of the column in the SQL database.
3	`Failure`	The query failed to retrieve any data.

Function Type

This function is *passive*.

Return Type

`int`

Example

```
Int Status := GetQueryStatus("Query1");
```

GetQueryTime(*query*)

ARGUMENT	TYPE	DESCRIPTION
`query`	`string`	The name of the query, as found on the SQL Manager tree.

Description

Retrieves the last time that the given query attempted to execute. If the query has not executed as expected, verify that the SQL Manager is connected and that the network is configured properly.

Function Type

This function is *passive*.

Return Type

```
int
```

Example

```
int Time := GetQueryTime("Query1");
```

GetRealTag(*index*)

ARGUMENT	TYPE	DESCRIPTION
`index`	`int`	Tag index number.

Description

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

Function Type

This function is *active*.

Return Type

```
float
```

Example

```
float Value = GetRealTag(10);
```

GetRestartCode(*index*)

ARGUMENT	TYPE	DESCRIPTION
`index`	`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`

Example

```
cstring Value = GetRestartCode(0);
```

GetRestartInfo(*index*)

ARGUMENT	TYPE	DESCRIPTION
`Index`	`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
```

Example

```
cstring Value = GetRestartInfo(0);
```

GetRestartText(*index*)

ARGUMENT	TYPE	DESCRIPTION
`Index`	`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
```

Example

```
cstring Value = GetRestartText(0);
```

GetRestartTime(*index*)

ARGUMENT	TYPE	DESCRIPTION
`Index`	`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
```

Example

```
int Time = GetRestartTime(0);
```

GetSQLConnectionStatus(void)

ARGUMENT	TYPE	DESCRIPTION
`none`		

Description

Retrieves the status of the SQL Manager connection.

VALUE	STATE	DESCRIPTION
`0`	`Pending`	The status of the SQL Manager connection is in an indeterminate state. This is because the manager has yet to run, the manager is actively querying data, or the manager is disabled.
`1`	`Success`	The manager successfully opened a connection with the SQL server.
`2`	`Failure`	The manager failed to open a connection with the remote SQL server. Verify that the network is configured correctly and that the login credentials are correct.

Function Type

This function is *passive*.

Return Type

```
int
```

Example

```
Int Status = GetSQLConnectionStatus();
```

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 work only if the tag is a string tag.

Function Type

This function is *active*.

Return Type

```
cstring
```

Example

```
cstring Value = GetStringTag(10);
```

GetTagLabel(*index*)

ARGUMENT	TYPE	DESCRIPTION
`index`	`int`	Tag index number.

Description

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

Function Type

This function is *passive*.

Return Type

```
cstring
```

Example

```
cstring Label = GetTagLabel(10);
```

GetSystemIo(*prop*)

ARGUMENT	TYPE	DESCRIPTION
`prop`	`cstring`	IO Point.

Description

Returns the value of the IO point specified by *prop*.

Property	DESCRIPTION
`“DI”`	Digital Input.
`“DO”`	Digital Output.
`“AI”`	Analog Input.
`“VI”`	Power Supply Voltage.

Function Type

This function is *passive*.

Return Type

`int`

Example

```
int DI = GetSystemIo(“DI”);
int DO = GetSystemIo(“DO”);
int AI = GetSystemIo(“AI”);
int VI = GetSystemIo(“VI”);
```

GetTcpSocketPeer(*port, interface*)

ARGUMENT	TYPE	DESCRIPTION
`port`	`int`	The tcp port.
`interface`	`int`	The interface.

Description

Returns the Tcp socket peer.

Function Type

This function is *passive*.

Return Type

```
int
```

Example

```
int Peer = GetTcpSocketPeer(0, 1);
```

GetUpDownData(*data*, *limit*)

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

Description

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

```
int 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

```
int Delta = GetUpDownStep(DispCount, 100);
```

GetVersionInfo(*code*)

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

Description

Returns information about the various version numbers, as follows:

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
```

Example

```
int Boot = GetVersionInfo(1);
int App = GetVersionInfo(2);
int Config = GetVersionInfo(3);
```

GetWebParamHex(*param*)

ARGUMENT	TYPE	DESCRIPTION
`param`	`cstring`	The parameter to retrieve.

Description

Returns a parameter passed to custom web page through the URL query string. The parameter is interpreted as a hexadecimal integer and then returned. This function is typically used in code invokes via the embedded tag syntax of the custom website.

Function Type

This function is *passive*.

Return Type

```
int
```

Example

```
int Value1 = GetWebParamHex("Count");
```

GetWebParamInt(*param*)

ARGUMENT	TYPE	DESCRIPTION
`param`	`cstring`	The parameter to retrieve.

Description

Returns a parameter passed to custom web page through the URL query string. The parameter is interpreted as a decimal integer and then returned. This function is typically used in code invokes via the embedded tag syntax of the custom website.

Function Type

This function is *passive*.

Return Type

```
int
```

Example

```
int Value1 = GetWebParamInt("Count");
```

GetWebParamStr(*param*)

ARGUMENT	TYPE	DESCRIPTION
`param`	`cstring`	The parameter to retrieve.

Description

Returns a parameter passed to custom web page through the URL query string. The parameter is returned as a string containing the characters passed in the parameter.

Function Type

This function is *passive*.

Return Type

`cstring`

Example

```
cstring Value = GetWebParamStr("Test");
```

GetWifiProperty(*index, property*)

ARGUMENT	TYPE	DESCRIPTION
`index`	`int`	The offset to the Wi-Fi device
`property`	`cstring`	The property to retrieve

Description

Retrieves a property from the given Wi-Fi device using the property name as a string. The property name is not case sensitive. The possible options for the property name are given in the table below.

PROPERTY	DESCRIPTION
`online`	"TRUE" if the Wi-Fi device is online, else "FALSE"
`apmode`	"AP" if the Wi-Fi device is configured as access point, else "STATION" when configured in station (client) mode
`channel`	The Wi-Fi channel that the device is connected to
`peermac`	The MAC address of the Wi-Fi device
`state`	The current state of the connection
`network`	The currently connected network
`model`	The model of the Wi-Fi device
`version`	The firmware version of the Wi-Fi device
`signal`	The signal given as an RSSI reading. Values closer to zero indicate a stronger signal
`time`	The time since the connection was established

Function Type

The function is *passive*.

Return Type

```
cstring
```

Example

```
cstring Signal = GetWifiProperty(0, "signal");
```

GotoNext()

ARGUMENT	TYPE	DESCRIPTION
`none`		

Description

Causes the Crimson® device to move forward 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 Crimson® device'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 last page to be shown on the Crimson® device's display. The page is extracted from a history buffer, so "previous" refers to the previously displayed page, not the previous page in the Display Page navigation window.

Function Type

This function is *active*.

Return Type

This function does not return a value.

Example

```
GotoPrevious();
```

GreaterEqR64(*arg1*, *arg2*)

ARGUMENT	TYPE	DESCRIPTION
`arg1`	`int`	The value to be compared.
`arg2`	`int`	The value to compare to.

Description

Compares the value of *arg1* to *arg2* using 64-bit double precision floating point math and returns 1 if *arg1* is greater than or equal to *arg2*, and 0 otherwise. This is the double precision equivalent of `a >= b`. The input operands *arg1* and *arg2* 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

```
int Result = GreaterEqR64(a[0], b[0]);
```

GreaterR64(*arg1*, *arg2*)

ARGUMENT	TYPE	DESCRIPTION
`arg1`	`int`	The value to be compared.
`arg2`	`int`	The value to compare to.

Description

Compares the value of *arg1* to *arg2* using 64-bit double precision floating point math and returns 1 if *arg1* is greater than *arg2*, 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

```
int 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 contains a bitmask representing the various user defined rights, with bit 0 (i.e., the bit with a value of 0x01) representing User Right 1, bit 1 (i.e., the bit with a value of 0x02) representing User Right 2 and so on. The function is typically used in programs that perform actions that might be subject to security.

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 contains a bitmask representing the various user defined rights, with bit 0 (i.e., the bit with a value of 0x01) representing User Right 1, bit 1 (i.e., the bit with a value of 0x02) representing User Right 2 and so on. The function is typically used in programs that perform actions that might be subject to security.

Function Type

This function is *passive*.

Return Type

```
Int
```

Example

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

HideAllPopups(void)

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.

Example

```
HideAllPopups();
```

HidePopup(void)

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();
```

IntR64(*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

This function does not return a value.

Example

```
IntR64(result[0], tag[0]);
```

IntToR64(*result*, *tag*)

ARGUMENT	TYPE	DESCRIPTION
`result`	`int`	The result.
`tag`	`int`	The value to be converted.

Description

Converts the value stored in *tag* from an integer to a 64-bit double precision number and stores the result in *result*. The tag result should be an entry in an integer array with an extent such that at least two registers can be accessed from that point. 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

This function does not return a value.

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 number 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
```

Example

```
int Result = IsBatchNameValid(“MyBatch”);
```

IsBatteryLow(void)

ARGUMENT	TYPE	DESCRIPTION
`none`		

Description

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

Function Type

This function is *passive*.

Return Type

```
int
```

Example

```
int Result = IsBatteryLow();
```

IsDeviceOnline(*device*)

ARGUMENT	TYPE	DESCRIPTION
`device`	`int`	The index of the device to be checked.

Description

Reports if device identified by *device* is online or not. A device is considered offline after a repeated sequence of communications errors. When a device is in the offline state, it will be polled periodically until it returns online. The device number is displayed in Crimson®'s status bar of the Communications category when the device is selected.

Function Type

This function is *passive*.

Return Type

```
int
```

Example

```
int Result = IsDeviceOnline(1);
```

IsLoggingActive(void)

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
```

Example

```
int Result = IsLoggingActive();
```

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. The port number is displayed in Crimson®'s status bar of the Communications category when the port is selected.

Function Type

This function is *passive*.

Return Type

```
int
```

Example

```
int Result = IsPortRemote(1);
```

IsTcpSocketActive(*port, interface*)

ARGUMENT	TYPE	DESCRIPTION
`port`	`int`	The tcp port.
`interface`	`int`	The tcp interface.

Description

Indicates whether the TCP socket is active.

Function Type

This function is *passive*.

Return Type

`int`

Example

```
int State = IsTcpSocketActive(0, 1);
```

IsSQLSyncRunning(void)

ARGUMENT	TYPE	DESCRIPTION
`none`		

Description

Returns whether the SQL Sync Service is currently attempting to synchronize with an SQL Server.

VALUE	STATE	DESCRIPTION
0	Not Running	The SQL Sync Service is not synchronizing with an SQL server.
1	Running	The service is currently synchronizing with an SQL server.

Function Type

This function is *passive*.

Return Type

```
int
```

Example

```
int IsRunning = IsSQLSyncRunning();
```

IsWriteQueueEmpty(*dev*)

ARGUMENT	TYPE	DESCRIPTION
`dev`	`int`	The device number for which to get the queue state.

Description

Returns the state of the write 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. Note that if a communication error occurs while the write queue is not empty, or if data is written during a such an error, the queue will be emptied but the data to be written will remain pending. The pending data will be written once the communication error is cleared. Therefore, this function cannot be used to reliably determine if a device has pending writes when communication errors are present on the device.

Function Type

This function is *passive*.

Return Type

```
int
```

Example

```
int QueueEmpty = IsWriteQueueEmpty(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; returns *true* if the function is successful or *false* if the function fails.

Function Type

This function is *active*.

Return Type

```
int
```

Example

```
int Result = KillDirectory("MyDirectory");
```

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

```
cstring Area = Left("717-555-5555", 3);
```

Len(*string*)

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

Description

Returns the number of characters in *string*, excluding the null character.

Function Type

This function is *passive*.

Return Type

```
int
```

Example

```
int Size = Len(Input);
```

LessEqR64(*arg1, arg2*)

ARGUMENT	TYPE	DESCRIPTION
`arg1`	`int`	The value to be compared.
`arg2`	`int`	The value to compare to.

Description

Compares the value of *arg1* to *arg2* using 64-bit double precision floating point math and returns 1 if *arg1* is less than or equal to *arg2*, and 0 otherwise. This is the double precision equivalent of `a <= b`. The input operands *arg1* and *arg2* 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

```
int Result = LessEqR64(a[0], b[0]);
```

LessR64(*arg1, arg2*)

ARGUMENT	TYPE	DESCRIPTION
`arg1`	`int`	The value to be compared.
`arg2`	`int`	The value to compare to.

Description

Compares the value of *arg1* to *arg2* using 64-bit double precision floating point math and returns 1 if *arg1* is less than *arg2*, 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

```
int Result = 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.

Description

Loads the inspection file from the Crimson® device memory card to the camera memory. The number to be placed in the *port* argument is the port number to which the driver is bound. 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 memory card. This function will return *true* if the transfer is successful, *false* otherwise. The camera device number is displayed in Crimson's status bar of the Communications category when the device is selected.

Note that this function is best called in a user program that executes in the background, so the Crimson device has sufficient time to access the memory card.

Function Type

This function is *active*.

Return Type

```
int
```

Example

```
int Result = LoadCameraSetup(4, 0, 1, “in0.isp”);
```

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 1 is used to save and load the user list, complete with usernames, 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`

Example

```
int Result = LoadSecurityDatabase(0, “database”);
```

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

```
float Data = 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

```
float Data = log10(5.0);
```

log10R64(*result, tag*)

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

Description

Calculates the base-10 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 64bit 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

This function does not return a value.

Example

```
log10R64(result[0], tag1[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.

Example

```
LogBatchComment(0, “My Batch Comment”);
```

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.

Example

```
LogBatchHeader(0, “My Batch Header”);
```

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

This function does not return a value.

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.

Example

```
LogHeader(1, "Start of Shift");
```

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

This function does not return a value.

Example

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

LogSave(void)

ARGUMENT	TYPE	DESCRIPTION
`none`		

Description

Forces the data logger to save to the memory card. Note that this function should **not** be called periodically. It is intended only for punctual use. An overuse of this function may result in memory card damage and loss of data.

Function Type

This function is *active*.

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 have a different data type from that expected by the communications driver.

Function Type

This function is *passive*.

Return Type

```
float
```

Example

```
float 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 have a different data type from that expected by the communications driver.

Function Type

This function is *passive*.

Return Type

```
int
```

Example

```
int n = MakeInt(fp);
```

Max(*arg1*, arg2)

ARGUMENT	TYPE	DESCRIPTION
`arg1`	`int / float`	The first value to be compared.
`arg2`	`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 argument type.

Example

```
int 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

This function does not return a value.

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

```
int 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

```
int Value = 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

```
cstring Exchange = Mid("717-555-5555", 4, 3)
```

Min(*arg1*, arg2)

ARGUMENT	TYPE	DESCRIPTION
`arg1`	`int / float`	The first value to be compared.
`arg2`	`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 arguments type.

Example

```
int 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

This function does not return a value.

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 *passive*.

Return Type

```
int
```

Example

```
int 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

This function does not return a value.

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

```
int Result = ModU32(tag1, tag2);
```

MountCompactFlash(*enable*)

ARGUMENT	TYPE	DESCRIPTION
`enable`	`int`	The desired mount or dismount state.

Description

Mounts or dismounts the memory card as a drive accessible from Windows Explorer. If *enable* is set to 1, then the card will be mounted. If *enable* is set to 0, then the card will be dismounted. The device will restart itself automatically after calling this function. This function exposes the same functionality as the Mount Flash and Dismount Flash options found under the Link menu in the Crimson® 3.2 software.

Function Type

This function is *active*.

Return Type

This function does not return a value.

Example

```
MountCompactFlash(0);
```

Note: This function is deprecated in Crimson 3.2.

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, as follows:

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`

Example

```
int Result = MoveFiles("target", "source", 1);
```

MulDiv(*arg1*, arg2, arg3)

ARGUMENT	TYPE	DESCRIPTION
`arg1`	`int`	The first value.
`arg2`	`int`	The second value.
`arg3`	`int`	The third value.

Description

Returns (`arg1*arg2)/arg3`. Intermediate math is done with 64-bit integers to avoid overflows.

Function Type

This function is *passive*.

Return Type

`int`

Example

```
int Result = 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 under `AddR64()`.

Function Type

This function is *active*.

Return Type

This function does not return a value.

Example

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

MulU32(*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

```
int Result = MulU32(tag1, tag2);
```

MuteSiren(void)

ARGUMENT	TYPE	DESCRIPTION
`none`		

Description

Turns off the Crimson® device's internal siren.

Function Type

This function is *active*.

Return Type

This function does not return a value.

Example

```
MuteSiren();
```

NetworkPing(*address*, *timeout*)

ARGUMENT	TYPE	DESCRIPTION
`address`	`int`	The target address to send a ping request to.
`timeout`	`int`	The time in millisecond to wait for a response.

Description

Sends an ICMP echo request (commonly referred to as a ping) to the specified IP address and waits for the given timeout period for a reply. The timeout parameter should be given in milliseconds. If a valid response is received within the timeout period, then the function will return 1. If no response is received within the timeout period, or this function is called while the Ethernet port is disabled, then the function will return 0.

Function Type

This function is *passive*.

Return Type

`int`

Examples

```
int IP = TextToAddr("192.168.1.100");
Result = NetworkPing(IP, 5000);

Int IP = ResolveDNS("redlion.net");
Result = NetworkPing(IP, 5000);
```

NewBatch(*name*)

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

Description

Starts a batch called *name*. The Crimson® 3.2 filing system now supports both FAT16 and FAT32. If the memory card has been formatted to use FAT32, then long filenames may be used. If the memory card has been formatted using FAT16, then long filenames are not supported. Restarting a batch already on the memory card will append the data. If a new batch exceeds the maximum number of batches to be kept, the oldest batch will be deleted. If *name* is empty, the function is equivalent to `EndBatch()`. Batch status is retained during a power cycle. Note that 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(void)

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(*arg1*, arg2)

ARGUMENT	TYPE	DESCRIPTION
`arg1`	`int`	The first value to be compared.
`arg2`	`int`	The second value to be compared.

Description

Compares the value of *arg1* against *arg2* using 64-bit double precision floating point math and returns 1 if *arg1* is not equal to *arg2*, and 0 otherwise. This is the double precision equivalent of `arg1!=arg2`. Note that comparing floating point values for equality can be error prone due to rounding errors. The input operands *arg1* and *arg2* 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

```
int Result = 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 memory card. This function is restricted to a maximum of four open files at any given time. The memory card cannot be unmounted while a file is open. The Crimson® 3.2 filing system now supports both FAT16 and FAT32. If the memory card has been formatted to use FAT32, then long filenames may be used. If the memory card has been formatted using FAT16, then long filenames are not supported. Note 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 within the Crimson 3.2 Software Guide. 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.

When using this function with a FlexEdge® device the file can be opened from the internal memory, Memory card, or Memory stick. By default, the function will open a file from the internal memory. If you wish to open a file elsewhere you must specify the drive letter.

Function Type

This function is *active*.

Return Type

```
int
```

Example

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

int hFile = OpenFile("D:/LOGS/LOG1/01010101.csv", 0);  // Memory Stick  *FlexEdge Only*
int hFile = OpenFile("E:/LOGS/LOG1/01010101.csv", 0);  // SD Card  *FlexEdge Only)
```

Pi(void)

ARGUMENT	TYPE	DESCRIPTION
`None`		

Description

Returns *pi* as a floating-point number.

Function Type

This function is *passive*.

Return Type

```
float
```

Example

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

PlayRTTTL(*tune*)

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

Description

Plays a tune using the Crimson® device'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,16g,16f,16g,16g#.,16g#,16g,16g#,16
g,16f.,1 6f,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

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

PortClose(*port*)

ARGUMENT	TYPE	DESCRIPTION
`port`	`int`	The raw port to be closed.

Description

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. The port number is displayed in Crimson®'s status bar of the Communications category when the port is selected.

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 state of the CTS line on the serial port indicated by *port*. The port must be configured to use a raw driver. The port number is displayed in Crimson®'s status bar of the Communications category when the port is selected.

Function Type

This function is *active*.

Return Type

```
int
```

Example

```
int CtsState = PortGetCTS(2);
```

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 is met…

- *end* is non-zero and a character matching `end` is received.
- *timeout* is non-zero, and that period passes with no characters 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. In the event of a timeout, the received characters will only be returned if both the end and length parameters are zero. If either the end or length parameters are non-zero (or if both are non-zero), then the function will return an empty string. This function is used together with raw port drivers to implement custom protocols using Crimson®'s programming language. The port number is displayed in Crimson's status bar of the Communications category when the port is selected.

Function Type

This function is *active*.

Return Type

```
cstring
```

Example

```
cstring 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 immediately return. The port driver will handle handshaking and control of transmitter enable lines, as required. The port number is displayed in Crimson®'s status bar of the Communications category when the port is selected.

Function Type

This function is *active*.

Return Type

This function does not return a value.

Example

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

PortPrintEx(*port, string*)

ARGUMENT	TYPE	DESCRIPTION
`port`	`int`	The raw port to be written to.
`string`	`string`	The string to be printed on the specified port.

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 immediately return. The port driver will handle handshaking and control of transmitter enable lines, as required. Sending data over a TCP/IP raw port will attempt to send the data in a single packet if possible or use as few packets as possible. The port number is displayed in Crimson®'s status bar of the Communications category when the port is selected.

Function Type

This function is *active*.

Return Type

This function does not return a value.

Example

```
PortPrintEx(4, "ABCDEFGHIJKLMNOPQRSTUVWXYZ");
```

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 indicated by *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. The port number is displayed in Crimson's status bar of the Communications category when the port is selected.

Function Type

This function is *active*.

Return Type

```
int
```

Example

```
int Data = PortRead(1, 100);
```

PortSendData(port, data, count)

ARGUMENT	TYPE	DESCRIPTION
`port`	`int`	The raw port to be written to.
`data`	`int`	The first element of the array of data to be sent.
`count`	`int`	The number of elements to send.

Description

Transmits *count* number of elements of the array starting at *data* 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 immediately return. The port driver will handle handshaking and control of transmitter enable lines, as required. Data sent over a TCP/IP raw port will be sent in a single packet if possible. Each element in the *data* input array should represent one byte's worth of the desired data to be sent. The elements in the array can be any value from 0 to 255, inclusive. This function provides an alternative to the text based `PortPrint()` function, allowing for binary data to be sent. The port number is displayed in Crimson®'s status bar of the Communications category when the port is selected.

Function Type

This function is *active*.

Return Type

This function does not return a value.

Example

```
PortSendData(4, Data[0], 16);
```

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 state of the RTS line on the serial port specified by *port*. The port must be configured to use a raw driver and be on one of the serial ports. The *state* argument can take values 0 or 1, only. The port number is displayed in Crimson®'s status bar of the Communications category when the port is selected.

Function Type

This function is *active*.

Return Type

This function does not return a value.

Example

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

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. The port number is displayed in Crimson®'s status bar of the Communications category when the port is selected.

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

This function does not return a value.

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 of *power*.

Function Type

This function is *passive*.

Return Type

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

Example

```
int 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 the power of *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

This function does not return a value.

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 one to create an 8 bits-per-pixel bitmap, while a value of zero will create a 16 bits-per-pixel bitmap. The latter value will produce much larger files, as 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
```

Example

```
int Result = PrintScreenToFile(“capture”, “”, 1);
```

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 indicated by *data* to the specified *file*. Returns 1 for success and -1 for failure.

Function Type

This function is *active*.

Return Type

`int`

Example

```
int hFile = OpenFile("MyFile");
if( hFile ) {

    int Result = PutFileByte(hFile, 100);
}
```

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

Writes 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`

Example

```
int hFile = OpenFile("MyFile");
if( hFile ) {

   int Result = PutFileData(hFile, tag1[0], 10);
}
```

R64ToInt(*value*)

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

Description

Converts the 64-bit double precision floating point value stored in *value* as an array with extent 2 to a signed integer and returns the result. The array *value* will typically 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 *value* must be able to be represented by 32-bit integer for this conversion to be successful. See the entry for `AddR64()` for more information.

Function Type

This function is *passive*.

Return Type

```
int
```

Example

```
int Result = R64ToInt(x[0]);
```

R64ToReal(*value*)

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

Description

Converts the 64-bit double precision floating point value stored in *value* as an array with extent 2 in *x* to a 32-bit floating point number and returns the result. Typically, the array *value* will contain a 64-bit floating point value obtained as a result from one of the 64-bit math functions provided. Note that the number represented by the array *value* must be able to be represented by 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

```
float 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

```
int 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 of up to 512 characters in length from the specified file. This function does not look for a line feed or carriage return, but instead reads a specified number of bytes. The string returned by `ReadFile()` will include as many lines as required to reach the number of characters to be read. Line feed and carriage return characters will be part of the returned string.

Function Type

This function is *active*.

Return Type

```
cstring
```

Example

```
cstring 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

```
cstring 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 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

This function does not return a value.

Example

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

RenameFile(*file, name*)

ARGUMENT	TYPE	DESCRIPTION
`file`	`int`	The file handle as returned by `OpenFile`.
`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 Crimson® 3.2 filing system now supports both FAT16 and FAT32. If the memory card has been formatted to use FAT32, then long filenames may be used. If the memory card has been formatted using FAT16, then long filenames are not supported.

Function Type

This function is *active*.

Return Type

`int`

Example

```
int hFile = OpenFile("OldName.txt");
int Result = RenameFile(hFile , "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

```
int 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

```
cstring Local = Right("717-555-5555", 8);
```

RShU32(*arg1, arg2*)

ARGUMENT	TYPE	DESCRIPTION
`arg1`	`int`	The value to be shifted.
`arg2`	`int`	The amount to shift by.

Description

Returns the value *arg1* shifted *arg2* bits to the right in an unsigned context. This is the unsigned equivalent to *arg1* >> *arg2*.

Function Type

This function is *passive*.

Return Type

```
int
```

Example

```
int Shifted = RShU32(tag1, tag2);
```

RunAllQueries(void)

ARGUMENT	TYPE	DESCRIPTION
`none`		

Description

Instructs the SQL Manager to run all configured queries.

Function Type

This function is *active*.

Return Type

This function does not return a value.

Example

```
RunAllQueries();
```

RunQuery(*query*)

ARGUMENT	TYPE	DESCRIPTION
`query`	`string`	The name of the query, as found on the SQL Manager tree.

Description

Instructs the SQL Manager to run the specified *query*.

Function Type

This function is *active*.

Return Type

This function does not return a value.

Example

```
RunQuery("Query1");
```

RxCAN(*port, data, id*)

ARGUMENT	TYPE	DESCRIPTION
`port`	`int`	The raw 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()` on CAN port indicated by *port*. The first four bytes of the received message will be packed (big endian) in the array element indicated by *data* 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 or 0 upon failure.

Function Type

This function is *active*.

Return Type

`int`

Example

```
int Result = RxCAN(8, Data, 0x12345678);
```

RxCANInit(*port, id, dlc*)

ARGUMENT	TYPE	DESCRIPTION
`port`	`int`	The raw CAN port.
`id`	`int`	29-bit CAN Identifier.
`dlc`	`int`	Data Length Count of 1 – 8 bytes.

Description

Initializes the programmatic transfer of CAN messages via the Raw 29-bit CAN Port driver on a CAN Port. The function will return a value of 1 upon success or a value of 0 upon failure. Calls should be made only after the system has started, and each 29-bit identifier should only be initialized only one time.

Function Type

This function is *active*.

Return Type

`int`

Example

```
int Result = 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.

Description

Saves the inspection file uploaded from the camera on the Crimson® device memory 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 displayed in the Crimson 3.2 status bar when the camera is selected. More than one camera can be connected under a single driver. The argument *index* represents the inspection file number within the camera. The *file* is the path and filename where the inspection file should be saved on memory card. This function will return *true* if the transfer is successful and *false* otherwise. This function should be called in a user program that runs in the background so the HMI has sufficient time to access the memory card.

Function Type

This function is *active*.

Return Type

```
int
```

Example

```
int Result = SaveCameraSetup(4, 0, 1, “\\in0.isp”);
```

SaveConfigFile(*file*)

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

Description

Saves the current boot loader, firmware and database image to a CI3 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

```
int Result = SaveConfigFile(“image.ci3”);
```

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 usernames, 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
```

Example

```
int Result = SaveSecurityDatabase(0 ,"database");
```

Scale(*data, raw1, raw2, eng1, eng2*)

ARGUMENT	TYPE	DESCRIPTION
`data`	`int`	The value to be scaled.
`raw1`	`int`	The minimum raw value stored in `data`.
`raw2`	`int`	The maximum raw value stored in `data`.
`eng1`	`int`	The engineering value corresponding to `r1`.
`eng2`	`int`	The engineering value corresponding to `r2`.

Description

Linearly scales the *data* argument, assuming it to contain values between *raw1* and *raw2*, and producing a return value between *eng1* and *eng2*. 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

```
int 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 Crimson® device 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 Crimson® device 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);
```

SendFileTo(*rcpt, file*)

ARGUMENT	TYPE	DESCRIPTION
`rcpt`	`cstring`	The email of the recipient(s).
`file`	`cstring`	The path of the file to send.

Description

Sends an email from the Crimson® device to the specified recipient(s) with the attached file specified. Elements of a recipient list of more than one item should be separated by a semicolon. 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 *active*.

Return Type

This function does not return a value.

Example

```
SendFileTo("email1@gmail.com;email2@gmail.com", "E:/LOGS/LOG1/260706.csv");
```

SendFileToAck(*rcpt, file, ack*)

ARGUMENT	TYPE	DESCRIPTION
`rcpt`	`cstring`	The email of the recipient(s).
`file`	`cstring`	The path of the file to send.
`ack`	`int`	Indication of success.

Description

Sends an email from the Crimson® device to the specified recipient(s) with the attached *file* specified. Elements of a recipient list containing more than one item should be separated by a semicolon. The function returns immediately, having first added the required email to the system's mail queue. The tag specified as *ack* will increment with completion of each successful action requested. The message will be sent using the appropriate mail transport as configured in the database.

Function Type

This function is *active*.

Return Type

This function does not return a value.

Example

```
  SendFileToAck(“email1@gmail.com;email2@gmail.com”, “E:/LOGS/LOG1/260706.csv”,
Tag1);
```

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 Crimson® device. 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 *active*.

Return Type

This function does not return a value.

Example

```
SendMail(1, “Test Subject Line”, “Test Body Text”);
```

SendMailTo(*rcpt, subject, message*)

ARGUMENT	TYPE	DESCRIPTION
`Rcpt`	`cstring`	The email of the recipient(s).
`subject`	`cstring`	The subject of the email.
`message`	`cstring`	The message in the email.

Description

Sends an email from the Crimson® device to the specified recipient(s) with the specified subject and message. Elements of a recipient list of more than one item should be separated by a semicolon. 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 *active*.

Return Type

This function does not return a value.

Example

```
SendMailTo("email1@gmail.com;email2@gmail.com", "Subject", "Message");
```

SendMailToAck(*rcpt, subject, message, ack*)

ARGUMENT	TYPE	DESCRIPTION
`rcpt`	`cstring`	The email of the recipient(s).
`subject`	`cstring`	The subject of the email.
`message`	`cstring`	The message in the email.
`ack`	`int`	Indication of success.

Description

Sends an email from the Crimson® device to the specified recipient(s) with the specified subject and message. Elements of a recipient list of more than one item should be separated by a semicolon. The function returns immediately, having first added the required email to the system's mail queue. The tag specified as *ack* will increment with completion of each successful action requested. The message will be sent using the appropriate mail transport as configured in the database.

Function Type

This function is *active*.

Return Type

This function does not return a value.

Example

```
SendMailToAck("email1@gmail.com;email2@gmail.com", "Subject", "Message", Tag1);
```

Set(*tag, value*)

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

Description

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 data write behavior is not required.

Function Type

This function is *active*.

Return Type

This function does not return a value.

Example

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

SetIconLed(*id*, *state*)

ARGUMENT	TYPE	DESCRIPTION
`id`	`int`	The icon LED ID enumeration.
`state`	`int`	The icon LED state.

Description

Sets the state of the specified icon LED to the required state, as follows:

ID	LED
1	`Alarm`
2	`Orb`
3	`Home`

Function Type

This function is *passive*.

Return Type

This function does not return a value.

Example

```
SetIconLed(1, 0);
```

SetIntTag(*index, value*)

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

Description

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 requires that the target tag be an integer.

Function Type

This function is *active*.

Return Type

This function does not return a value.

Example

```
int index = FindTagIndex(“Power”);

SetIntTag(index, 1234);
```

SetSystemIo(*prop, value*)

ARGUMENT	TYPE	DESCRIPTION
`prop`	`cstring`	IO Point
`value`	`int`	Value

Description

Outputs the *value* of the IO point specified by *prop* for FlexEdge® models that support On-Board I/O. Digital Output Operating Mode should be set to Programmable in the On-Board I/O section of the Device Configuration category for this function to take effect.

Property	DESCRIPTION
`“DO”`	Digital Output.

Function Type

This function is *active*.

Return Type

This function does not return a value.

Example

```
SetSystemIo(“DO”, 1);
```

SetLanguage(*code*)

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

Description

Sets the operator interface'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);
```

SetPersonalityFloat(*name, data*)

ARGUMENT	TYPE	DESCRIPTION
`name`	`cstring`	The Key Name of the Device Personality.
`data`	`float`	The desired Key Value of the Device Personality.

Description

Sets the specified Device Personality Key Name to the specified Key Value as a float. The function `CommitPersonality()` should be called to commit the new Key Value to internal memory.

Function Type

This function is *active*.

Return Type

This function does not return a value.

Example

```
SetPersonalityFloat("tags.daily.average", Tag1);
```

SetPersonalityInt(*name, data*)

ARGUMENT	TYPE	DESCRIPTION
`name`	`cstring`	The Key Name of the Device Personality.
`data`	`int`	The desired Key Value of the Device Personality.

Description

Sets the specified Device Personality Key Name to the specified Key Value as an integer. The function `CommitPersonality()` should be called to commit the new Key Value to internal memory.

Function Type

This function is *active*.

Return Type

This function does not return a value.

Example

```
SetPersonalityInt("net.faces.eth1.sendmss", Tag1);
```

SetPersonalityIp(*name, data*)

ARGUMENT	TYPE	DESCRIPTION
`name`	`cstring`	The Key Name of the Device Personality.
`data`	`int`	The desired Key Value of the Device Personality.

Description

Sets the specified Device Personality Key Name to the specified Key Value as an IP address. The function `CommitPersonality()` should be called to commit the new Key Value to internal memory.

Function Type

This function is *active*.

Return Type

This function does not return a value.

Example

```
SetPersonalityIp("net.faces.eth1.address", Tag1);
```

SetPersonalityString(*name, data*)

ARGUMENT	TYPE	DESCRIPTION
`name`	`cstring`	The Key Name of the Device Personality.
`Data`	`cstring`	The desired Key Value of the Device Personality.

Description

Sets the specified Device Personality Key Name to the specified Key Value as a string. The function `CommitPersonality()` should be called to commit the new Key Value to internal memory.

Function Type

This function is *active*.

Return Type

This function does not return a value.

Example

```
SetPersonalityString(services.smtp.user, Tag1)
```

SetRealTag(*index*, *value*)

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

Description

Sets the tag specified by *index* to the specified *value*. The index can be found from the tag label using the function `FindTagIndex()`. The function will only be executed if the selected tag is a floating-point type.

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

Sets the tag specified by *index* to the specified value. The index can be found from the tag label using the function `FindTagIndex()`. The function will only be executed if the selected tag is a string type.

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

```
int State = Sgn(Level)+1;
```

ShowMenu(*name*)

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

Description

Shows the page specified by the argument *name*, as a popup menu. 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 the page specified by the argument *name*, as a popup page. The popup will be centered on the display and shown on top of the existing page and any existing popups. The popup will remain visible, and the function will not return until a call is made to `EndModal()`, at which point the 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 but 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 the page specified by the argument *name*, as a nested popup page. 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.

Example

```
ShowNested(Page2);
```

ShowPopup(*name*)

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

Description

Shows the page specified by the argument *name*, as a popup page. 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(Page1)
```

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

```
float 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

This function does not return a value.

Example

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

SirenOn(void)

ARGUMENT	TYPE	DESCRIPTION
`none`		

Description

Turns on the operator interface’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

```
float 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 the function `AddR64()`.

Function Type

This function is *active*.

Return Type

This function does not return a value.

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

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

StopRTTTL(void)

ARGUMENT	TYPE	DESCRIPTION
`none`		

Description

Stops the current RTTTL tune invoked by the `PlayRTTTL()` function.

Function Type

This function is *active*.

Return Type

This function does not return a value.

Example

```
StopRTTTL();
```

StopSystem(void)

ARGUMENT	TYPE	DESCRIPTION
`none`		

Description

Stops the Crimson® device 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

```
int Text = Strip("Mississippi", 's');
```

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

This function does not return a value.

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

```
int 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

```
int 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

```
float 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

This function does not return a value.

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 contains a bitmask representing the various user defined rights, with bit 0 (i.e., the bit with a value of 0x01) representing User Right 1, bit 1 (i.e., 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 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(*address*)

ARGUMENT	TYPE	DESCRIPTION
`address`	`cstring`	The addressed in dotted-decimal form.

Description

Converts a dotted-decimal string specified by the argument *address* into a 32-bit IP address.

Function Type

This function is *passive*.

Return Type

```
int
```

Example

```
int 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

```
float 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*. The *radix* value defines the number base - use 8 for octal, 10 for decimal and 16 for hexadecimal. 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

```
int Data = TextToInt("1234", 10);
```

TextToL64(*input, output, radix*)

ARGUMENT	TYPE	DESCRIPTION
`input`	`cstring`	The text to be converted.
`output`	`int`	The destination of the 64-bit result.
`radix`	`int`	The number base.

Description

Interprets the value stored in the string *input* as a 64-bit double precision integer 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. The *radix* value defines the number base - use 8 for octal, 10 for decimal and 16 for hexadecimal. After execution of this function, the value stored in *result* is suitable for use in other 64-bit math functions.

Function Type

This function is *active*.

Return Type

This function does not return a value.

Example

```
TextToL64(input, output[0], 10);
```

TextToR64(*input, output*)

ARGUMENT	TYPE	DESCRIPTION
`input`	`cstring`	The text to be converted.
`output`	`int`	The destination of 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

This function does not return a value.

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()` function to produce a value that references a particular time and date.

Function Type

This function is *passive*.

Return Type

```
int
```

Example

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

TxCAN(*port, data, id*)

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

Description

Sends CAN messages on a port that has been initialized with a call to `TxCANInit()`. The first four bytes of the message to transmit should be stored using Big Endian byte ordering in the first element of the array, with any further bytes following in the subsequent array entries in the same format. The function returns a value of 1 upon success. The port number is displayed in Crimson®'s status bar of the Communications category when the port is selected.

Function Type

This function is *active*.

Return Type

```
int
```

Example

```
int Result = TxCAN(8, Data, 0x12345677);
```

TxCANInit(*port, id, dlc*)

ARGUMENT	TYPE	DESCRIPTION
`port`	`int`	The raw CAN port.
`id`	`int`	29-bit CAN identifier.
`dlc`	`int`	Data Length Count of 1 – 8 bytes.

Description

Initializes CAN messages to be sent via the CAN Port. This function returns a value of 1 upon success or a value of 0 indicating failure. Calls should be made only after the system has started and each 29 -bit identifier should only be initialized once. The port number is displayed in Crimson®'s status bar of the Communications category when the port is selected.

Function Type

This function is *active*.

Return Type

```
int
```

Example

```
int Result = 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

Selects the inspection file to be used by the camera. The *port* argument is the port number to which the driver is bound. The *camera* argument is the device number displayed in the Crimson® status bar when the camera is selected. More than one camera can be connected under a single driver. The *index* argument represents the inspection file number within the camera. This function will return true if successful, false otherwise. This function should be called in a user program that runs in the background. Calling in the foreground will cause the User Interface to pause.

Function Type

This function is *active*.

Return Type

```
int
```

Example

```
int Result = UseCameraSetup(4, 0, 1);
```

UserLogOff(void)

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(void)

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® 3.2 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()`, this function waits for up to the time specified by the *time* parameter to allow the data to be read. The return value is 1 if the read completed within that period, or 0 otherwise.

Function Type

This function is *active*.

Return Type

`int`

Example

```
int Result = WaitData(array1[8], 10, 1000);
```

WriteAll(void)

ARGUMENT	TYPE	DESCRIPTION
`none`		

Description

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

Function Type

This function is *active*.

Return Type

This function does not return a value.

Example

```
WriteAll();
```

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

```
int 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

```
int Count = WriteFileLine(hFile, “Writing text to file.”);
```

Chapter 3 System Variables

This chapter describes the system variables available within Crimson® 3.2. These system variables can be invoked within actions or expressions, as described in the Crimson 3.2 Software Guide. System variables are used either to reflect the state of the system, or to modify the behavior of the system in some way. When used to reflect system state, a system variable is Read-Only. When used to modify system behavior, a system variable can be assigned a Read / Write value.

ActiveAlarms

Description

Returns a count of the currently active alarms.

Variable Type

`int`

Access Type

Read-Only

CommsError

Description

Returns a bitmask indicating whether 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 (i.e., 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 many times the display is updating per second.

Variable Type

`int`

Access Type

Read-Only

IconBrightness

Description

Contains a value indicating the brightness of the icon LEDs from 0 to 100, with zero being off.

Variable Type

`int`

Access Type

Read / Write

IsPressed

Description

Returns 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 that 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 that TimeZoneMins can only be viewed or changed if the Time Manager is enabled.

Variable Type

```
int
```

Access Type

Read / Write

Unaccepted Alarms

Description

Returns the number of unaccepted alarms in the system.

Variable Type

`int`

Access Type

Read

UnacceptedAndAutoAlarms

Description

Returns the number of alarms that are unaccepted in addition the number of alarms that are currently active and configured to be auto-accepted.

Variable Type

`int`

Access Type

Read-Only

UseDST

Description

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

Variable Type

`flag`

Access Type

Read / Write