Contents

Introduction

What is FreeScript?

FreeScript is a scripting language created by DKT for programming the FreeWay A/V controller. When a FreeScript file is loaded into the FreeWay, it is compiled internally, resulting in binary data, which tells the FreeWay's processor, principally:

- How to handle data coming into the various ports, and
- When to send messages out, and in what format.

Editing FreeScript Files

FreeScript files are text files which observe the rules of the language. They can, if required, be created, read and edited within a simple text editor such as Microsoft® Notepad. This means, for example, that you could create sophisticated system designs using a simple PDA device with a basic text editor. A variety of third-party general-purpose script editing tools (e.g. SlickEdit) are also available which may be used for this purpose.

Learning FreeScript

If you are familiar with programming languages such as 'C', FreeScript will be very familiar to you. FreeScript is a simplification of the 'C' language borrowing many of its basic principles and omitting more of its advanced features. If you are new to programming FreeScript is simple enough to master very quickly.

This manual should be read in conjunction with the Freeway User Guide that covers how to use the FreeWay and its numerous interfaces. The manual is split into 4 main sections:

- **Getting Started** – using a small script as an example, this section gives a brief overview of the anatomy of a script file and an introduction to the main concepts of the language.
- **Language Reference** – this provides a more formal and thorough description of the language and its rules
- **System Functions** – this section describes all of the built-in system functions provided by FreeWay with examples of how to use them.
- **Quick Reference** – this section can be used as a quick reference for the FreeScript system functions and can be kept handy for reference when you're writing your script.

Getting Started

This section uses an example script to acquaint you with the general principles and anatomy of script files. So lets get stuck in and look at the example script file section by section. For clarity, extracts from the script file are written `in this font`. The script will also demonstrate some good coding practices such as commenting, variable and function naming, and code structure.

Example Script

Here's the script in its entirety. The general principles are described in the comments.

```
//-------------------------------------------------------------
// Project: FreeScript Example Script
// File:    script.txt
// Author:  Yan Speake
//
// Notes:
// ------
// This example script shows how we would control a DVD player
// using infra-red (IR) commands.
//
//-------------------------------------------------------------
// Variable declarations
// Declare the variables we'll be using in the script
//-------------------------------------------------------------
string sIRCommand;         // holds an IR command string

//-------------------------------------------------------------
// Constant declarations
// These are effectively variables but their values wont be changed
//-------------------------------------------------------------
string sVERSION;           // version number of the script
float  fDVDON;
float  fDVDSTOP;
float  fDVDPLAY;

//-------------------------------------------------------------
// Function declarations
// Declare the functions we'll be using in the script
// ------------------------------------------------------------
DVDInit();                        // set DVD player to a known state
DVDControl(float fDVDControl);    // control the DVD player

//-----------------------------------------------------
// Function: HubInit()
// Purpose: Initialise the FreeWay for its application
//-----------------------------------------------------
HubInit()
{
  // Set the script version number here
  sVERSION = "V1.10e";

  // Print a Message to the debug port
  // ---------------------------------
  // declare string to hold the welcome message
  string sWelcomeMessage;

  // add the version number to a welcome message
  sWelcomeMessage = "FreeWay DVD Player Version: " + sVERSION + "\n\r";

  // print the welcome message
  DebugPrint(sWelcomeMessage);

  // Now Initialise the DVD Player
  // -----------------------------
  DVDInit();
}
```

```
//-----------------------------------------------------
// Function: DVDInit()
// Purpose: Initialise the DVD Player to a known state
//-----------------------------------------------------
DVDInit()
{
  // Initialise DVD control constants
  fDVDON   = 1;
  fDVDSTOP = 2;
  fDVDPLAY = 3;

  // Turn the DVD Player on if it isn't already
  DVDControl(fDVDON);

  // Wait 5 seconds to allow the DVD player to turn on
  Delayms(5000);

  // Now make sure the DVD player is stopped
  DVDControl(fDVDSTOP);
}

//-----------------------------------------------------
// Function: DVDControl()
// Purpose: Control a Toshiba SD900E DVD Player using
//          InfraRed commands on IR output port IR1
//-----------------------------------------------------
DVDControl(float fDVDControl)
{
  // Initialise with a dummy string
  sIRCommand = "";

  // test the passed argument
  if (fDVDControl == fDVDON) {
    sIRCommand = "[PF68L8C62459708B71A272312X42F54D4CDFEC580P4R01]";
  }
  if (fDVDControl == fDVDSTOP) {
    sIRCommand = "[PF68L8C62459708B71A272312X42FDFEC45580P4D4CR01]";
  }
  if (fDVDControl == fDVDPLAY) {
    sIRCommand = "[PF68L8C62459708B71A272312X42F5580P4D4CDFEC4R01]";
  }
  // transmit the Infra-Red command on IR port IR1
  SendIR(1,0,0,0,sIRCommand);
}
```

Example Script in Detail

Script Header

```
//-------------------------------------------------------------
// Project: FreeScript Example Script
// File:    script.txt
// Author:  Yan Speake
//
// Notes:
// ------
// This example script shows how we would control a DVD player
// using infra-red (IR) commands.
//
//-------------------------------------------------------------
```

Though not obligatory your script should start with some form of comment like the one shown – this is good coding practice. The above extract demonstrates how to use comments.

Comments

A comment line starts with a `//`. The compiler will ignore any text that follows a `//`. A comment can also be appended to a line of code as the next script extract demonstrates:

```
// Variable declarations
// Declare the variables we'll be using in the script
//-------------------------------------------------------------
string sIRCommand;         // holds an IR command string
```

This extract demonstrates the following concepts: variable types, variable declaration and variable naming conventions.

Variable Types

There are two variable types available: **float** and **string**. The above extract is an example of a string variable that can hold text values up to 256 characters in length.

Variable Declaration

Before a variable can be used, it needs to be declared – this lets the compiler know about it before you use it. The above extract declares the string variable `sIRCommand`.

Note that the declaration ends with a semi-colon (;) - most (but not all) lines of script code must end in a semicolon. Missing semicolons are usually the first things to look for when you get compiler errors.

Variable Naming Conventions

Note that the name of the string variable begins with a small `s`. This tells us that the variable is of type string. When we use this variable later on in the script we can easily tell that it's of type `string` and wont try to assign a numeric value to it . Though this is not obligatory, it is good coding practice that will minimise compiler errors later on. So for example if we were to declare a `float` variable we would similarly precede it with a small `f`.

The next extract shows more variable declarations but with a slight difference.

```
//-------------------------------------------------------------
// Constant declarations
// These are effectively variables but their values wont be changed
//-------------------------------------------------------------
string sVERSION;           // version number of the script
float  fDVDON;
float  fDVDSTOP;
float  fDVDPLAY;
```

These variables will effectively act as constants i.e. they will be given a value once during initialisation and their values wont change thereafter – they are used to make the code more readable. By convention, constants are always in UPPER CASE letters to differentiate them from variables whose values will change. Again this is not obligatory – just a practice to reduce mistakes and make the code more readable.

The next script extract demonstrates function declarations and function arguments.

Function Declaration

```
//-------------------------------------------------------------
// Function declarations
// Declare the functions we'll be using in the script
// -------------------------------------------------------------
DVDInit();                      // set DVD player to a known state
DVDControl(float fDVDControl);   // control the DVD player
```

Lets look at each in turn. These are all functions that we will use later in the script and, as for variables, we need to let the compiler know about them first.

The DVDInit function is used to initialise the DVD Player to a known state (turned on but stopped). Note that a function declaration has no type. In the declaration, the function name is followed by brackets (). This tells the compiler that it's dealing with a function not a variable. Again, the declaration ends in a semi-colon.

DVDControl is a function that we will use later on to control the DVD player. When we use DVDControl later on, we will pass it an argument telling it how we want to control it (Play, Stop, Rewind etc). We declare this in the argument list of the function: float fDVDControl. This does two things

1. It tells the compiler that the function DVDControl takes one argument of type float called fDVDControl
2. It also declares the float variable fDVDControl

Note that while variables can be declared outside and inside function bodies, even in function declarations – functions can only be declared outside function bodies.

Function Naming Convention

Note that the above functions are related as they all deal with a DVD player. For this reason all the functions start with DVD. Though not obligatory this practice will make your script much more logical and readable. Note also that any variables related to the DVD player will also begin with DVD for the same reason.

The next code extract shows the HubInit function. This demonstrates function definition, system functions and use of variables.

```
//-----------------------------------------------------
// Function: HubInit()
// Purpose: Initialise the FreeWay for its application
//-----------------------------------------------------
HubInit()
{
  // Set the script version number here
  sVERSION = "V1.10e";

  // Print a Message to the debug port
  // ---------------------------------
  // declare string to hold the welcome message
  string sWelcomeMessage;

  // add the version number to a welcome message
  sWelcomeMessage = "FreeWay DVD Player Version: " + sVERSION + "\n\r";

  // print the welcome message
  DebugPrint(sWelcomeMessage);

  // Now Initialise the DVD Player
  // ---------------------------------
  DVDInit();
}
```

Variable initialisation

The first line of HubInit initialises the string constant sVERSION. Note that because it is a string the value it's given is enclosed in quotes "".

The next line shows a declaration of the string variable sWelcomeMessage. This is an example of declaring a string within a function body. You may want to do this if the variable is only ever used within that function. If you are familiar with other programming languages its worth noting that, even though the variable is declared locally (i.e. within a function body) its scope is still global (i.e. the variable can still be accessed outside the function).

String concatenation

The next line shows how constant and variable strings can be added together using the '+' operator. Note that the \n\r characters are special control characters for carriage return & line feed.

System functions

The next line `DebugPrint(sWelcomeMessage)` calls a built-in system function used for sending strings to the debug interface. Because it's a system function, the compiler already knows about it and it doesn't need to be declared or defined. Note that we pass the string variable to it as an argument. Monitoring the version numbers of your scripts is a very good practice.

User Functions

The final line calls one of our user functions `DVDInit()`. Even though we haven't defined what that function does yet - the compiler still knows about it because it has been previously declared.

The next code extract is the definition for the function `DVDinit()`.

```
//-----------------------------------------------------
// Function: DVDInit()
// Purpose: Initialise the DVD Player to a known state
//-----------------------------------------------------
DVDInit()
{
  // Initialise DVD control constants
  fDVDON   = 1;
  fDVDSTOP = 2;
  fDVDPLAY = 3;

  // Turn the DVD Player on if it isn't already
  DVDControl(fDVDON);

  // Wait 5 seconds to allow the DVD player to turn on
  Delayms(5000);

  // Now make sure the DVD player is stopped
  DVDControl(fDVDSTOP);
}
```

The first three lines initialise the `float` constants declared at the top of the script. The purpose of these constants will now become clear. The next line makes a call to the other user function `DVDControl()`. Note that the command is passed to the function as a `float` argument - using the float constant is much more readable than just passing numbers.

The `Delayms()` function is another system function that, again, requires no declaration. It simply generates a delay specified by the argument you pass it – in milliseconds. The above example makes the script hang around for 5 seconds while the DVD player initialises itself.

The next function shows how arguments passed to a function can be tested and then something useful done with them.

```
//-----------------------------------------------------
// Function: DVDControl()
// Purpose: Control a Toshiba S900 DVD Player using
//          InfraRed commands on IR output port IR1
//-----------------------------------------------------
DVDControl(float fDVDControl)
{
  // Initialise with a dummy string
  sIRCommand = "";

  // test the passed argument
  if (fDVDControl == fDVDON) {
    sIRCommand = "[PF68L8C62459708B71A272312X42F54D4CDFEC580P4R01]";
  }
  if (fDVDControl == fDVDSTOP) {
    sIRCommand = "[PF68L8C62459708B71A272312X42FDFEC45580P4D4CR01]";
  }
  if (fDVDControl == fDVDPLAY) {
    sIRCommand = "[PF68L8C62459708B71A272312X42F5580P4D4CDFEC4R01]";
  }
```

```
    // transmit the Infra-Red command on IR port IR1
    SendIR(1,0,0,0,sIRCommand);
  }
```

Note that the name of the function must match that of its declaration.

The float value that we pass into the function is tested using the `if(condition)` operation. If the condition tested is true then the code within the following `{}` brackets will be executed. In this case, an Infra-Red command string is initialised. The final function is a system function for transmitting IR codes from the Infra-Red output ports.

That's it. Obviously for practical applications we'd want to control the DVD player more comprehensively by supporting commands for Pause and Rewind, etc. We'd probably also want to add functions to control other devices such as VCRs and satellite receivers - or whatever your application requires. Some devices may also need to be controlled using RS232, RS485 or via Ethernet. If we had implemented a user interface for this application then to control the DVD player, it would call the DVDControl() function from outside the script (refer to the FreeWay User Manual for more information on this).

The following section now provides a more thorough explanation of the language concepts covered above.

SECTION 1 – LANGUAGE REFERENCE

Comments

Data Types

Functions

Operators

Conditional Operations

Loops

String Operations

Common Mistakes

Comments

Probably the most useful aspect of your script will be the comments. Without these, your script will become more difficult to read, understand and debug. Use them often. Comments can be inserted at any point in your script using the '//' characters. Note that the FreeWay compiler will ignore all characters from the beginning of the '//' comment declaration to the end of that line. Here are some examples of how to include comments within your code.

```
//-------------------------------------------------------------
string sVERSION;          // version number of the script
float  fDVDON;
```

Note that in the following example, a compiler error would arise as the code 'float fDVDON;' would be ignored and treated as a comment.

```
//-------------------------------------------------------------
string sVERSION;          // version number of the script  float  fDVDON;
```

Data Types

There are only two data types in FreeScript:

- float
- string

float

This is used for positive & negative integer and fractional numeric values. For example:

```
float value1;
float value2;

value1 = -2;
value1 = 1.234567;
```

Values can be positive or negative.

string

This is used for all textual or ASCII values. It represents a string of 8-bit characters. The length of a string can be up to 256 characters. All of the following examples will generate a 'hello' string.

```
string message1;
string message2;

message1 = "hello";                  // a string
message2 = "hell" + "o";        // adding two strings
```

Functions

There are two main types of functions: User Functions and System Functions

User Functions

User functions are those functions that you will write yourself which may call other user or system functions, the rules of which are covered in this section.

System Functions

System functions are split into two categories: normal system functions and system event functions.

Normal system functions are functions built-in to the FreeWay & FreeScript and perform specific system tasks such as sending data to serial ports and controlling its LEDs.

The system *event* functions are special in that they are automatically called when certain system events occur. For example, receiving messages on serial ports, or alarms going off. Note that system event functions will only ever be called if they are included in your script.

While user functions always need to be declared before they are used (see later), system functions do not (as FreeWay already knows about them).

Example Function

Here is an example of a user function that we will refer to during the course of this section:

```
//-----------------------------------------------------
// Function: DVDControl()
// Purpose: Control a Toshiba SD900E DVD Player using
//          InfraRed commands on IR output port IR1
//-----------------------------------------------------
DVDControl(float fDVDControl)
{
  // test the passed argument
  if (fDVDControl == fDVDON) {
    sIRCommand = "[PF68L8C62459708B71A272312X42F54D4CDFEC580P4R01]";
  }
  else {
    sIRCommand = "";       // error - use a dummy string
  }
  // transmit the Infra-Red command on IR port IR1
  SendIR(1,0,0,0,sIRCommand);
}
```

This is a function that controls a DVD player (simplified for brevity – it only supports the ON command). The function is passed an argument to tell the function what to do.

Function Name

The very first thing that is required when creating a function is a unique **name**. This enables the function to be identified and called by other functions within your script. In the case above, the name of the function is `DVDControl`. The name of the function should be chosen carefully to allow you to quickly identify it later. The following rules apply to the function name:

- It must be unique within your script
- It must be no more than 32 characters long

- It should include no spaces or other punctuation
- It should only contain alphabetical letters and numbers
- The first character should not be a number - it should be a letter

Function Parameters

The function name is *immediately* followed by **parameters**, enclosed within parentheses '()' – also referred to as **arguments**. This area can be left empty but the parentheses should still be present. The parameters are variable names, enabling data values to be passed between functions. So, for example, you might have a general-purpose function, which transmits data on serial ports. The actual data that you want to transmit will vary, so you would declare a function something like this:

```
// Note that the type of the parameter should precede its name
TransmitData(string sData);
```

To send the data, you would simply set the value of data and call the TransmitData function, as shown below:

```
TransmitData("Hello");
```

Note that when a variable is used within the title line of a function in this way, this also serves to declare the variable, so no other declaration of that variable should exist within your script.

A function can have up to eight parameters. When creating a function with multiple parameters, simply separate each parameter with a comma. For example:

```
TransmitData(string sData1, float fData2, string sData3);
```

Similarly, multiple parameters can be passed between functions. The function call should pass the same amount of data in the same order (using the same comma separators) as in the function definition, otherwise compiler errors may occur.

Note that if you want to call functions from outside your script (using a URL for example – see the FreeWay User Guide) then the following restrictions apply:

- Only 4 parameters can be passed
- They must be of type `float`

Function Body

The **body of the function** then follows, which or may not include comments (we recommend the liberal use of comments, as this will help you when testing and trouble-shooting your code). There may be any number of '{ }' pairs within the body of the function but the end of the function is always denoted by a concluding '**}**'.

Line Termination

Note that each line of code *within* the function is terminated by a semicolon. This character should precede any 'in-line' comments as shown below:

```
fDVDON   = 1;     // DVD On command
fDVDSTOP = 2;     // DVD stop command
```

The following line would result in an error:

```
fDVDON   = 1      // DVD On command
fDVDSTOP = 2;     // DVD stop command
```

as the compiler will 'see' this as

```
fDVDON = 1fDVDSTOP = 2;
```

The only lines not requiring a semicolon are:

- The 'title line' of your function
- Lines terminating in a '{' or a '}'
- Conditional tests (`if` and `else` - see later)
- Loop command (`while` – see later)
- Comment lines

Note that it is only within the body of functions that setting, initialisation and mathematical/logical manipulation of variable values can take place. To initialise variables on power-up, do so within the `HubInit()` function. Note that the only valid operation outside of functions is declaring variables (and, of course, making comments).

Declaring Functions

Note that a function must be declared before being called. e.g.:

```
func1(float a, string b);
```

The best place to declare your functions are at the top of the script file before the HubInit() function. All your variable declarations can be put in this area as well. Note that although variable declarations can occur within function bodies, function declarations cannot.

Nesting Functions

Functions may be called within other functions. Nesting is permitted up to 32 layers deep.

Use of Parentheses

Note that for each '{' there should also be a '}' at some point in your code and that if these are 'nested' then the compiler will always treat a '}' as relating to the most recent occurrence of '{'. This is a universal coding convention and needs to be very carefully observed.

You will dramatically boost the clarity of your layout, and reduce the likelihood of errors, if you ensure that each time you use a '{' the indentation of subsequent lines is increased by one 'tab' and that each time you use a '}' the indentation of that line and subsequent lines is reduced by one 'tab' as shown in the example below.

```
MyFunc()
{
  SerialSend(fPort,"SETL");
  if (fIO == 0) {
    SerialSend(fPortNo,"INPMUTE");
  } else {
    SerialSend(fPortNo,"OUTMUTE ");
  }
  SerialSend(fPortNo," ");
}
```

A good practise, when creating functions would be to immediately follow a '{' with a '}' and then to place the remaining code between these elements.

Understanding System Event Functions

FreeWay provides a number of system event functions that, if included in your script, will automatically be called when certain events occur. These are:

Alarm()	- called when an alarm event happens
DigitalInput()	- called when a digital input state changes
HubInit()	- called when FreeWay powers up
SerialReceive()	- called when a message is received on a serial (or Telnet) port
IRReceive()	- called when the Infra-Red Receiver decodes an RC5 message

If you are familiar with programming concepts, you may have come across interrupts and their service functions. Generally an interrupt service function will be called and executed when some external event happens regardless of what code is already being executed (i.e. the current code is interrupted to run the interrupt service function). It's important to understand that FreeScript event functions *do not* act like interrupts.

If an event occurs (such as receiving a serial port message, or a digital input state change) FreeScript event functions will only be called after the current function (either user or system function) has finished. This has the following implications:

- Keep the amount of time you spend in any function to a minimum
- Keep your delays (using the Delayms() function) to a minimum
- Keep your loops as short as possible

Operators

In order to manipulate the values of variables, a series of familiar symbols such as '=', '+' and so on, along with some less familiar but useful ones, are available. Collectively, these symbols are known as operators.

Here is a complete list of the operators available to you within the FreeScript language:

Operator	Operation	Example
=	Sets a value	x = 3 y = "hello"
+	Simple addition	x = y + 2
-	Simple subtraction	x = y - 2
*	Simple Multiplication	x = y * 2
/	Simple Division	x = y/2
%	Remainder	x = y%2
==	Comparator	if (x == 2)
!=	Inequality	if (x != 2)
&&	Logical AND	if (x && b) { c=1; }
||	Logical OR	if (x || b) { c=1; }
!	Logical NOT	x = !y; if (!x) { c=1; }
&	Bitwise AND	z = x & y;
|	Bitwise OR	z = x | y;
>	Greater than	if (x > y) { c=1; }
<	Less than	if (x < y) { c=1; }
>=	Greater than or equal to	if (x >= y) { c=1; }
<=	Less than or equal to	if (x => y) { c=1; }

Assignment (=)

Use this operator to assign the value on the right hand side of the statement to the variable on the left hand side.

```
fValue1 = 2.34;
fValue2 = fValue1;

sString1 = "Hello";
sString2 = sString1;
```

Assignments must be of the same type. The following examples will cause unpredictable results:

```
fValue1 = "hello";          // wrong !!
sString1 = 2.34;            // wrong !!
```

Be careful not to use the assignment operator in if(condition) statements – a common error. Unpredictable results will occur.

```
if(fValue1 = 3) {fValue2 = 3;}   // wrong !!
if(fValue1 == 3) {fValue2 = 3;}  // correct
```

Addition (+)

This operator applies to both strings and floats but with different results for example:

```
fValue1  = 3 + fValue2;    // adds 2 numbers together

sString1 = "Hell" + "o";   // concatenates two strings
```

Subtraction (-)

This operator only applies to `float` values – you can't use it for strings.

```
fValue1 = fValue2 – 4.5;
```

Multiplication (*)

This operator performs simple multiplication of `float` values.

```
FValue1 = 1000000 * 0.0000001;    // equals 1
```

Division (/)

This operator performs simple division of `float` values.

```
FValue1 = 1000000 / 1000000;      // equals 1
```

Comparison (==)

You can only use this operator in the `if(condition)` statement to compare two float values. You cannot compare two strings. Be careful not to confuse == with the assignment operator (=) in `if` statements.

```
if(fValue1 = 3) {fValue2 = 3;}   // wrong !!
if(fValue1 == 3) {fValue2 = 3;}  // correct
```

Logical AND (&&)

This operator can be used to test compound conditions in if(condition) statements. Don't confuse it with the Bitwise AND operator (&).

```
if(fA && fB)     {fC=1;}
else             {fC=0;}
```

In this example, `fC` will only be assigned a value of 1 if both `fA` and `fB` are non-zero. If one of them is zero `fC` will be assigned the value zero.

Logical OR (||)

This operator can be used to test compound conditions in if(condition) statements. Don't confuse it with the Bitwise OR operator (|).

```
if(fA || fB)     {fC=1;}
else             {fC=0;}
```

In this example `fC` will only be assigned a value of 1 if either `fA` and `fB` are non-zero. If both of them are zero `fC` will be assigned the value zero.

Logical NOT (!)

This operator is used to invert the logic of a variable - making a true a value false and vice versa. It's best demonstrated by example.

```
fValue1 = 1;              // fValue1 = 1 (true)
fValue2 = !fValue1;       // fValue1 is inverted, fValue2 = 0 (false)
```

The operator will convert any non-zero value into a zero value and any zero value into a 1 value.

The operator can also be used in `if(condition)` statements to invert the logic of a condition. For example:

```
fValue1 = 0;

if(!fValue1)    {fC=1;}
else            {fC=0;}
```

In this case `fValue1` is false but has been inverted to true in the `if` condition. Therefore, `fC` will be assigned a value of 1.

Bitwise AND (&)

This operator is used to perform the bitwise AND operation.

```
fValue3 = fValue1 & fValue2;
```

`fValue3` will only be assigned a value of 1 if both `fValue1` and `fValue2` are equal to 1. If one or both of them are zero then `fValue3` will be assigned a value of zero.

Bitwise OR (|)

This operator is used to perform the bitwise OR operation.

```
fValue3 = fValue1 | fValue2;
```

`fValue3` will be assigned a value of 1 if either `fValue1` or `fValue2` are equal to 1. If both of them are zero then `fValue3` will be assigned a value of zero.

Greater Than (>)

This operator is only used in `if(condition)` statements. For example

```
fValue1 = 2;
fvalue2 = 3;

if (fValue1 > fValue2) {fValue3 = 6;}  // condition is false
```

Less Than (<)

This operator is only used in `if(condition)` statements. For example

```
fValue1 = 2;
fvalue2 = 3;

if (fValue1 < fValue2) {fValue3 = 6;}  // condition is true
```

Greater Than or Equal To (>=)

This operator is only used in `if(condition)` statements. For example

```
fValue1 = 2;
fvalue2 = 3;

if (fValue1 >= fValue2) {fValue3 = 6;}  // condition is false
```

Less Than or Equal To (<=)

This operator is only used in `if(condition)` statements. For example

```
fValue1 = 2;
fvalue2 = 3;
```

```
if (fValue1 <= fValue2) {fValue3 = 6;}  // condition is true
```

Conditional Statements

Conditional statements are the decision makers within your code. For example, you can test the value of a variable and, depending on the result, execute a different section of code.

For example, in order to fire a preset in a piece of equipment, you might create a variable called `fPresetNo` and a function called `FirePreset`, which acts on the value of `fPresetNo`. Before calling `FirePreset`, you might want to check that `fPresetNo` is in the range 1-16. You would do this as follows:

```
if ((fPresetNo <= 16) && (fPresetNo >= 1))
{
  FirePreset(fPresetNo);
}
```

You might want to control what happens if these conditions are not met (error-trapping). This is done using the '`else`' statement. For example, if `fPresetNo` is not in the valid range, we might want to set an error flag:

```
if ((fPresetNo <= 16) && (fPresetNo >= 1))
{
  FirePreset(fPresetNo);
}
else
{
  errFlag=1;
}
```

Many programming languages will provide an '`else if`' facility within conditional statements. FreeScript doesn't. The same result is achieved in FreeScript using 'nested ifs' as shown below:

```
if (fPresetNo <= 16) && (fPresetNo >= 1)
{
  FirePreset(fPresetNo);
}
else
{
  if (fPresetNo == 0)
  {
    errFlag=1;
  }
  else
  {
    errFlag=2;
  }
}
```

Here, the preset will be fired if it is in the range 1-16 inclusive. Otherwise, if it is '0' then the `errFlag` variable is set to 1 and, if it is any other value, the `errFlag` variable is set to 2.

Loops

Loops enable you to repeat a section of code until a certain condition is met. For example, if you had a network of controllable devices on one of the serial ports and these devices each had a unique ID then you might want to repeat a section of code, once for each connected device.

The following loop statement is available within FreeScript

```
while (condition){ }
```

For example, you might want to send a message out of a serial port 5 times. This would be done as follows:

```
fCount = 1;
while(fCount <= 5)
{
  SerialSend(1, "This is the message");
  fCount = fCount + 1;
}
```

What happens here?

First, the variable `fCount` is initialised to the numerical value 1. Then we enter a loop which is repeated for as long as the variable `fCount` is less than or equal to 5. However, each time we go through the loop (known as an *iteration*), the value of `fCount` is incremented. Each time the loop has been executed, the condition is re-tested and, if true, the loop runs again. The outcome is that the loop is executed five times because after the fifth iteration, the value of `fCount` is incremented to 6 and the condition is no longer true.

Great care should be taken not to create endless loops. An example of such a loop is shown below:

```
fCount = 5;
while(fCount >= 5)
{
  SerialSend(1, "This is the message");
}
```

Here, the value of `fCount` is initialised at 5 and the loop will continue to run while `fCount` is greater than or equal to 5. After each iteration, `fCount` is not adjusted, so it will never go below 5!

The consequence of an endless loop is that the processor inside the FreeWay would be permanently occupied with executing this loop and other functions within the script would not be able to run, which would mean that control of your A/V installation would completely break down. So you can see why great care should be exercised when creating loops.

String manipulation

Several system functions and operators are available which work well for manipulating string variables.

Concatenation

To concatenate (add together) two string variables, simply use the '+' operator. So if you had a string variable called sMsg1 and another called sMsg2 whose values were, respectively "Hello" and "World", you can add them together and place them in another variable, sMsg3, as follows:

```
sMsg3 = sMsg1 + sMsg2;
```

Now, sMsg3 would have the value "HelloWorld".

You could achieve the same in several ways, such as:

```
sMsg3 = sMsg1 + "World";
```

or

```
sMsg3 = "Hello" + sMsg1;
```

Using ASCII codes

If we know the ASCII character code for a character (see Quick Reference Section), we can insert it into a string using the '\' operator. For example, the ASCII code for a space is 32. So to put a space between the two words above, we could use:

```
sMsg3 = "Hello\032" + sMsg2;  // sMsg3 is now set to "Hello World"
```

Note that there must always be 3 digits following the \ for the ASCII code.

Using Numerical Values

We can add numerical variables to string variables in the same way and they will be treated as strings for that operation. For example, if we have a variable fValue1 whose value was 5 and we performed the operation:

```
sMsg = "Preset" + fValue1;
```

The string variable sMsg now has the value "Preset5".

Strings as Arrays

We can treat strings as arrays to replace or add characters to them. For example:

```
sMsg1 = Hello;
sMsg1[1] = \097;              // ASCII code for 'a' is 97 decimal
sMsg1[1] = \x61;              // ASCII code for 'a' is 61 hexadecimal
```

sMsg1 now says 'Hallo'. This technique allows you to access the n^{th} element of the string variable. Note that the array index starts from 0 not 1. Note that we can only use ASCII codes with this technique.

Other tools for manipulating strings

midstr()

This system function returns a subset of the contents of a string variable

```
midstr(string sSrc, float fStart, float fLength)
```

'sSrc' is the name of the variable to be operated upon; 'fStart' is the start-point within the string; 'fLength' is the number of characters to be 'extracted' from the string. E.g.:

```
string sNewWord;
sNewWord = midstr("Testing", 3, 4);
```

will return "ting".

strlen()

This system function returns the length (the number of characters) of a string variable. E.g.

```
strlen("Howdedoody");
```

Will return a value of 10.

format()

The **format** function has been provided to enable numbers to be converted to strings. The function syntax is as follows:

```
float fValue1;
string sMessage;

fValue = 7;

sMessage = format(fValue1, 1, 0);  // integer format: sMessage = "7"
sMessage = format(fValue1, 2, 0);  // float format: sMessage = "7.000000"
```

Other String Rules

- Maximum function/variable name length is 24 characters
- String variable can be up to 256 characters in length and can contain any 8-bit character.
- Names must start with a letter (a-z, A-Z), subsequent characters can be numbers or letters (a-z, A-Z, 0-9) but no punctuation.
- Names are case sensitive, i.e. Msg and msg are two completely different variables.
- Every function and variable must have a unique name.
- Names must not conflict with keywords (listed below).
- Names must not conflict with system function names.

Keywords

Reserved keywords are:

if	else	while
float	string	

Common Errors

Here are some pitfalls that you may fall into when first familiarising yourself with the FreeScript language:

General

The best advice for writing scripts is to write your scripts in small chunks at a time. Each time you add a piece of functionality save the script, compile it and test it. This way you will build your script with confidence that previous code is working.

Do not be tempted to write huge amounts of script and then compile it. If, in the likely event, that there are some errors it will be more difficult to deduce where they are. You will probably end up wasting time stripping the script back to bare bones again to find the errors.

Missing Semicolon ;

This is probably the most common script error. If you get a compiler error check that you are not missing the required semicolon at the end of each statement.

Variables

Multiple declarations

Remember that if a variable is used by a function as a parameter, it should not:

- Be declared anywhere else
- Be used as a parameter within any other function

No declaration

All variables must be declared in one of the following ways:

- Explicitly e.g.

```
float x;
string y;
```

- Implicitly within a function title e.g.

```
MyFunc(float x, string y){}
```

Usage precedes declaration

Note the sequence in which variables are declared, initialised and used within your script is very important. You cannot use or initialise a variable until it has been declared.

Invalid initialisation

Variable declaration can happen within and outside of function bodies but variable initialisation should only take place within function bodies e.g.

```
float fInputID;     // declare outside a function

Function1()
{
  float fOutputID;  // declare inside a function
```

```
  fInputID  =2;     // initialise inside a function only
  fOutputID =3;     // initialise inside a function only
}
```

Note that the *only* valid operation outside of functions is declaring variables (and, of course, making comments).

Functions

Invalid Name

A function name should be less than 24 characters, have no punctuation, start with a letter and be unique within your script. Take care not to use the names of system functions which you may not necessarily be using, or be aware of, in your script.

Parameter Mismatch

When calling a function you should ensure that the *quantity*, *type* and *sequence* of the variables you pass match those in the declaration of the function being called.

Bracket Mismatch

Ensure that all opening brackets, of any type, are complemented by the same number of closing brackets within the function.

Conditional Statements

Equality tester

Note that in using the equality tester within 'if' statements, the assignment operator often gets used by mistake. So the syntax should be `if(a==b){}` rather than `if(a=b){}`.

'Else If' and 'If Then' used

Though common in other languages, these statements are not valid in FreeScript and should be replaced simply by `else{}` and `if{}` respectively.

Incorrect nesting

Remember to indent correctly and that '}' always relates to the last occurrence of '{'.

Loops

Endless loop

This will kill your script and you need to be very careful that your condition will always be met at some point in the near future when entering a loop.

De-bugging tools

There are several methods that you can use to assist in de-bugging your scripts and systems.

A report is generated when your script is compiled by the FreeWay. This can be useful in highlighting errors within your script.

You can deliberately put code into your script that is used for de-bugging purposes. For example, you could have a message sent out of one of the serial ports indicating information of interest such as variable values, events being triggered, incoming data and so on. You can then view the output from the serial port using a terminal programme running on your computer or laptop.

It may be that no serial ports can be 'spared' for de-bugging purposes so we have provided a facility to use the Telnet port as a de-bugging port during configuration of your system. In order to do this, you will need to put the Telnet port into 'debug mode'. This is done using the menu system.

Then you will need to put instructions into your script to send relevant information to the debug port. This is done using the `DebugPrint()` embedded function. The syntax for this is very simple and is shown below:

```
DebugPrint(string sDebug);
```

The parameter '`sDebug`' is simply the information that you want to output from the debug port. So, for example, if you wanted to send out an alert every time a valid message was received on serial port 1, you would do it as follows:

```
SerialReceive(float fPort)
{
  if (fPort == 1)
  {
    DebugPrint("Got a message on Port 1!\n\r");
  }
}
```

You could make this more sophisticated by outputting the actual data that had just come in, as follows:

```
SerialReceive(float fPort)
{
  string sIncoming;
  string sOutgoing;

  if (fPort == 1)
  {
    sIncoming = SerialGet(1);
    sOutgoing = "Just got this message on Port 1 - " + sIncoming;
    DebugPrint(sOutgoing);
  }
}
```

So each time a valid message is received on Serial Port 1, the message will be echoed out of the Telnet port, whenever that port is in Debug mode, for you to inspect using a terminal emulator program.

Note that when the Telnet port is not in debug mode, DebugPrint instructions will be ignored by the FreeWay processor.

SECTION 2 – SYSTEM FUNCTIONS

Function List by Type

Functions are listed by type as follows:

Serial Port Functions

ConfigPort()	Set serial port comms parameters
SetBaud()	Set serial port baud rate
SerialReceive()	SystemEvent when a serial (or Ethernet) message is received
SerialGet()	Get a received serial message
SerialSend()	Send a serial (or Ethernet) message
SetEndOfMsg()	Specify a message terminator for Rx
SetMsgLength()	Specify a message length for Rx
SetMsgTimeout()	Specify a message timeout for Rx

IR Functions

IRReceive()	System Event when RC5 IR is received
SendIR()	Transmit learned IR commands

EIB Functions

EIBPhysical()	Set FreeWAy physical address
EIBReceive()	System Event when EIB telegram is received
EIBRegister()	Register an EIB group
EIBSendFloat()	Send float value to the EIB bus
EIBSendString()	Send string to the EIB bus

Alarm Functions

Alarm()	System Event when an alarm occurs
SetAlarm()	Configure alarms
SetLocation()	Set location for astronomic clock

String Manipulation Functions

Format()	Change a float value into a string
MidStr()	Extract one string from another
Strlen()	Get the length of a string

Telnet Functions

ConfigTelnet()	Select a Telnet or TCP connection
OpenTelnet()	Opena Telnet/TCP connection
CloseTelnet()	Close a Telnet/TCP connection

UDP Functions

OpenUDP()	Listen to a UDP port
ConfigSendUDP()	Configure UDP tx parameters

Digital I/O Functions

DigitalIn()	System Event when digital input changes
GetDin()	Read digital input status
SetDout()	Set a digital output state

Date & Time Functions

GetDate()	Get the date
GetTime()	Get the time

Other Functions

HubInit()	System Event when FreeWay powers up
DebugPrint()	Send a message to the debug window
Delayms()	Add a delay in milli-seconds
SetHubLED()	Change a front panel LED

Alphabetical Function List

Functions are listed alphabetically as follows:

- Alarm()
- CloseTelnet()
- ConfigPort()
- ConfigSendUDP()
- ConfigTelnet()
- DebugPrint()
- Delayms()
- DigitalIn()
- EIBPhysical()
- EIBReceive()
- EIBRegister()
- EIBSendFloat()
- EIBSendString()
- Format()
- GetDate()
- GetDin()
- GetTime()
- HubInit()
- IRReceive()
- Midstr()
- OpenTelnet()
- OpenUDP()
- SendIR()
- SerialGet()
- SerialReceive()
- SerialSend()
- SetAlarm()
- SetBaud()
- SetDout()
- SetEndOfMsg()
- SetHubLed()
- SetLocation()
- SetMsgLength()
- SetMsgTimeout()
- Strlen()

Alarm

Syntax

```
Alarm(float fAlarm)
```

Arguments

type	name	description
float	fAlarm	the number of the alarm event (1 to 8) which has occurred

Returns

None.

Purpose

This is a system event function and will be called automatically when one of the alarm conditions is true. Eight alarm events are available. You need to first set up an alarm using the SetAlarm() system function. Alarm events can be setup to occur:

- Once only
- Daily
- Weekly
- At Dusk or Dawn

See Also

```
SetAlarm(float fAlarm, string sConfig)
```

Example

The following example sets up a one-time alarm in `HubInit()`. When the alarm occurs a debug message is printed.

```
HubInit()
{
  // one time alarm on 27th November 2004 at 9 in the morning
  SetAlarm(1, “O 27/11/2004 09:00”);
}

Alarm(float fAlarm)
{
  if (fAlarm == 1) {
    DebugPrint(“Its my birthday!! \n\r”);
  }
}
```

CloseTelnet

Syntax

```
CloseTelnet(float fPort)
```

Arguments

type	name	description
float	fPort	specifies one of four the Freeway ports. This can be 10, 11, 12 or 13

Returns

None.

Purpose

Closes a Telnet port that was previously opened using OpenTelnet().

See Also

```
OpenTelnet(float fPort, float fCS, float fIPport string sIPaddr)

ConfigTelnet(float fPort, float fMode)

string SerialGet(float fPort)

SerialReceive(float fPort)

SerialSend(string sOutput)
```

Example

```
HubInit()
{
  // Open a Telnet client port to a Linn Kivor Jukebox
  OpenTelnet(10, 0, 6789, "192.168.1.27");

  // send a message to it
  string sMessage;
  sMessage = "Hello Kivor";
  SerialSend(10, sMessage);

  // close the port again
  CloseTelnet(10);
}
```

ConfigPort

Syntax

```
ConfigPort(float fPort, string sConfig, float fHandshake)
```

Arguments

type	name	description
float	fPort	the number of the port we are configuring
		1-6 RS-232 ports 1-6
		7,8 RS-485 ports 7 & 8
		9 RS-485 RJ45 port
string	sConfig	the configuration string for the port settings
		The format is "`data bits, parity, stop bits`" where
		data bits = 7 or 8
		parity = o (odd), e (even), n (none)
		stop bits = 1 or 0
		The default setting is "8,n,1" – 8 data bits, no parity, 1 stop bit
float	fHandshake	0 = no handshaking, 1 = handshaking enabled via the RTS/CTS lines

Returns

None.

Purpose

This function configures the advanced serial port settings for each port. You can configure the number of data bits (7 or 8), the parity (odd, even or none), the number of stop bits (1 or 0) and the handshaking (RTS/CTS or none). The default setting is for 8-data bits, no parity, 1 stop bit and no handshaking that is typical for most applications. These settings apply to all the RS-232 ports but are not applicable for the Telnet ports. The handshaking parameter is not applicable for the RS-485 ports.

See Also

```
SetBaud(string sBaud)
```

Example

The following example configures RS-232 port 4 for 7 data bits, even parity, 1 stop bit, and no handshaking

```
HubInit()
{
  ConfigPort(3, “7,e,1”, 0);
}
```

ConfigSendUDP

Syntax

```
ConfigSendUDP(float fPort, string sRemoteIP, float fDestPort)
```

Arguments

type	name	description
float	fPort	the number of the UDP port we are configuring 14, 15, 16 or 17
string	sRemoteIP	the IP address of the remote UDP port.
		An IP address of “255.255.255.255” will broadcast to all UDP ports.
float	fDestPort	is the UDP port number at the destination

Returns

None.

Purpose

This function configures how and where a FreeWay UDP port sends its data. There are 4 FreeWay UDP ports available: 14 thru 17. For each of the FreeWay ports you can specify a single IP address to send to (when you want to communicate with a single remote FreeWay for example). If you specify the broadcast address(255.255.255.255) the FreeWay will send a message to all open UDP ports on the network. You must also specify which UDP port you want to send the message to.

See Also

```
OpenUDP(float fPort, float fListenPort)

SerialReceive(float fPort)

SerialSend(string sOutput)
```

Example

The following example configures FreeWay port 14 to communicate with UDP port 5600 on a remote FreeWay with IP Address 192.168.7.110. It then send a message to the remote FreeWay.

```
HubInit()
{
  OpenUDP(14, 5600);
  ConfigSendUDP(14, “192.168.7.110”, 5600);
  SerialSend(14, “Hello”);
}
```

ConfigTelnet

Syntax

```
ConfigTelnet(float fPort, float fMode)
```

Arguments

type	name	description
float	fPort	the number of the Telnet port we are configuring 10, 11, 12 or 13
float	fMode	the Telnet mode. 0 is for Telnet mode (default), 1 is for raw TCP mode

Returns

None.

Purpose

This function configures how a FreeWay Telnet port sends & receives its data. There are 4 FreeWay Telnet ports available: 10 thru 13. For each of the FreeWay Telnet ports you can specify whether it acts as a Telnet interface or as a raw TCP interface. For a given port **you need to call ConfigTelnet() before you make a call to OpenTelnet()**.

See Also

```
OpenTelnet(float fPort, float fCS, float fIPport string sIPaddr)

CloseTelnet(float fPort)

string SerialGet(float fPort)

SerialReceive(float fPort)

SerialSend(string sOutput)
```

Example

```
HubInit()
{
  // Open a TCP client port to a TCP server on 192.168.1.27 port 6789
  ConfigTelnet(10,1)
  OpenTelnet(10, 0, 6789, “192.168.1.27”);

  // send a message to it
  string sMessage;
  sMessage = “Hello\n\r”;
  SerialSend(10, sMessage);
}
```

DebugPrint

Syntax

```
DebugPrint(string sDebug)
```

Arguments

type	name	description
string	sDebug	the message to print out to the debug port

Returns

None.

Purpose

This is a very useful function for debugging your script. It will print out the string defined in `sDebug` to the Telnet debug interface if it is enabled. The function will be ignored if the debug interface is not active.

See Also

Example

```
HubInit()
{
  // just print a string
  DebugPrint("We are in HubInit() \n\r");

  // add a text value to the string
  string sMessage;
  string sDay;
  sDay = "Monday";
  sMessage = "Today is " + sDay + "\n\r";
  DebugPrint(sMessage);    // prints "Today is Monday"
}
```

Delayms

Syntax

```
Delayms(float fDelay)
```

Arguments

type	name	description
float	fDelay	The amount of time to wait in milliseconds (0-5000)

Returns

None.

Purpose

This function introduces a delay as specified by the `fDelay` argument in milliseconds. Up to 5 seconds of delay can be achieved. You should use this function with some care. If you stay in a function too long with `Delayms()` then the FreeWay will not be able to process the system events such as alarms & serial receive.

See Also

None

Example

```
HubInit()
{
  // wait for 2.5 seconds
  Delayms(2500);
}
```

DigitalIn

Syntax

```
DigitalIn(float fDinPort)
```

Arguments

type	name	description
float	fDinPort	The digital input port (1-6) on which a change of state was detected

Returns

None.

Purpose

This is a system event function that is automatically called when a change of state is detected on a Digital Input. The digital input that has changed is specified in the `fDinPort` parameter. To get the actual state of the Digital Input call the `GetDin()` function that should be called as soon as possible inside DigitalIn().

Note that the Digital Inputs are only available when a FreeWay is fitted with an IR option card.

See Also

```
float GetDin(float fDinPort)
```

Example

```
DigitalIn(float fDinPort)
{
  // get the state of the digital input
  float fDinState;
  fDinState = GetDin(fDinPort);

  // print out a message
  string sMessage;
  sMessage="Digital In "+format(fDinPort,1,0)+"state = "+format(fDinState,1,0);
  DebugPrint(sMessage);  // e.g. "Digital In 1 state = 0"
}
```

EIBPhysical

Syntax

```
EIBPhysical(string sEibPhysical)
```

Arguments

type	name	description
string	sEibPhysical	the FreeWay EIB physical address e.g. “0.0.1”

Returns

None.

Purpose

This function configures the FreeWay’s EIB Physical address. This is the physical address used in ETS for the FreeWay’s dummy device.

See Also

```
EIBRegister(float fEibRegHandle, string sEibRegGroup, float fEibRegType)
EIBReceive(float fEibRxHandle, float fEibRxValue, string sEibRxValue)
```

Example

The following example configures FreeWay to have a physical address of 0.1.2

```
HubInit()
{
  EIBPhysical(“0.1.2”);
}
```

EIBRegister

Syntax

```
EIBRegister(float fEibRegHandle, string sEibRegGroup, float fEibRegType)
```

Arguments

type	name	description
float	sEibRegHandle	a unique handle to identify the EIB group from 1 to 64
string	sEibRegGroup	the EIB group address that you want to register e.g. “0/0/1” or “0/3”
float	fEibType	the type of Datapoint that the Group uses. Options are:

Type	Data Type	Format
1	Boolean	1-bit
3	3-bit Controlled	4-bit
4	Character Set	8-bit
5	8-Bit Unsigned Value	8-bit
6	8-bit Signed Value	8-bit
7	2-octet Unsigned Value	2-octet
8	2-octet Signed Value	2-octet
9	2-octet Float Value	2-octet
10	Time	3-octet
11	Date	3-octet
12	4-octet Unsigned Value	4-octet
13	4-octet Signed Value	4-octet
14	4-octet Float Value	4-octet
15	Access	4-octet

Returns

None.

Purpose

This function registers an EIB group address with FreeWay. This allows the FreeWay to read from and write to this group address. Each group that you register must have its own unique handle, fEibRegHandle, which is used by other functions to refer to registered groups. The handle must be between 1 and 64. The group address can be in 3-level (“0/1/2”) or 2-level (“0/2”) format. The Data Type of the group must also be specified.

You can re-register a group at any time within a script. You can also un-register a group at any time by setting the Type to 0.

See Also

`EIBPhysical(string sEibPhysical)`

`EIBReceive(float fEibRxHandle, float fEibRxValue, string sEibRxValue)`

`EIBSendFloat(float fEibTxfHandle, float fEibTxfValue)`

`EIBSendString(float fEibTxfHandle, string sEibTxsValue)`

Example

The following example configures FreeWay to have a physical address of 0.1.2, and registers a Type 1 (Boolean) group.

```
HubInit()
{
  EIBPhysical(“0.1.2”);

  // Group Handle = 1
  // Group Address = 0/1/2
  // Data Type = 1 = Boolean
  EIBRegister(1,”0/1/2”,1);
}
```

EIBReceive

Syntax

```
EIBReceive(float fEibRxHandle, float fEibRxValue, string sEibRxValue)
```

Arguments

type	name	description
float	fEibRxHandle	the group handle previously registered with EIBRegister()
float	fEibRxValue	the floating point value of the data received from the group
float	sEibRxValue	a string version of the data received from the group

Returns

None.

Purpose

This system event function is automatically called when a message is received from an EIB group previously registered using EIBRegister(). The function is passed a handle, fEibRxHandle, which can be used to determine which registered group sent a message. The function is also passed the message data in both floating point and string formats. This provides flexibility in how the message is interpreted by the script.

See Also

```
EIBRegister(float fEibRegHandle, string sEibRegGroup, float fEibRegType)
EIBPhysical(string sEibPhysical)
EIBSendFloat(float fEibTxfHandle, float fEibTxfValue)
EIBSendString(float fEibTxfHandle, string sEibTxsValue)
```

Example

The following example receives a Boolean message from a group and prints a status message to the debug screen.

```
HubInit()
{
  EIBPhysical("0.1.2");

  // Group Handle = 1
  // Group Address = 0/1/2
  // Data Type = 1 = Boolean
  EIBRegister(1,"0/1/2",1);
```

```
}

EIBReceive(float fEibRxHandle, float fEibRxValue, string sEibRxValue)
{
  // check the group's handle
  if (fEibRxHandle == 1)
  {
    // Read the value as a string....
    if (sEibRxValue[0] == 0) {DebugPrint("Value = On");}
    if (sEibRxValue[0] == 1) {DebugPrint("Value = Off");}

    // ... or Read the value as a float
    if (fEibRxValue == 0) {DebugPrint("Value = On");}
    if (fEibRxValue == 1) {DebugPrint("Value = Off");}
  }
}
```

EIBSendFloat

Syntax

```
EIBSendFloat(float fEibTxfHandle, float fEibTxfValue)
```

Arguments

type	name	description
float	fEibTxfHandle	the group handle previously registered with EIBRegister()
float	fEibTxfValue	the floating point value of the data to send to the group

Returns

None.

Purpose

This function is used to send a floating point or integer value to an EIB group. The group must have been previously registered using the EIBRegister() function.

See Also

```
EIBRegister(float fEibRegHandle, string sEibRegGroup, float fEibRegType)

EIBPhysical(string sEibPhysical)

EIBReceive(float fEibRxHandle, float fEibRxValue, string sEibRxValue)

EIBSendString(float fEibTxfHandle, string sEibTxsValue)
```

Example

The following example sends a Boolean float message to a group.

```
HubInit()
{
  EIBPhysical("0.1.2");

  // Group Handle = 1
  // Group Address = 0/1/2
  // Data Type = 1 = Boolean
  EIBRegister(1,"0/1/2",1);

  // send a value of 1 to the group
  EIBSendFloat(1, 1)
}
```

EIBSendString

Syntax

```
EIBSendString(float fEibTxsHandle, string sEibTxsValue)
```

Arguments

type	name	description
float	fEibTxsHandle	the group handle previously registered with EIBRegister()
string	sEibTxsValue	the string value of the data to send to the group

Returns

None.

Purpose

This function is used to send a string value to an EIB group. The group must have been previously registered using the EIBRegister() function.

See Also

```
EIBRegister(float fEibRegHandle, string sEibRegGroup, float fEibRegType)

EIBPhysical(string sEibPhysical)

EIBReceive(float fEibRxHandle, float fEibRxValue, string sEibRxValue)

EIBSendFloat(float fEibTxfHandle, float fEibTxfValue)
```

Example

The following example sends a Boolean string message to a group.

```
string sEibString;

HubInit()
{
  EIBPhysical(“0.1.2”);

  // Group Handle = 1
  // Group Address = 0/1/2
  // Data Type = 1 = Boolean
  EIBRegister(1,”0/1/2”,1);

  // send a value of 1 to the group
  sEibString[0] = 1;
  EIBSendString(1, sEibString);
}
```

format

Syntax

```
string format(float fNum, float fFormat, float x)
```

Arguments

type	name	description
float	fNum	The value to convert into a string
Float	fFormat	The type of the value to convert
		1 Integer
		2 Float
Float	x	Reserved for future use – set to 0

Returns

The converted string value.

Purpose

This function will convert a numeric value to a string. This is useful for outputting numeric values to the debug port, for example.

See Also

```
string midstr(string sSrc, float fStart, float fLength)

float strlen(string sSrc)
```

Example

```
HubInit()
{
  string sMessage;

  // convert an integer value to string
  string sInteger;
  sInteger = format(23, 1, 0);
  sMessage = "Integer value is " + sInteger + "\n\r";
  DebugPrint(sMessage);  // "Integer value is 23"

  // convert a float value to string
  string fFloatValue;
  fFloatValue = format(1.234, 2, 0);
  sMessage = "Float value is " + fFloatValue + "\n\r";
  DebugPrint(sMessage);  // "Float value is 1.234"
}
```

GetDate

Syntax

```
string GetDate()
```

Arguments

None.

Returns

Returns a string formatted with the current date as “dd/mm/yyyy”.

Purpose

Use this function to get the current time in string format.

See Also

```
string GetTime()
```

Example

```
HubInit()
{
  string sDate;
  sDate = GetDate();

  string sMessage;
  sMessage = “The date is ” + sDate + “\n\r”;
  DebugPrint(sMessage);  // e.g. “The date is 27/11/2004”
}
```

GetDin

Syntax

```
float GetDin(float fDinPort)
```

Arguments

type	name	description
float	fDinPort	The digital input port (1-6) whose state we want to read

Returns

Return the state of the Digital Input, either 0 or 1.

Purpose

This function will return the current state of a Digital Input. It is normally called within the `DigitalIn()` system event function that informs us that a change of state has occurred on one of the digital inputs. The state returned is either 0 or 1.

Note that the Digital Inputs are only available when a FreeWay is fitted with an IR option card.

See Also

```
DigitalIn(float fDinPort)
```

Example

```
DigitalIn(float fDinPort)
{
  // get the state of the digital input
  float fDinState;
  fDinState = GetDin(fDinPort);

  // print out a message
  string sMessage;
  sMessage="Digital In "+format(fDinPort,1,0)+"state = "+format(fDinState,1,0);
  DebugPrint(sMessage);  // e.g. "Digital In 1 state = 0"
}
```

GetTime

Syntax

```
string GetTime()
```

Arguments

None.

Returns

Returns a string formatted with the current time.

Purpose

Use this function to get the current time in string format as “hh:mm”

See Also

```
string GetDate()
```

Example

```
HubInit()
{
  string sTime;
  sTime = GetTime();

  string sMessage;
  sMessage = “The time is ” + sTime + “\n\r”;
  DebugPrint(sMessage);  // e.g. “The time is 16:03”
}
```

HubInit

Syntax

```
HubInit()
```

Arguments

None.

Returns

None

Purpose

This system event function is always called when the FreeWay powers up. It should be used for initialising all the relevant ports and variables for your application.

See Also

Example

```
HubInit()
{
}
```

IRReceive

Syntax

```
IRReceive(float fAddress, float fData, float fToggle)
```

Arguments

type	name	description
float	fAddress	The address field of the received RC5 Infra-Red code
		Values from 0 to 31
float	fData	The data field of received RC5 Infra-Red code
		Values from 0 to 63
float	fToggle	The state of the toggle bit of the received RC5 Infra-Red code
		Values are 0 or 1

Returns

None.

Purpose

This system event function is automatically called when a valid RC5 protocol Infra-Red message is received via the FreeWay's Infra-Red receiver on the front or rear panel. The function provides you with the Address, Data, and Toggle bit of the RC5 message. Refer to the FreeWay User Guide for more information on RC5 and using this function.

See Also

Example

```
IRReceive(float fAddress, float fData, float fToggle)
{
  // check the address field first - we will respond to address 31
  if (fAddress == 31)
  {
    // check the data field next - we will respond to data value 20
    if (fData == 20)
    {
      // Print out a message to the debug port
      DebugPrint("Got RC5 Address 31 Data 20 \n\r");
    }
  }
}
```

midstr

Syntax

```
string midstr(string sSrc, float fStart, float fLength)
```

Arguments

type	name	description
string	sSrc	The source string to process
float	fStart	The start point for the new string (0 is the first character)
float	fLength	The length of the new string

Returns

The converted string value.

Purpose

This function will extract a sub-string from a string. The original string is in `sSrc`. The start point of the sub-string you want to extract is defined in `fStart` and the length of the sub-string is in `fLength`.

See Also

```
string format(float fNum, float fFormat, float x)

float strlen(string sSrc)
```

Example

```
HubInit()
{
  string sMessage;

  // here's the original string
  sMessage = "This is a long string";

  // Here's how we would extract the words 'long string'
  string sNewString;
  sNewString = midstr(sMessage,10,11);
  DebugPrint(sNewString);  // "long string"
}
```

OpenTelnet

Syntax

```
OpenTelnet(float fPort, float fCS, float fIPport string sIPaddr)
```

Arguments

type	name	description
float	fPort	specifies one of four the Freeway ports. This can be 10, 11, 12 or 13
float	fCS	specifies if the port is a Client or Server port. 0 is for client 1 is for server
float	fIPport	the TCP/IP port number of Telnet port on the remote equipment. E.g. 6789
string	sIPaddr	the IP address of the remote equipment e.g. “192.168.7.23”

Returns

None.

Purpose

Before using a Telnet port it needs to be opened using this function. The equipment should be connected and enabled before using this function. It is OK if the connection is already open when you call `OpenTelnet()`. It is also possible to open multiple Telnet ports using the same TCP/IP port number.

Note that this Telnet interface is totally separate from the one used for debugging & configuration – that interface does NOT need to be setup using this function.

See Also

```
ConfigTelnet(float fPort, float fMode)

CloseTelnet(float fPort)

string SerialGet(float fPort)

SerialReceive(float fPort)

SerialSend(string sOutput)
```

Example

```
HubInit()
{
  // Open a Telnet client port to a Linn Kivor Jukebox
  OpenTelnet(10, 0, 6789, “192.168.1.27”);
  // send a message to it
```

```
  string sMessage;
  sMessage = “Hello Kivor”;
  SerialSend(10, sMessage);
}
```

OpenUDP

Syntax

```
OpenUDP(float fPort, float fListenPort)
```

Arguments

type	name	description
float	fPort	specifies one of four the Freeway ports. This can be 14, 15, 16 or 17
float	fListenPort	specifies the UDP port that FreeWay will listen to. E.g. 3456.

Returns

None.

Purpose

Before using a UDP port it needs to be opened using this function.

See Also

```
string SerialGet(float fPort)

SerialReceive(float fPort)

SerialSend(string sOutput)

ConfigSendUDP(float fPort, string sRemoteIP, float fDestPort)
```

Example

The following example configures FreeWay port 14 to communicate with UDP port 5600 on a remote FreeWay with IP Address 192.168.7.110. It then send a message to the remote FreeWay.

```
HubInit()
{
  OpenUDP(14, 5600);
  ConfigSendUDP(14, “192.168.7.110”, 5600);
  SerialSend(14, “Hello”);
}
```

sendIR

Syntax

```
sendIR(float fIR1, float fIR2, float fIR3, float fIR4, string sIROut)
```

Arguments

type	name	description
float	fIR1	0 or 1, set to 1 to enable Infra-Red output on IR output port IR1
float	fIR2	0 or 1, set to 1 to enable Infra-Red output on IR output port IR2
float	fIR3	0 or 1, set to 1 to enable Infra-Red output on IR output port IR3
float	fIR4	0 or 1, set to 1 to enable Infra-Red output on IR output port IR4
string	sIROut	the Infra-Red command to transmit

Returns

None.

Purpose

This function is used to transmit Infra-Red commands to the Infra-Red output ports IR1-4. You can output the command to any combination of the four IR outputs using the enable flags `fIR1-4`. The command that you send should be captured using the FreeWay's Infra-Red Learn facility.

Note that the Infra-Red outputs are only available when a FreeWay is fitted with an IR option card.

See Also

Example

```
HubInit()
{
  // transmit an IR command on IR ports 1 & 2
  string sIRCommand;
  sIRCommand = "[PF68L836741E0083418C3X42F7BDEFF7FFFFB2F7BDEFF7FFFFB0P2CDFR04]";
  sendIR(1,1,0,0,sIRCommand);
}
```

SerialGet

Syntax

```
string SerialGet(float fPort)
```

Arguments

type	name	description
float	fPort	the port number of the serial port to get a message from
		1-6 RS-232 ports 1-6
		7,8 RS-485 ports 7 & 8
		9 RS-485 RJ45 port
		10-13 Telnet ports
		14-17 UDP ports

Returns

A string containing the message received from the serial, Telnet or UDP ports.

Purpose

This function will return, as a string, the message received from the serial, Telnet or UDP port specified in `fPort`. This function should only be called after a `SerialReceive()` function that tells us that a message has been received on a particular port. We then use the `SerialGet()` function to get the message from that port.

The message that is returned will largely depend on how we have defined our message structure with the `SetLength()`, `SetTimeout()` and `SetEndOfMsg()` functions. By default, messages will be delivered if there are no further bytes received for 100ms, or if the message length exceeds 256 characters.

See Also

`SerialReceive(float fPort)`

`SetMsgLength(float fPort, float fLength)`

`SetMsgTimeout(float fPort, float fTime)`

`SetEndOfMsg(float fPort, float fEom)`

Example

```
// system event function called when a message is received on a port
SerialReceive(float fPort)
{
  // get the string from the port
  string sMessage;
  sMessage = SerialGet(fPort);

  // print out the string
  string sDebug;
  sDebug = “Got message: ” + sMessage + “ on port: ” + format(fPort,1,0) + “\n\r”;
  DebugPrint(sDebug);  // e.g. “Got message: hello on port: 1”
}
```

SerialReceive

Syntax

```
SerialReceive(float fPort)
```

Arguments

type	name	description
float	fPort	the port number of the serial port that has received a message
		1-6 RS-232 ports 1-6
		7,8 RS-485 ports 7 & 8
		9 RS-485 RJ45 port
		10-13 Telnet ports
		14-17 UDP Ports

Returns

None.

Purpose

This system event function is automatically called when a message is received on one of the serial, Telnet or UDP ports. The port is specified in the `fPort` argument. We then use the `SerialGet()` function to get the message from that port.

See Also

`string SerialGet(float fPort)`

`SetMsgLength(float fPort, float fLength)`

`SetMsgTimeout(float fPort, float fTime)`

`SetEndOfMsg(float fPort, float fEom)`

Example

```
// system event function called when a message is received on a port
SerialReceive(float fPort)
{
  // get the string from the port
  string sMessage;
  sMessage = SerialGet(fPort);

  // print out the string
  string sDebug;
  sDebug = “Got message: ”+sMessage+“ on port: ”+format(fPort,1,0) + “\n\r”;
  DebugPrint(sDebug);  // e.g. “Got message: hello on port: 1”
}
```

SerialSend

Syntax

```
SerialSend(float fPort, string sOutput)
```

Arguments

type	name	description
float	fPort	the port number we are sending a message to 1-6 RS-232 ports 1-6 7,8 RS-485 ports 7 & 8 9 RS-485 RJ45 port 10-13 Telnet ports 14-17 UDP Ports
string	sOutput	the string to send to the port

Returns

None.

Purpose

This function will transmit the string defined in `sOutput` to the port specified in `fPort`.

See Also

```
string SerialGet(float fPort)

SerialReceive(float fPort)
```

Example

```
HubInit()
{
  // RS-232 port 2 operates 2400 bits per second
  SetBaud(2, “2400”);

  // define a message
  string sMessage;
  sMessage = “This is port 2”;

  // send the message
  SerialSend(2, sMessage);
}
```

SetAlarm

Syntax

```
SetAlarm(float fAlarm, string sConfig)
```

Arguments

type	name	description	
float	fAlarm	the number of the alarm event (1 to 8) to set up	
string	sConfig	one of five different alarm configuration strings	
		“O dd/mm/yyyy hh:mm”	One-time alarm on this date/time
		“D hh:mm”	Daily alarm at this time
		“W dd/mm/yyyy hh:mm”	Weekly alarm starting from this day/time
		“A +/-hh:mm”	Alarm at dawn + or – an offset
		“P +/-hh:mm”	Alarm at dusk + or – an offset

Returns

None.

Purpose

This function is used to configure one of up to eight alarm events that are handled with the `Alarm()` system event function. Alarm events can be setup to occur:

- Once only
- Daily
- Weekly
- At dawn or dusk

To use the dawn and dusk alarms you must also specify the location using SetLocation(). The maximum offsets for dusk and dawn alarms are 6 hours.

See Also

```
Alarm(float fAlarm)
SetLocation(float fLat, float fLon)
```

Example

The following example sets up four different alarms in `HubInit()`.

```
HubInit()
{
  // alarm 1 is a one time alarm on 27th November 2004 at 9 in the morning
  SetAlarm(1, “O 27/11/2004 09:00”);

  // alarm 2 is a daily alarm at 9 at night
  SetAlarm(2, “D 21:00”);

  // alarm 3 is a weekly alarm at 12 noon after 1st January 2004
  SetAlarm(3, “W 01/01/2004 12:00”);

  // alarm 4 is a dawn alarm set for 1 hour after dawn
  // The location is London’s Oxford Street
  SetLocation(51.51, -0.148);
  SetAlarm(4, “A +01:00”);
}
```

SetBaud

Syntax

```
SetBaud(float fPort, string sBaud)
```

Arguments

type	name	description
float	fPort	the port number of the serial port we are configuring
		1-6 RS-232 ports 1-6
		7,8 RS-485 ports 7 & 8
		9 RS-485 RJ45 port
string	sBaud	the baud rate for the port
		for RS-232 300, 600, 1200, 2400, 4800, 9600, 19200, 38400, 57600, 115200
		for RS-485 600, 1200, 2400, 4800, 9600, 19200, 38400, 57600, 115200, 230400

Returns

None.

Purpose

This function sets the baud rate for a serial port. This function must be called before using a serial port or the baud rate and port operation will be undefined. Note that the baud rate must be a string enclosed in quotes e.g. “9600”.

See Also

```
ConfigPort(float fPort, string sConfig, float fHandshake)
```

Example

```
HubInit()
{
  // RS-232 port 1 operates 9600 bits per second
  SetBaud(1, “9600”);
}
```

SetDout

Syntax

```
SetDout(float fDout, float fState)
```

Arguments

type	name	description
float	fDout	The digital output port (1-6) whose state we want to change
float	fState	The new state of the digital output port (0 or 1)

Returns

None

Purpose

This function changes the state of one of the 6 digital output ports to 0 or 1. Note that the Digital Outputs are only available on when a FreeWay is fitted with an IR option card.

See Also

```
DigitalIn(float fDinPort)

float GetDin(float fDinPort)
```

Example

```
ChangeDigitalOut()
{
  // set the state of digital output 2 to a 1
  SetDout(2,1);
}
```

SetEndOfMsg

Syntax

```
SetEndOfMsg(float fPort, float fEom)
```

Arguments

type	name	description
float	fPort	the port number of the port we are configuring
		1-6 RS-232 ports 1-6
		7,8 RS-485 ports 7 & 8
		9 RS-485 RJ45 port
		10-13 Telnet ports
float	fEom	the character that defines the end of a received message
		0 (default) disables this feature.

Returns

None.

Purpose

Depending on the communications protocol of the equipment you are communicating with, the messages received from that equipment may always end with a certain character, perhaps a Carriage Return for ASCII protocols. The SetEndOfMsg() function lets you define that character so that the `SerialReceive()` and `SerialGet()` functions will provide you with a full message to process. By default this feature is disabled (`fEom = 0`).

Note that the `fEom` character is defined as a `float`. Refer to the ASCII table at the end of this manual to get the decimal values of ASCII characters. For example, a Carriage Return is 13.

Note that this function only defines how *received* messages are handled. For messages that you transmit, it's up to you to add the correct end-of-message character as part of your transmission.

See Also

```
string SerialGet(float fPort)
SerialReceive(float fPort)
SetMsgLength(float fPort, float fLength)
SetMsgTimeout(float fPort, float fTime)
```

Example

```
HubInit()
{
  // RS-232 port 3 operates 4800 bits per second
  SetBaud(3, “4800”);

  // port 3’s messages end with a Carriage Return
  SetEndOfMsg(3, 13);
}
```

SetHubLed

Syntax

```
SetHubLed(float fLed, float fState)
```

Arguments

type	name	description
float	fLed	the number of the front panel LED, 1 to 4 from left to right
float	fState	state of the LED, 1 is ON, 0 is OFF.
		A value of 2 puts the LED into system mode (see below)

Returns

None.

Purpose

This function lets you change the state of the front panel LEDs. These can be useful for a variety of debugging uses and as status indications. By default the LEDs are in system mode where they take on specific status indications:

LED 1	Power on	LED 2	IR Learn active
LED 3	Script activity	LED 4	Serial port activity

The system mode for each LED can be overridden by writing a 1 or 0 value to the LED. Write a 2 value to the LED to return it to system mode.

See Also

Example

```
HubInit()
{
  // Turn off LED 1 at start of HubInit
  SetHubLed(1, 0);

  // rest of HubInit function here..

  // Return LED 1 to system mode at end of HubInit
  SetHubLed(1, 2);
}
```

SetMsgLength

Syntax

```
SetMsgLength(float fPort, float fLength)
```

Arguments

type	name	description
float	fPort	the port number of the port we are configuring
		1-6 RS-232 ports 1-6
		7,8 RS-485 ports 7 & 8
		9 RS-485 RJ45 port
		10-13 Telnet ports
float	fLength	defines the length of the message in bytes
		If set to 0 (default) the feature is disabled.

Returns

None.

Purpose

Depending on the communications protocol of the equipment you are communicating with, the messages received from that equipment may always be a fixed length. The `SetMsgLength()` function lets you define that length so that the `SerialReceive()` and `SerialGet()` functions will provide you with a full message to process. By default this feature is disabled (`fLength = 0`).

See Also

`string SerialGet(float fPort)`

`SerialReceive(float fPort)`

`SetEndOfMsg(float fPort, float fEom)`

`SetMsgTimeout(float fPort, float fTime)`

Example

```
HubInit()
{
  // RS-232 port 4 operates 9600 bits per second
  SetBaud(4, "9600");
  // port 4's messages are always 10 bytes long
  SetEndOfMsg(4, 10);
}
```

SetMsgTimeout

Syntax

```
SetMsgTimeout(float fPort, float fTime)
```

Arguments

type	name	description	
float	fPort	the port number of the port we are configuring	
		1-6	RS-232 ports 1-6
		7,8	RS-485 ports 7 & 8
		9	RS-485 RJ45 port
		10-13	Telnet ports
float	fTime	defines the period of time (in milliseconds) after which, if no further bytes have been received, the SerialReceive() function will be called. The default is 100ms.	

Returns

None.

Purpose

In order to receive a full protocol message from the `SerialReceive()` and `SerialGet()` functions, this function lets you define a period of time after which, if no further bytes are received by the port, the message will be presented.

See Also

`string SerialGet(float fPort)`

`SerialReceive(float fPort)`

`SetEndOfMsg(float fPort, float fEom)`

`SetMsgLength(float fPort, float fLength)`

Example

```
HubInit()
{
  // RS-232 port 5 operates 9600 bits per second
  SetBaud(5, “9600”);

  // present port 5’s messages after 500ms have elapsed
  SetTimeout(5, 500);
}
```

SetLocation

Syntax

```
SetLocation(float fLat, float fLon)
```

Arguments

type	name	description
float	fLat	the Latitude value
float	fLon	the Longitude value

Returns

None.

Purpose

This function is used to specify the FreeWay's location so that it can calculate the dawn and dusk alarm times. You can obtain these values from any online map service, for example: www.multimap.com.

See Also

`Alarm(float fAlarm)`

`SetAlarm(float fAlarm, sConfig)`

Example

The following example sets a dawn alarm in `HubInit()`.

```
HubInit()
{
  // alarm 1 is a dawn alarm set for 1 hour before dawn
  // The location is London's Oxford Street
  SetLocation(51.51, -0.148);
  SetAlarm(1, "A -01:00");
}
```

strlen

Syntax

```
float strlen(string sSrc)
```

Arguments

type	name	description
String	sSrc	The source string to process

Returns

The length of the source string in bytes.

Purpose

This function returns the length of the source string.

See Also

```
string format(float fNum, float fFormat, float x)

string midstr(string sSrc, float fStart, float fLength)
```

Example

```
HubInit()
{
  string sMessage;

  // here's the original string
  sMessage = "This is a long string";

  // Now we get the length
  float fStringLength;
  fStringLength = strlen(sMessage);

  // print out a message
  sMessage = "String length = " + format(fStringLength,1,0) + " bytes\n\r";
  DebugPrint(sMessage);  // "String length = 21 bytes"
}
```

SECTION 3 – QUICK REFERENCE

ASCII Table

Operator Table

System Function Reference

ASCII Table

DEC	HX	OCT	Sym
000	00	000	NUL
001	01	001	SOH
002	02	002	STX
003	03	003	ETX
004	04	004	EOT
005	05	005	ENQ
006	06	006	ACK
007	07	007	BEL
008	08	010	BS
009	09	011	TAB
010	0A	012	NL
011	0B	013	VT
012	0C	014	NP
013	0D	015	CR
014	0E	016	SO
015	0F	017	SI
016	10	020	DLE
017	11	021	DC1
018	12	022	DC2
019	13	023	DC3
020	14	024	DC4
021	15	025	NAK
022	16	026	SYN
023	17	027	ETB
024	18	030	CAN
025	19	031	EM
026	1A	032	SUB
027	1B	033	ESC
028	1C	034	FS
029	1D	035	GS
030	1E	036	RS
031	1F	037	US

DEC	HX	OCT	Sym
032	20	040	SP
033	21	041	!
034	22	042	"
035	23	043	#
036	24	044	$
037	25	045	%
038	26	046	&
039	27	047	'
040	28	050	(
041	29	051	)
042	2A	052	*
043	2B	053	+
044	2C	054	,
045	2D	055	-
046	2E	056	.
047	2F	057	/
048	30	060	0
049	31	061	1
050	32	062	2
051	33	063	3
052	34	064	4
053	35	065	5
054	36	066	6
055	37	067	7
056	38	070	8
057	39	071	9
058	3A	072	:
059	3B	073	;
060	3C	074	<
061	3D	075	=
062	3E	076	>
063	3F	077	?

DEC	HX	OCT	Sym
064	40	100	@
065	41	101	A
066	42	102	B
067	43	103	C
068	44	104	D
069	45	105	E
070	46	106	F
071	47	107	G
072	48	110	H
073	49	111	I
074	4A	112	J
075	4B	113	K
076	4C	114	L
077	4D	115	M
078	4E	116	N
079	4F	117	O
080	50	120	P
081	51	121	Q
082	52	122	R
083	53	123	S
084	54	124	T
085	55	125	U
086	56	126	V
087	57	127	W
088	58	130	X
089	59	131	Y
090	5A	132	Z
091	5B	133	[
092	5C	134	\
093	5D	135	]
094	5E	136	^
095	5F	137	_

DEC	HX	OCT	Sym
096	60	140	`
097	61	141	a
098	62	142	b
099	63	143	c
100	64	144	d
101	65	145	e
102	66	146	f
103	67	147	g
104	68	150	h
105	69	151	i
106	6A	152	j
107	6B	153	k
108	6C	154	l
109	6D	155	m
110	6E	156	n
111	6F	157	o
112	70	160	p
113	71	161	q
114	72	162	r
115	73	163	s
116	74	164	t
117	75	165	u
118	76	166	v
119	77	167	w
120	78	170	x
121	79	171	y
122	7A	172	z
123	7B	173	{
124	7C	174	|
125	7D	175	}
126	7E	176	~
127	7F	177	DEL

DEC – decimal value

HX – hexadecimal value

OCT – octal value

Sym – ASCII symbol

Operator Table

Operator	Operation	Example
=	Sets a value	x = 3 y = “hello”
+	Simple addition	x = x + 2
-	Simple subtraction	x = x-2
*	Simple Multiplication	x = x * 2
/	Simple Division	x = x/2
%	Remainder	x = y%2
==	Comparator	if (x==2)
!=	Inequality	if (x != 2)
&&	Logical AND	if (x && b) { c=1; }
||	Logical OR	if (x || b) { c=1; }
!	Logical NOT	x = !y; if (!x) { c=1; }
&	Bitwise AND	z = x & y;
|	Bitwise OR	z = x | y;
>	Greater than	if (x > y) { c=1; }
<	Less than	if (x < y) { c=1; }
>=	Greater than or equal to	if (x >= y) { c=1; }
<=	Less than or equal to	if (x => y) { c=1; }

System Functions

Function	Parameters	Meaning	Description
CloseTelnet(float fPort)	fPort	Freeway port number 10, 11, 12 or 13	Clsoe telnet port
ConfigPort(float fPort, string sConfig, float fHandshake)	fPort sConfig fHandshake	1-6 for RS232; 7-9 for RS485 “data bits (7, 8), parity (o,e,n), stop bits (0,1)” 0 – no handshaking, 1 – RTS/CTS handshaking	Advanced serial port settings for data bits, parity, stop bits and handshaking
ConfigSendUDP(float fPort, string RemoteIP, float fDestPort)	fPort sRemoteIP fDestPort	FreeWay port number 14, 15, 16 or 17 UDP address to connect to e.g. “192.168.1.101” Remote UDP port number (0-65535)	Configures the transmission settings for UDP ports
ConfigTelnet(float fPort, float fMode)	fPort fMode	FreeWay port number 10, 11, 12 or 13 0 is Telnet, 1 is TCP	Configures a Telnet port as Telnet or raw TCP
DebugPrint(string sDebug)	sDebug	string to be output	Outputs a string to the Telnet interface when in debug mode.
Delayms(float fDelay)	fDelay	number of milli-seconds to wait (0-5000)	Wait for `fDelay` number of milli-seconds.
EIBRegister(float fEibRegHandle, string sEibRegGroup, float fEibRegType)	fEibRegHandle sEibRegGroup fEibRegType	Handle of the EIB group to register, from 1 to 64 The EIB Group Address to register e.g. “1/2/3” The DataPoint Type of the Group, from 0 to 16	Register an EIB group

Function	Parameters	Meaning	Description
EIBSendFloat(float fEibTxfHandle, float fEibTxfValue)	fEibTxfHandle fEibTxfValue	Handle of the registered group from 1 to 64 Value to send	Send a floating point or integer value to the EIB bus
EIBSendString(float fEibTxsHandle, string sEibTxsValue)	fEibTxsHandle sEibTxsValue	Handle of the registered group from 1 to 64 Value to send	Send a string value or value array to the EIB bus
string format(float fNum, float fFormat, float x)	fNum fFormat x	number to convert to string format of string: 1 - integer; 2 - float (reserved for future use)	Convert a number into a string.
GetDin(float fDinPort)	fDinPort	the Digital Input port (1 – 6)	Returns the current state of the digital input fDinPort
string GetTime()			Returns the current time as “hh:mm”
string midstr(string sSrc, float fStart, float fLength)	sSrc fStart fLength	input string start point for new string (0 is first character) length of new string	Create a new string from a section of the input string
OpenTelnet(float fPort, string sIpaddr)	fPort fCS fIPport sIPaddr	Freeway port number 10, 11, 12 or 13 Specifies a client(0) or server(1) port IP port number (0-65535) IP address to connect to e.g. “192.168.1.101”	Open telnet port, this must be done before calling e.g. “SerialSend(10,ssss)”

Function	Parameters	Meaning	Description
sendIR(float fIR1, float fIR2, float fIR3, float fIR4, string sIROut)	fIR1 fIR2 fIR3 fIR4 sIROut	1/0 enables/disables infra red transmit on output port IR1 1/0 enables/disables infra red transmit on output port IR2 1/0 enables/disables infra red transmit on output port IR3 1/0 enables/disables infra red transmit on output port IR4 the infra-red command string to send	Outputs an Infra-Red command on the Infra-Red output ports
string SerialGet(float fPort)	fPort	1-6 for RS232; 7-9 for RS485; 10-13 for Telnet; 14-17 for UDP	Returns next message from a port
SerialSend(float fPort, string sOutput)	fPort sOutput	1-6 for RS232; 7-9 for RS485; 10-13 for Telnet; 14-17 for UDP string to send out of serial port	Transmits data out of a port
SetAlarm(float fAlarm, string sConfig)	fAlarm sConfig	which alarm to set (1-8) configuration string: ▪ “O dd/mm/yyyy hh:mm” – one time alarm ▪ “D hh:mm” – daily alarm ▪ “W dd/mm/yyyy hh:mm” – weekly alarm ▪ “A +/-hh:mm” – dawn alarm ▪ “P +/-hh:mm” – dusk alarm	Set alarm. This will cause “Alarm()” to be called at the specified time.
SetBaud(float fPort, string sBaud)	fPort sBaud	1-6 for RS232; 7-9 for RS485 (for RS232) – any standard baud 300-115200 (for RS485) – any standard baud 600-230400	Set the baud rate (bits per second data rate) of a serial port.

Function	Parameters	Meaning	Description
SetDout(float fDout, float fState)	fDout fState	1-6 the digital output port to change 1 or 0, the new state of the digital output	Changes the state of a digital output port
SetEndOfMsg(float fPort, float fEom)	fPort fEom	1-6 for RS232; 7-9 for RS485; 10-13 for Telnet; 14-17 for UDP character marking end of message (0-255)	Defines how received messages end for a particular port. If set to zero (default) then the feature is disabled.
SetHubLed(float fLed, float fState)	fLed fState	The front panel LED to change (1-4) The new state of the LED (1 = on, 0 = off, 2 = system mode)	Changes the state of the LEDs on the FreeWay's front panel
SetMsgLength(float fPort, float fLength)	fPort fLength	1-6 for RS232; 7-9 for RS485; 10-13 for Telnet; 14-17 for UDP defines the length of a message (0-255)	Defines the length of received messages from a port. If set to zero (default) then the feature is disabled.
SetMsgTimeout(float fPort, float fTime)	fPort fTime	1-6 for RS232; 7-9 for RS485; 10-13 for Telnet; 14-17 for UDP length of 'silence' in milli-seconds marking the end of a message (0-10000)	Defines the timeout for a particular serial port. Defaults to 100ms
float strlen(string sSrc)	sSrc	input string	Returns the length of the input string

System Event Functions

Event name	Description
Alarm(float fAlarm)	This event function is called when alarm fAlarm has occurred.
DigitalIn(float fDinPort)	This event function is called when a change is detected on a digital input port fDinPort
EIBReceive EIBReceive(float fEibRxHandle, float fEibRxValue, string sEibRxValue)	This event function is called when a elegram of a registered group is received on the EIB bus
HubInit()	This event function is called at power-up.
IRReceive(float fAddress, float fData, float fToggle)	This event function is called when a valid RC5 protocol Infra-Red message is received
SerialReceive(float fPort)	This event function is called when a complete message has been received on port fPort