



PICED Logic Engine Programmer's Guide

Software Version 4.13

Wednesday, 4 February 2015

PICED Logic Programmer's Guide

Copyright Notice

© 2015 Schneider Electric (Australia). All rights reserved

Trademarks

Clipsal is a registered trademark of Schneider Electric (Australia) Pty Ltd.

C-Bus is a registered trademark of Schneider Electric (Australia) Pty Ltd

PICED is a registered trademark of Schneider Electric (Australia) Pty Ltd

Home Management Series is a registered trademark of Schneider Electric (Australia) Pty Ltd

Windows is a trademark of Microsoft Corporation

All other logos and trademarks are the property of their respective owners

Disclaimer

Schneider Electric (Australia) reserves the right to change specifications or designs described in this manual without notice and without obligation.

Table of Contents

Section	Page
1 Introduction	4
1.1 Typographic Conventions	5
1.2 Programs	5
1.3 Operation	5
2 Quick Start Guide	8
2.1 Conditional Logic	8
2.2 Modules	13
2.3 Creating a Logic Project	13
2.4 For users with Programming Skills	20
3 Using the Logic Engine	22
3.1 Logic Editor	22
3.2 Compiling	30
3.3 Running Logic	31
3.4 Logic Engine Options	31
4 Logic Engine Language	34
4.1 Program Structure	34
4.2 Code Formatting	35
4.3 Identifiers	35
4.4 Comments	37
4.5 Constants	38
4.6 Variables	38
4.7 Types	39
4.8 Assignment	42
4.9 Displaying Data	42
4.10 Operators	45
4.11 Standard Functions	53
4.12 Tags	62
4.13 Date Functions	63
4.14 Time Functions	66
4.15 C-Bus Functions	71
4.16 Timer Functions	99
4.17 System IO Functions	101
4.18 Special Days	132
4.19 String Functions	134
4.20 Other Functions	143
4.21 C-Bus Unit Functions	150
4.22 Flow Control	153
4.23 Sub-Programs	168
4.24 Modules	176
4.25 Graphics	183
4.26 Serial IO	197

4.27	Internet	209
4.28	Page Properties	233
4.29	Component Properties	234
4.30	Profiles	244
4.31	Media Transport Control	245
4.32	Complex Data Types	250
4.33	Files	261
4.34	ZigBee Functions	267
5	Debugging Programs	273
5.1	Error Types	273
5.2	Debugging Support Features	273
5.3	Debugging Methods	274
6	Error Messages	277
6.1	Compilation Errors	277
6.2	Run Time Errors	287
6.3	Resolving Compilation Errors	290
7	FAQ	292
7.1	When to use logic	292
7.2	Using Counters	292
7.3	Program Execution	293
7.4	Random Event Times	295
7.5	Logic Engine Security	297
7.6	Handling Triggers	297
7.7	Logic Catch-up	298
7.8	Handling Sets of Loads	298
7.9	Controlling Modules from Components or Schedules	298
7.10	Running Modules Infrequently	299
7.11	Simplifying Logic Conditions	299
7.12	Efficient Code	300
7.13	Fixing Errors	301
7.14	Tracking a Group Address	301
7.15	Logic Templates	302
7.16	How Much Logic Is Possible	303
7.17	Function indices start from 0, not 1	307
7.18	Displaying logic data	307
8	Appendix	308
8.1	Hexadecimal Numbers	308
8.2	Binary Numbers	308
8.3	Character and String Formats	309
8.4	Ladder Logic	312
8.5	Flow Charts	313
8.6	Functional Blocks	314
8.7	Pascal	314
8.8	Tutorial Answers	322

1 Introduction

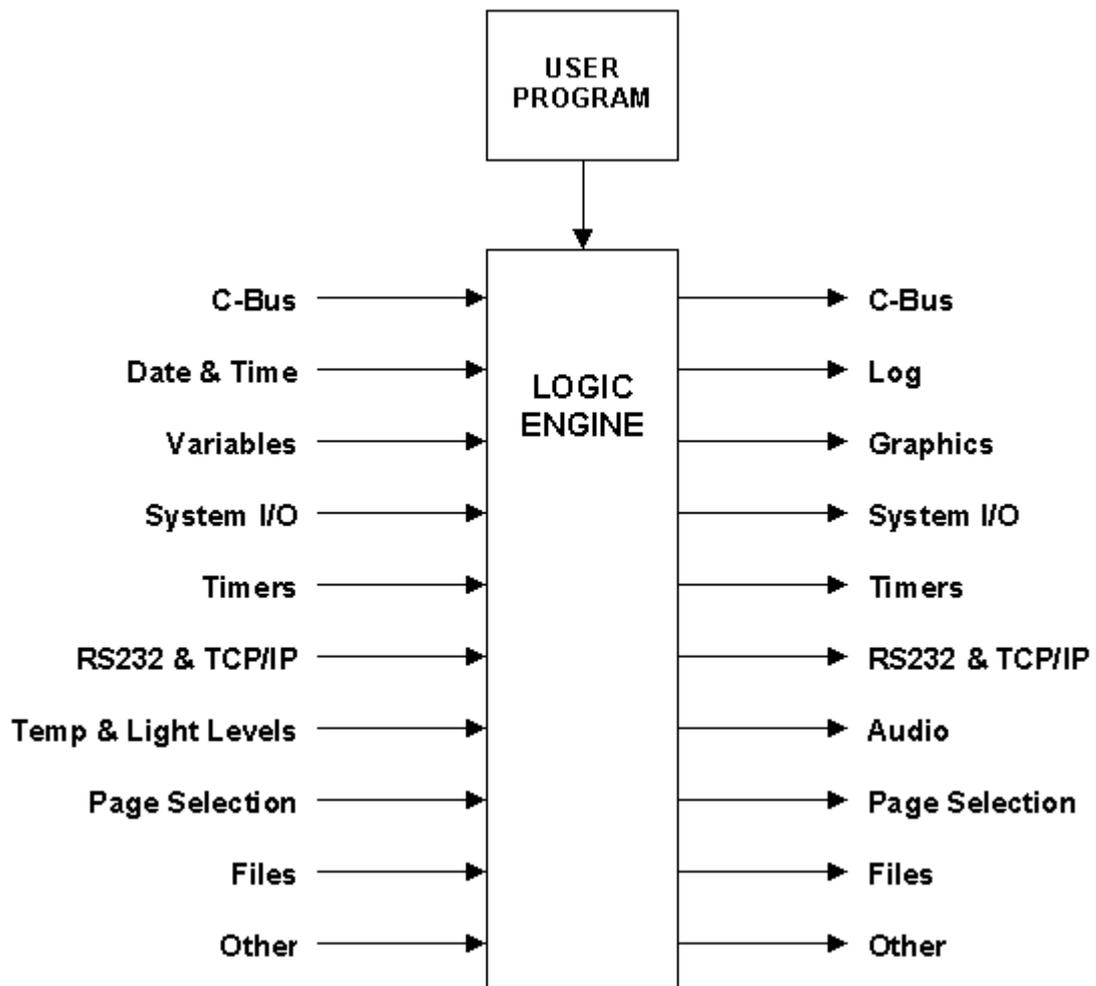
The Logic Engine complements the other functions of the PICED software by allowing the user to implement new or customised system behaviour.

The Logic Engine executes programs for the user to implement features like:

- Scheduling (time and date based events)
- Logic (conditional events)
- Combinations of Scheduling and logic
- Calculations

Functions which PICED can currently support (Scenes, Schedules, Irrigation, Special Days) will not need to be performed within logic, but the logic can interact with them. **It is much more efficient to use the Scene Manager to implement Scenes than to try to implement them using logic.** The same applies to Schedules and Irrigation.

The Logic Engine uses the User [Program](#) to provide instructions for how it should behave. It makes decisions based on the Logic Engine inputs and controls various outputs, as shown below :



Logic can be used for Colour C-Touch, PAC, and Black & White C-Touch Mark II projects, but not for Black & White C-Touch Mark I projects.

1.1 Typographic Conventions

Throughout this document, text representing lines of code is written in Courier font, and is generally indented. For example :

```
SetLightingGroup("Porch Light", on);
Delay("0:10:00");
SetLightingGroup("Porch Light", off);
```

Where the [section](#) of the code is important, the section name will appear in braces { } before the code. For example :

```
{ var declarations }
i : integer;
```

Where part of the code has been left out for clarity, an ellipsis (...) within braces { } is used. For example :

```
{ var declarations }
i : integer;
{ ... }
i := 3;
```

When reference is made to a software button or menu item, the name (or text) is written in **bold**. Menu and submenu items are separated by a vertical bar. For example, **Edit | Undo** would refer to the "Undo" menu item in the "Edit" menu.

All topics are cross referenced. A cross reference looks like [this](#).

Some topics are of a more advanced nature and are not relevant to most users. These will have the icon shown below to show that you can skip the section unless you are requiring the more advanced functions of the Logic Engine.



1.2 Programs

A program is a set of instructions which define how the Logic Engine is to operate.

Ideally, it would be nice to be able to give instructions in a human language (for example, English), such as :

```
At 7:00PM, switch the kitchen lights on.
```

However, human languages are not sufficiently precise, and are often ambiguous. To guarantee that the Logic Engine will perform exactly what is desired, it is necessary to use a "computer language". The example above would be written in the Logic Engine language as :

```
once time = "7:00PM" then SetLightingLevel("Kitchen", ON);
```

which isn't quite as simple, but is still very readable.

A user Program is often referred to as "code".

1.3 Operation

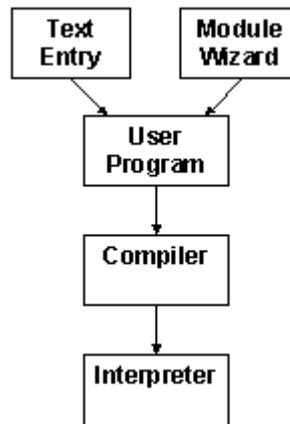
The Logic Engine has an [Editor](#) which allows the user to enter their [programs](#). The [Logic Engine](#)

[Language](#) section describes the language used for the logic programs.

When the user program is executed, it is referred to as a "scan". The user program will be executed (scanned) five times per second (i.e. every 200ms).

Certain actions can be performed when the Logic Engine first starts. These are put in the [Initialisation](#) section. Other actions are executed every time something changes. These are put in the [Modules](#) section.

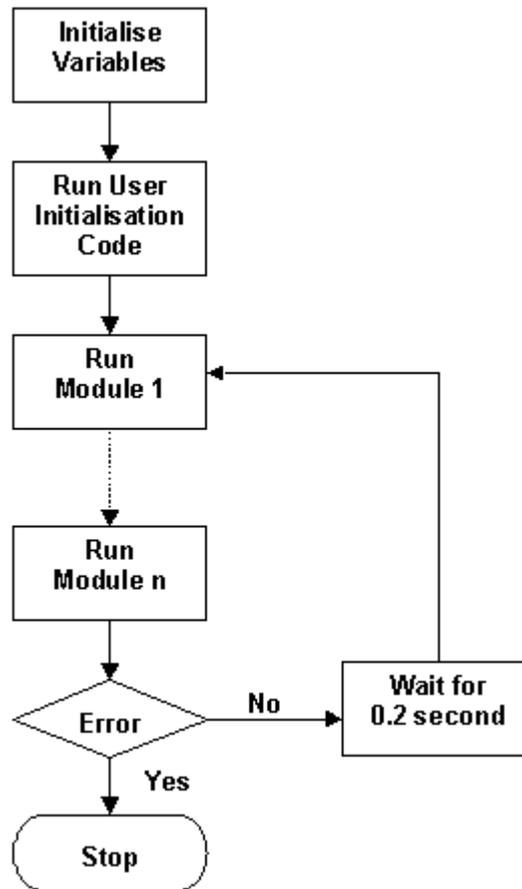
The data flow of the Logic Engine is summarised by the flow chart below.



The steps in the operation of the Logic Engine are as follows :

1. Creation of the user [Program](#), either by direct entry of the [program text](#), or by means of the [Module Wizard](#)
2. [Compilation](#) of the program
3. [Running](#) the logic in an Interpreter

The operation of the Logic Engine is shown in the flow chart below :



The first step in the process is to initialise all variables. The users [Initialisation Code](#) is then executed. The [Modules](#) are then all executed in order. If there are no errors, then the Logic Engine waits for 0.2 seconds and then it runs the Modules again. When an error occurs, the Logic Engine stops (see [Logic Engine Options](#)).

See also [Program Execution](#)

2 Quick Start Guide

This Quick Start Guide introduces the concepts necessary to implement basic logic functions. If you wish to learn all of the details of the Logic Engine, you may skip this section, and proceed with the [Logic Engine Language](#).

2.1 Conditional Logic

The most common structure used in user [Programs](#) is the [IF](#) or [ONCE](#) statement. It is used to perform an action if certain conditions are true.

For example, if you want to switch on the porch light at 7:00 PM every night, the statement would be :

```
if time = "7:00PM" then
    SetLightingState("Porch Light", ON);
```

The IF statement consists of five parts :

- the word IF
- the [condition](#) under which something is to be done
- the word THEN
- the action that is to be done when the condition is true (this is called the [statement](#))
- a semicolon (;)

So the general form (or syntax) of an IF statement is :

```
if condition then
    statement;
```

Condition

The condition is an expression which describes the circumstances under which the statement is to be executed. The condition could be based on the time, date, C-Bus levels or many other things.

In the example above, the condition is :

```
time = "7:00PM"
```

This condition will be true when the current time is 7:00 PM. Hence at 7:00 PM every day, the statement (switching on the Porch Light) will be executed.

Statement

The statement is an action or list of actions to be performed when the expression in the condition is true. The statement can do things like setting a C-Bus Group Address to a level, setting a Scene or selecting a PICED page.

In the example above, the statement is :

```
SetLightingState("Porch Light", ON);
```

This sets the state of a Lighting Group Address called "Porch Light" to ON.

Examples

If the porch light was to only be switched on every Friday night, then the above example would have

its condition changed to :

```
if (time = "7:00PM") and (DayOfWeek = "Friday") then
    SetLightingState("Porch Light", ON);
```

In the above case, there are two parts of the condition, joined by an [AND](#) operator. Both of the parts of the condition have to be true in order for the statement (switching on the Porch light) to be executed.

If the porch light was to be switched on every Friday night, but was also to switch off two hours later, then the statement would be changed to :

```
if (time = "7:00PM") and (DayOfWeek = "Friday") then
begin
    SetLightingState("Porch Light", ON);
    Delay("2:00:00");
    SetLightingState("Porch Light", OFF);
end;
```

Now the statement consists of several statements within a BEGIN / END [Block](#). In this case, there is a statement to switch on the Porch light, then another to delay for two hours, then another to switch off the Porch light.

Edge Triggered Conditions

If you had an outside light level sensor connected to C-Bus and you wanted to switch on the porch light when the level drops below 50%, then you might write the (incorrect) code as :

```
if GetLightingLevel("Outside Sensor") < 50% then
    SetLightingState("Porch Light", ON);
```

The problem is that once it gets dark outside, the condition will be true for many hours. On each program [scan](#) (each second), the Porch light will be switched on. This has two problems :

- it creates a lot of unnecessary C-Bus messages
- if you manually switch the porch light off, the logic will switch it straight back on again

In this case, we really only want the light switched on when the light level [first](#) goes below 50%. If you want a statement to be only executed when something first becomes true, then it is necessary to use the [Once Statement](#). The above code should be written as :

```
once GetLightingLevel("Outside Sensor") < 50% then
    SetLightingState("Porch Light", ON);
```

In this case, when the light level drops below 50%, the porch light will be switched on. The porch light will not be switched on again until the light level has gone above 50% (which makes the condition false) and then drops below 50% again (which makes the condition true again).

The code examples above with `if time = "7:00PM"` will have the same problem and should also use a once statement.

Notes

Basic Conditional Logic, like the examples above, can be put together using the [Module Wizard](#) without having to know much about the logic [Language](#).

2.1.1 Conditions

The condition in a [Conditional Logic](#) statement is an expression which describes the circumstances

under which the statement is to be executed. The most common conditions used in basic logic are described below.

C-Bus

The levels of C-Bus Group Addresses can be used as part of a condition. Various [C-Bus Functions](#) can be used to obtain the level or state of a Group Address. The most common of these are :

[GetLightingLevel](#) : this gets the level of a C-Bus Lighting Group Address (level 0% to 100%)
[GetLightingState](#) : this gets the state of a C-Bus Lighting Group Address (on or off)

Examples

For a condition checking whether the Kitchen Light is on :

```
if GetLightingState("Kitchen Light") then...
```

The name of the C-Bus Group Address [Tag](#) is in double quotes.

For a condition checking whether the Lounge Light is less than 10% :

```
if GetLightingLevel("Lounge Light") < 10% then...
```

Note that the symbol < means "less than".

Date

The present date can be used as part of a condition. The most common [Date Functions](#) are :

[Date](#) : this is the current date
[DayOfWeek](#) : this is the current day of the week (Sunday, Monday etc)

Examples

For a condition checking whether the date is the first of April 2004 :

```
if Date = "1 April 2004" then...
```

Note that the date can be expressed as a [Date Tag](#) with a date within double quotes.

For a condition checking whether the day is not a Sunday :

```
if DayOfWeek <> "Sunday" then...
```

Note that the symbol <> means "not equal to".

Time

The present time can be used as part of a condition. The most common [Time Functions](#) are :

[Time](#) : This is the current time
[Sunrise](#) : This is the sunrise time today
[Sunset](#) : This is the sunset time today

Examples

For a condition checking whether the time is 23:00 (11PM) :

```
if Time = "23:00" then...
```

Note that the time can be expressed as a [Time Tag](#) with a time within double quotes.

For a condition checking whether the time is after sunset :

```
if Time > sunset then...
```

Note that the symbol > means "greater than".

Complex Conditions

Where more complex conditions are required, conditions can be combined with AND and OR operators. If you want to perform an action when all of a series of conditions are true, then use the [AND](#) operator. If you want to perform an action when any of a series of conditions are true, then use the [OR](#) operator.

Examples

For a condition checking whether the time before 6 AM and the day is not a weekend :

```
if (Time < "6:00") and (DayOfWeek <> "Saturday") and (DayOfWeek <> "Sunday")  
then...
```

Note that brackets must be used to group each part of the condition together.

For a condition checking whether any of the kitchen lights are on :

```
if GetLightingState("Kitchen 1") or GetLightingState("Kitchen 2") or  
GetLightingState("Kitchen 3") then...
```

See also

[Relational Operators](#)
[Boolean Operators](#)

2.1.2 Statements

The statement in a [Conditional Logic](#) statement is an action or list of actions to be performed when the expression in the condition is true. The most common statements used in basic logic are described below.

C-Bus

The levels of C-Bus Group Addresses can be set as part of a statement. Various [C-Bus Procedures](#) can be used to set the level or state of a Group Address. The most common of these are :

[SetLightingLevel](#) : this sets the level of a C-Bus Lighting Group Address (level 0% to 100%)
[SetLightingState](#) : this sets the state of a C-Bus Lighting Group Address (on or off)

Examples

To Switch the Dining Light on :

```
SetLightingState("Dining Light", ON);
```

To ramp the passage light to 0% over 30 seconds :

```
SetLightingLevel("Passage Light", 0%, "30s");
```

Scenes

C-Bus Scenes can be controlled from Logic. The most common [Scene Procedure](#) is :

[SetScene](#) : this sets a C-Bus Scene

Example

To set a Scene called "All Off" :

```
SetScene("All Off");
```

Selecting Pages

The Page displayed by PICED can be set from Logic. The [ShowPage](#) procedure can be used for this.

Example

To show the page called "Alarm" :

```
ShowPage("Alarm");
```

Delay

Sometimes it is necessary to wait before executing another statement. The [Delay](#) procedure is used for this.

Example

To delay for 1 second :

```
Delay(1);
```

To delay for 1 hour and 30 minutes :

```
Delay("1:30:00");
```

Compound Statements

To combine several statements together, they can be put in a BEGIN / END [Block](#). The statements

will be executed in the order that they appear. A compound statement can be used anywhere that a statement can be used.

Example

To set the "All Off" scene, delay for 10 seconds, then switch on the "Arm" group when group "Leave Home" is first set:

```
once GetLightingState("Leave Home") then
begin
  SetScene("All Off");
  Delay(10);
  SetLightingState("Arm", ON);
end;
```

2.2 Modules

[Modules](#) are used to group together related bits of functionality. Typically a Module contains some [Conditional Logic](#).

To create a new Module for your program, open the [Logic Editor](#) by clicking on the **Logic** button on the tool bar. Open the [Module Wizard](#) by clicking on the **Wizard** button on the Logic Editor tool bar to create a new Module. This is demonstrated in the following section.

2.3 Creating a Logic Project

This section takes you through the steps involved with creating a basic logic project.

When a PICED project is created, the steps to follow are as follows :

1. [Determine the Requirements](#)
2. [Document the Requirements](#)
3. [Create the Project Structure](#)
4. [Create the Logic Modules](#)
5. [Test the Logic](#)
6. [Archive the Project](#)

2.3.1 Determine the Requirements

Before any work can commence, it is necessary to determine the requirements of the end user. This involves talking with the user, and suggesting possible solutions to their needs. At this stage, the implementation is not important, it is the outcome which needs to be determined.

For this exercise, we will pretend that the user wants the passage light to dim slowly to 50% if it is greater than that level at 10:00PM.

2.3.2 Document the Requirements

It is very important to document the requirements of a project for several reasons :

1. It ensures that the requirements will not be forgotten
2. It provides an agreed set of specifications which the user and the installer can agree on
3. It provides something which you can test against to ensure that the installation operates as

- promised
4. It helps to clarify the requirements in your mind.

It is likely that if you can't document a requirement, you will not be able to fulfil that requirement.

There are many ways of documenting requirements. Each are suitable in certain circumstances.

Text

There is nothing wrong with describing the requirement in text as long as you do so precisely. Unfortunately, human languages are often ambiguous and this can lead to confusion. This is why some of the alternative methods are used.

For our example, the requirement written as text would be :

"At 10:00PM, if the Passage Light is at a level of more than 50%, dim it to 50% over 30 seconds".

Truth Tables

A "truth table" is a diagram showing the combinations of the various elements which make up the requirement and the action which has to be taken.

For our example, the requirement represented as a truth table would be :

Time	Passage Light	Action
10:00 PM	over 50%	Dim Passage Light to 50% over 30 seconds
10:00 PM	50% or less	do nothing
not 10:00 PM	over 50%	do nothing
not 10:00 PM	50% or less	do nothing

Flow Charts

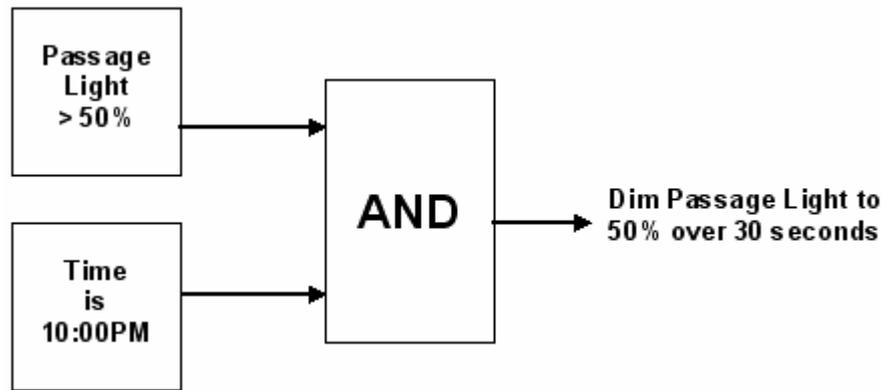
A [Flow Chart](#) can be used to represent a sequence of actions and decisions. Our example is not particularly suitable for representation with a flow chart.

Ladder Logic Diagrams

[Ladder Logic](#) is used to represent the relationships between "inputs" (conditions) and "outputs" (actions). Our example is not particularly suitable for representation with ladder logic.

Functional Block Diagrams

[Functional Blocks](#) can also be used to represent the relationships between "inputs" (conditions) and "outputs" (actions). Our example can be represented with a functional block diagram as shown below.



2.3.3 Create the Project Structure

The next step is to create the PICED Project structure. For our exercise, follow these steps :

1. Start the PICED software.
2. Select **Create a New Blank Project**
3. Click on **Next**
4. Select **Colour C-Touch**
5. Click on **Next**
6. Select the "home" project in the **C-Bus Project** list
7. Click on **Finish**

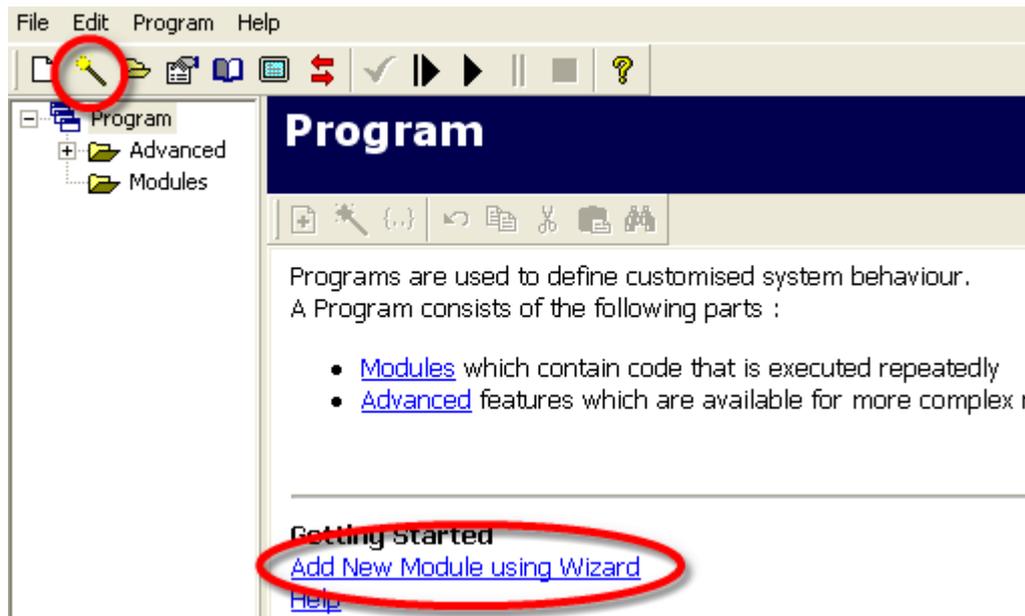
We now want to create Pages and Components for use with our Project. Place a Slider on the page, and set it to control the "Passage Light" group address. If this Group Address doesn't exist, click on the add button (looks like a +) next to the Group Address list. Refer to the PICED main help file for details on adding Components.

2.3.4 Create the Logic Modules

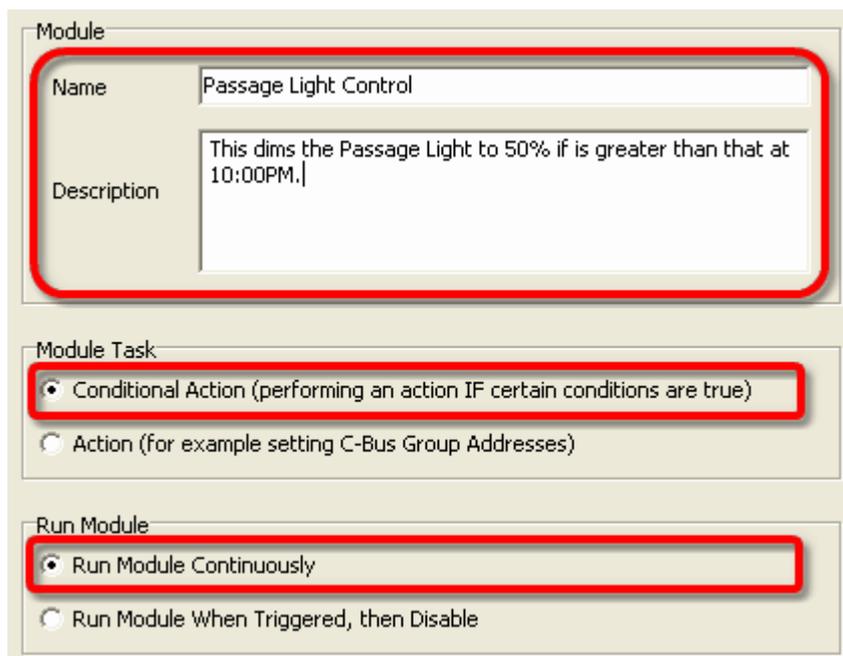
This step is where the logic is actually written.

Click on the logic button on the tool bar to open the [Logic Editor](#).

Add a new module using the Logic Wizard by clicking on the Logic Wizard icon or by clicking on **Add New Module using Wizard**.



The Module Wizard will be displayed. Enter a meaningful name for the Module and a description of what it does, as shown below. Select **Conditional Action** and **Run Module Continuously**. Click on **Next**.



The Conditions page is now shown. For our example, we want the Passage Light to be dimmed when the time is 10:00PM and if the level is greater than 50%.

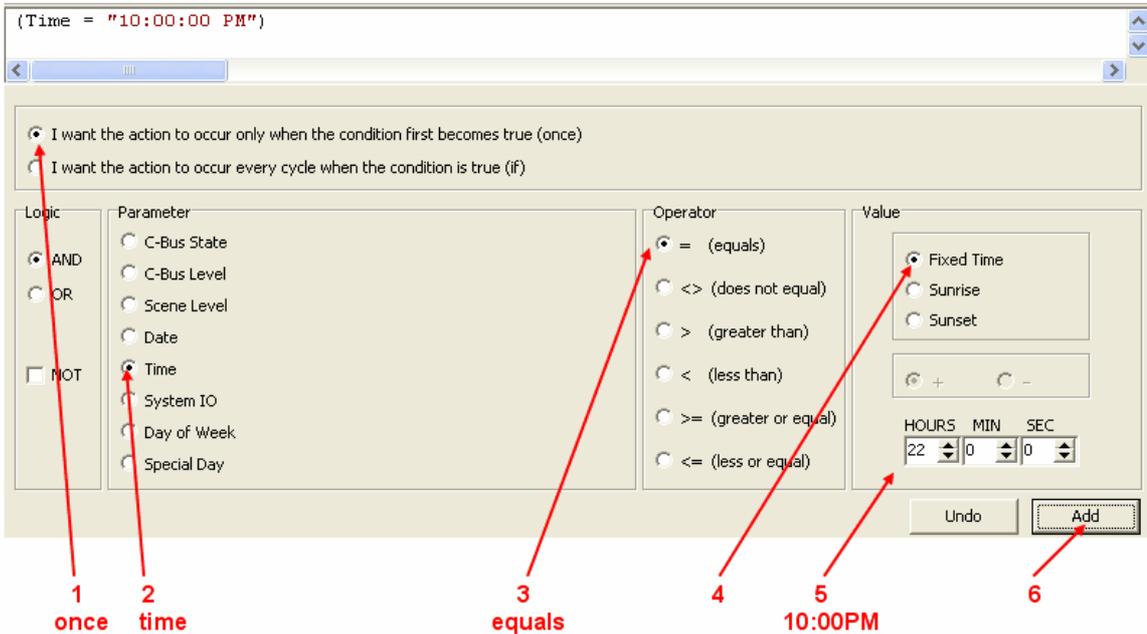
The first part of our condition is that the time is 10:00PM. Select the controls in the order shown below. Note that the condition can be "read" from left to right. When the **Add** button is clicked, the condition is displayed in the text at the top. The condition is shown as

(Time = "10:00:00 PM")

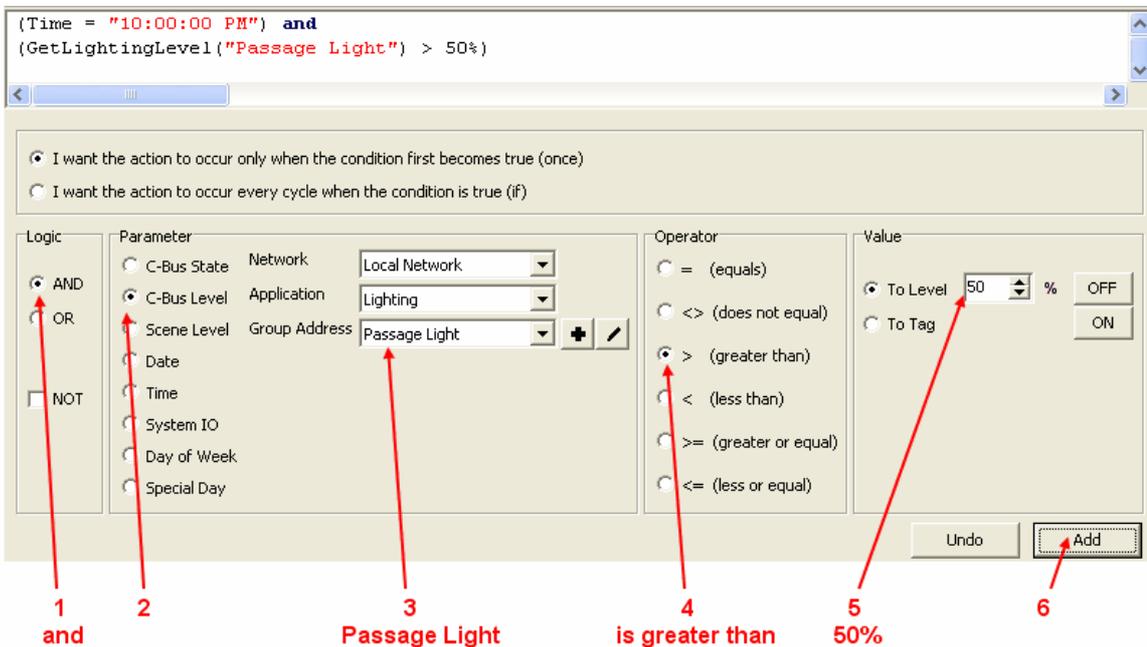
which is the logic engine's way of saying "the time is 10:00 PM".

Conditions

Please select the conditions under which you want the actions performed



The next condition can be added by following the steps below.



The **and** button is selected because we want to dim the Passage Light when the time is 10:00PM and the light is greater than 50%. The second condition is expressed in the logic engine language as

```
(GetLightingLevel("Passage Light") > 50%)
```

Now that the conditions have been created, click on **Next** to show the Actions page. Select the properties as shown below. When the **Add** button is clicked, the action will be shown at the top.

Actions

Please select the Actions you want to perform

1 set

2 Passage Light

3 to 50%

4 over 30 seconds

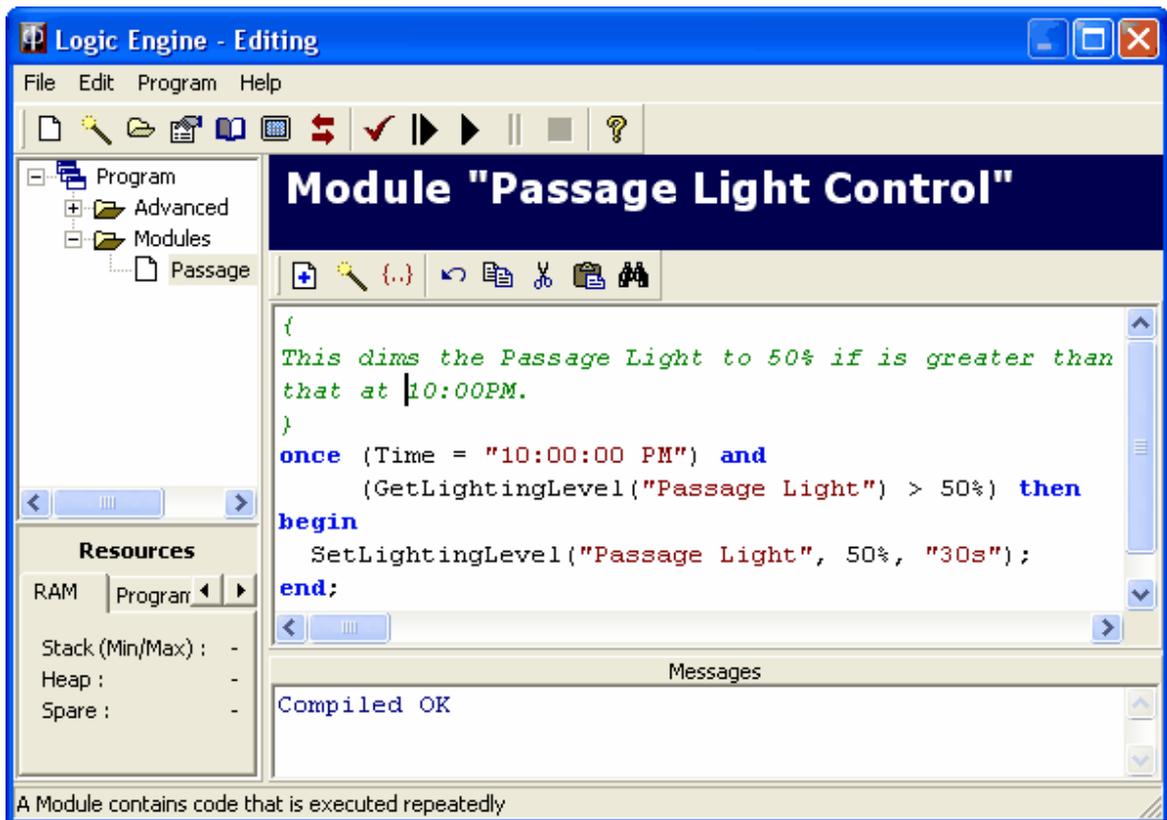
5

The action is expressed in the logic engine language as :

```
(SetLightingLevel("Passage Light", 50%, "30s");
```

Click on the **Finish** button to finish the Logic Wizard.

The logic code entered has had a few things added to it, and it now appears in the logic editor as shown below.



Notice that the description that was entered has been turned into a [Comment](#). The conditions and actions have been turned into a [Once Statement](#).

2.3.5 Test the Logic

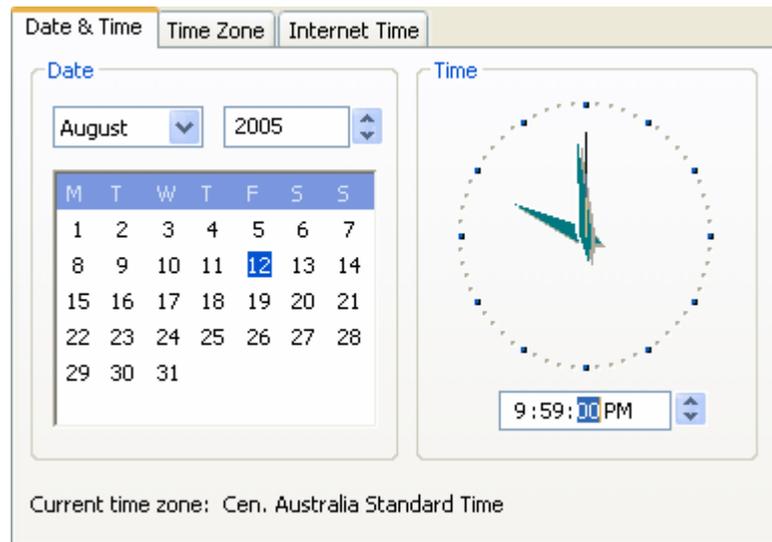
To test the logic, the first step is to [Compile](#) the logic. To do this, click on the Compile button (looks like a tick) on the [Tool Bar](#). If the code is all correct, you should see a message "Compiled OK" in the output window at the bottom of the Logic Editor.

To test the logic, we need to have it [Running](#). Click on the run button (looks like a triangle) on the [Tool Bar](#). The logic code will change to a grey colour and the top of the Logic Editor will say "Logic Engine - Running".

If you now close the Logic Editor, you will see an L next to an S at the bottom of the main form. The L indicates that the logic is running.

If you change the level of the Passage Light slider, nothing should happen. To test that our requirements have been implemented correctly, set the slider to 100%. Now change the computer clock to 9:59PM. To change the computer time :

1. Double click on the clock on the right of the task bar
2. The Date and Time Properties form will be displayed as shown below.
3. Set the new time
4. Click on **OK**



When the time changes to 10:00 PM, you should see the Passage Light slider slowly move down to 50%.

For completeness in the testing, set the slider to around 20%. Change the time to 9:59PM again. When the time changes to 10:00PM, the Passage Light level should not change.

When complete, change the computer time back to the correct setting.

If you have an error with the code, you will need to :

1. [Debug](#) the code
2. Compile the code again
3. Test it again
4. Repeat steps 1 to 3 until it works correctly.

2.3.6 Archive the Project

When you have completed the project, you should save the Project as an Archive and save the archive to a backup disk and ideally provide a copy to the user for their records. If you ever have a computer malfunction, this archive will enable you to recover your work without having to start again.

2.4 For users with Programming Skills

If you already have experience with computer programming, this section provides details of the similarities and differences between the logic engine and other programming languages.

The Logic Engine is based on ANSI [Pascal](#). If you are familiar with Pascal, there should be very little "learning curve" in using the Logic Engine. The main aspects of the Logic Engine which make it unique are summarised below.

Modules

A [Module](#) is a named unit of code. All Modules share common memory.

Modules can contain [delays](#). While a Module is delayed, the other Modules are unaffected. In this manner, it is a little like a multi-tasking environment.

Modules can also be [enabled](#) and [disabled](#).

Execution Scan

Every 200ms, all of the Modules are executed in the order in which they are listed. This is called a "scan".

On the first scan only, the [Initialisation](#) code is executed.

Edge Triggered Condition Statements

The [Once Statement](#) is an edge-triggered conditional statement. When the condition first changes from false to true, code can be executed.

Tags

[Tags](#) are text representations of integer values which make the code easier to read. They are not the same as [Constants](#), but perform a similar function.

See also [Program Execution](#), [Program Structure](#) and [Operation](#).

3 Using the Logic Engine

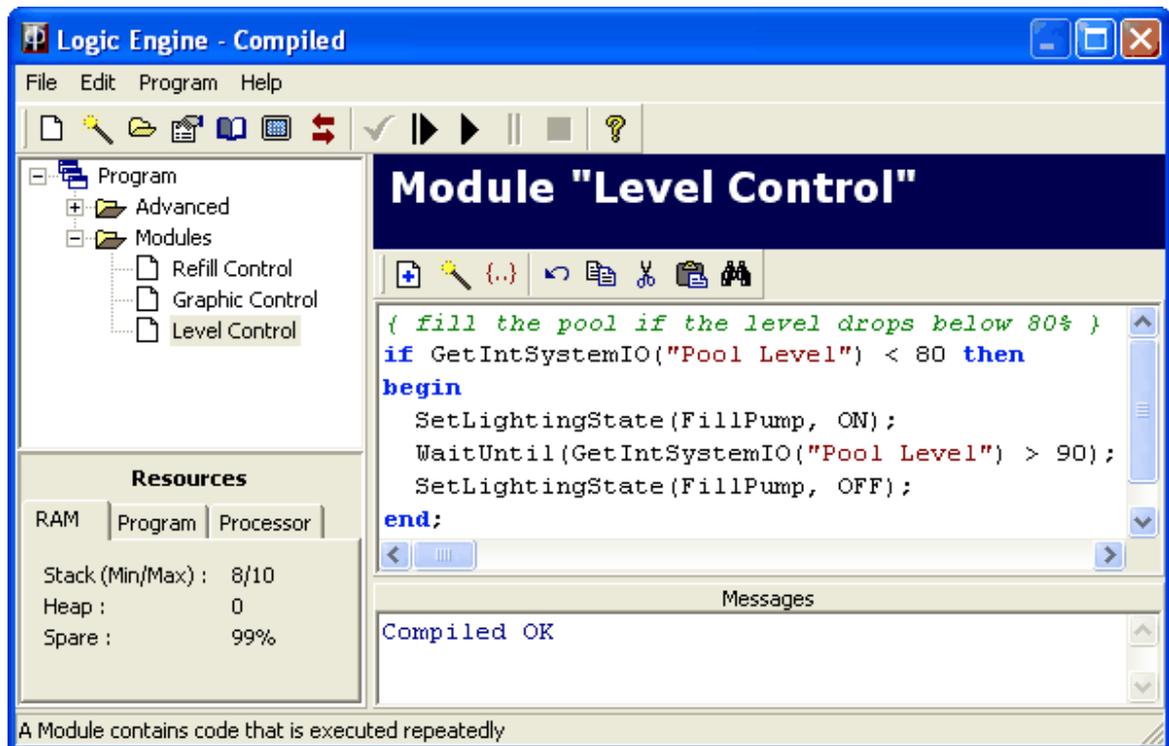
The steps to using the Logic Engine are :

- The Logic Engine [Editor](#) is used as a means of entering the user [program](#).
- The user program is then [Compiled](#)
- Any program [Errors](#) are fixed
- The program is [Run](#).
- If necessary, various [Debugging](#) methods can be applied to ensure that the program runs as desired.

3.1 Logic Editor

The Logic Editor is shown below. It consists of several sections :

- A [Toolbar](#) at the top, which provides access to common functions
- A [logic "tree"](#) on the left which provides access to various parts of the code
- A [code window](#) on the right where the user program is entered
- An [output window](#) at the bottom, where results are displayed
- A [Resources Window](#) at the bottom left showing the Resources used
- A Status Bar at the bottom showing Hints



3.1.1 Menu Items

The Logic Editor menu items provide access to many of the Logic Engine functions. Most of those functions are available elsewhere too.

File Menu

Save - saves the Project

Logic Report - generates a [Logic Report](#)

Close Form - close the Logic Editor

Edit Menu

Undo - undoes the last action in the Code Window

Copy - copies selected text in the [Code Window](#)

Cut - cuts selected text in the Code Window

Paste - pastes copied text in the Code Window

Find - finds some text in the Code Window

Find In All - finds some text in all nodes of the tree (starting with **constants**)

Find Next - finds the next occurrence of the text in the Code Window

Replace - replaces selected text in the Code Window

Go To Line Number - goes to the selected line number in the [Program](#)

Program Menu

Compile Logic - [Compiles](#) the logic

Run Once - [Runs](#) the Logic program once

Run Logic - Runs the Logic program continuously

Pause Logic - pauses the Logic program

Stop Logic - stops the Logic program

Help Menu

Help - provides access to the logic help file

There are also pop-up menus on the [Logic Tree](#) and the [Code Window](#).

3.1.2 Tool Bar

The Toolbar at the top of the [Logic Editor](#) allows quick access to commonly used features of the Logic Editor.

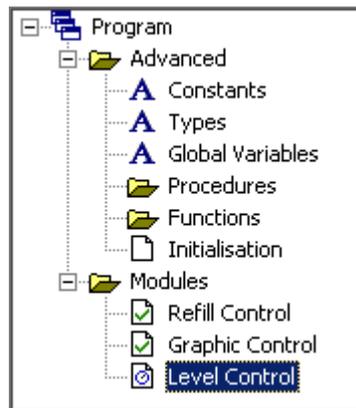


The features available via the toolbar are :

- Clearing all code
- [Module Wizard](#)
- Adding Module Groups
- [Logic Engine Options](#)
- Logic Engine [Report](#) (listing of the user Program)
- Displaying [Graphics Commands List](#)
- Editing [System IO](#) variables
- [Compiling](#)
- [Running Logic](#)

3.1.3 Logic Tree

The Logic Tree on the left of the [Logic Editor](#) provides access to the various parts of the code.



The two main parts of the Logic Tree are :

- **Advanced** - this provides access to the advanced parts of the Logic Engine
- **Modules** - this is where most of the user program is written

The Advanced section contains the following sections, which (with the exception of Initialisation) correspond to sections of standard Pascal code :

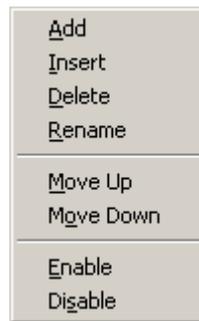
- [Constants](#)
- [Types](#)
- [Variables](#)
- [Procedures](#)
- [Functions](#)
- [Initialisation](#)

Each of these sections of the [program](#) is created automatically when the program is [Compiled](#), so it is not necessary for you to type in the headers (const, var etc).

To create new code sections :

- [Constants](#) : select the **Constants** node. Click on the **Add** button on the [Tool Bar](#) or pop-up menu. Enter the Constant name and value. Click on **OK**.
- [Types](#) : select the **Types** node. Type the new type definition in the [Code Window](#)
- [Variables](#) : select the **Variables** node. Click on the **Add** button. Enter the Variable(s) name and type. Click on **OK**.
- [Procedures](#) : select the **Procedures** node. Click on the **Add** button. Enter the Procedure name. Click on **OK**. Type the procedure body in the Code Window
- [Functions](#) : select the **Functions** node. Click on the **Add** button. Enter the Function name. Click on **OK**. Type the function body in the Code Window
- [Initialisation](#) : select the **Initialisation** node. Type the initialisation code in the Code Window
- [Modules](#) : select the **Modules** node. Click on the **Add** button. Enter the Module name. Click on **OK**. New Modules can also be added with the [Module Wizard](#) by clicking on the **Wizard** button.
- [Module Groups](#) : click on the **New Module Group** button on the [Tool Bar](#). Enter the name and click on **OK**.

If you right click on the Logic Tree, you will see a pop-up menu appear :



This allows you to perform functions like :

- Adding new code sections (see above)
- Inserting a new code section
- Deleting a code section
- Changing the order of the code sections (Move Up or Move Down)
- Manually Enabling and Disabling [Modules](#)
- Editing the Schedule controlling a Module (this menu item is only visible if a Schedule controls the Module)

When the logic is running, the state of Modules can be seen by the icons :

- A green tick indicates that the Module is enabled
- A red cross indicates that the Module is disabled
- A blue timer indicates that the Module is waiting for a [Delay Procedure](#) or a [WaitUntil Procedure](#).

When the logic is not running, an icon with a + on it indicates that the module is controlled (enabled or disabled) by a [Component or a Schedule](#).

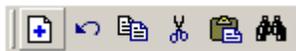
Rearranging tree nodes

The order of the Procedures, Functions, Modules and Module Groups can all be changed. There are two ways of doing this :

- select the node to be moved, the right click and select Move Up or Move Down
- select the node to be moved, then hold the left mouse button down and drag the node to the desired place on the tree

3.1.4 Code Window

The Code Window is the window on the right of the [Logic Editor](#) which is primarily used for editing user [programs](#) (code). The Code Window has its own toolbar :



The toolbar in the Code Window allows :

- [Adding new code](#) sections
- Editing (cut, copy, paste, undo)
- Searching

To enter code in a section of the program, click on the appropriate node in the [Logic Tree](#), and enter text in the Code Window.

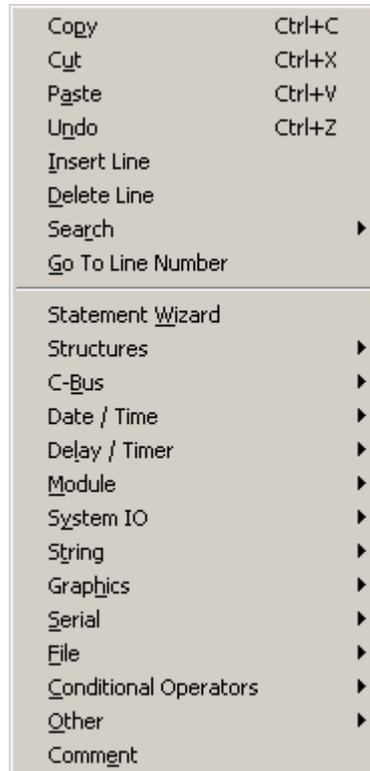
There are essentially four ways of generating code :

- The [Module Wizard](#) can also be used to automate the creation of code for [Modules](#)
- The [Statement Wizard](#) can be used for creating sections of code for a Module
- The pop-up menu can be used for automatically generating small sections of code (see below)

- Code can be typed in directly

Pop-up Menu

If you right click on the code window, a pop-up menu will appear :



This provides access to most of the Logic Engine language elements. Many forms are included to automate the generation of code.

You can also use the pop-up menu for editing to copy, cut and paste code, as well as undoing changes. A search facility is included to find words within the code window.

 Note that the The Logic Program can not be Edited while the Logic Engine is [running](#).

3.1.5 Output Window

The Output Window at the bottom on the [Logic Editor](#) is used to display :

- Results of the [Compilation](#)
- [Error Messages](#)
- [Data](#) written from the user program

3.1.6 Module Wizard

New [Modules](#) can be added with the Module Wizard by following this process :

- Click on the Wizard button on the [tool bar](#)
- Enter the [Wizard Details](#)
- Click on the **Next** button
- Enter the [Wizard Conditions](#)
- Click on the **Next** button
- Enter the [Wizard Actions](#)

- Click on the **Finish** button
- If needed, make any additions / changes manually in the [Code Window](#)

3.1.6.1 Wizard Details

The Module details can be entered on the first page of the [Module Wizard](#).

The following details for the Module can be entered :

- Name : this should be a meaningful name describing the Module
- Description : this should provide details of the purpose and operation of the Module
- Module Task : this selects whether the Module is to be a conditional action ([If](#) or [Once](#)) or just a series of [actions](#)
- Run Module : this selects whether the Module should run continuously or only when triggered

If you select the **Run Module When Triggered, then Disable** option, a [DisableModule](#) statement is added to the end of the Module to disable it when complete. A [DisableModule](#) statement is also added to the [Initialisation](#) section so that the Module is disabled on start-up. To run the disabled module, it is necessary to enable the Module. It will then run a single time and then disable itself again. To automatically create a Schedule to enable the Module, select the **Automatically Create Enabling Schedule** option. See also [Controlling Modules from Components or Schedules](#).

Click on the **Next** button when complete.

3.1.6.2 Wizard Conditions

The Module Conditions can be set on the second page of the [Module Wizard](#). The Conditions are the circumstances under which the [Actions](#) are to be executed. The types of Conditions which can be used include the following parameters :

- [C-Bus Level](#)
- [Scene Level](#)
- [Date](#)
- [Time](#)
- [System IO Variable](#) value

- [Day of the Week](#)
- [Special Day](#) type

The screenshot shows a configuration window for setting a condition. At the top, there are two radio buttons: "I want the action to occur only when the condition first becomes true (once)" (selected) and "I want the action to occur every cycle when the condition is true (if)". Below this, the interface is divided into several sections:

- Logic:** Contains radio buttons for "AND", "OR", and a checkbox for "NOT".
- Parameter:** A list of condition types with radio buttons: "C-Bus State" (selected), "C-Bus Level", "Scene Level", "Date", "Time", "System IO", "Day of Week", and "Special Day".
- Network:** A dropdown menu showing "Local Network".
- Application:** A dropdown menu showing "Lighting".
- Group Address:** A dropdown menu showing "<Unused>" with plus and minus icons.
- Operator:** A list of relational operators with radio buttons: "=" (equals), "<>" (does not equal), ">" (greater than), "<" (less than), ">=" (greater or equal), and "<=" (less or equal).
- Value:** Radio buttons for "OFF" (selected) and "ON".

At the bottom right, there are "Undo" and "Add" buttons.

The following steps need to be taken to apply the condition(s) :

- Select the type of condition ([if](#) / [once](#))
- Select the first condition (see below)
- Click on the **Add** button
- If there are more conditions, for each one :
 - Select the [Logic Operator](#) (AND, OR, NOT)
 - Select the condition (see below)
 - Click on the **Add** button
- Note that changes to the conditions can be manually type in (for example, adding brackets)
- When complete click on the **Next** button

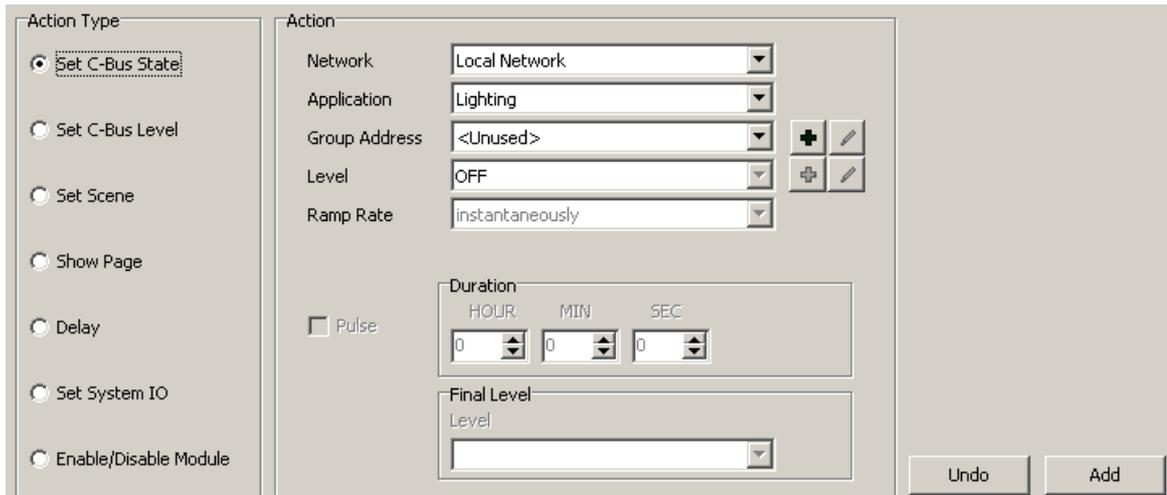
To specify a Condition, select a **Parameter** from the list, select a [Relational Operator](#) and then select a **Value**. Click on the **Add** button to add the Condition to the list.

Where multiple conditions are used, it is necessary to select a [Boolean Operator](#) from the **Logic** group on the left.

3.1.6.3 Wizard Actions

The Module Actions can be set on the third page of the [Module Wizard](#). An Action is something that the Logic Engine can do under specified [Conditions](#). The types of Actions which can be performed include the following :

- Set [C-Bus Level](#)
- Set [Scene](#)
- [Select Page](#)
- [Delay](#)
- Set [System IO Variable](#) value
- [Enable](#) / [Disable](#) Module



To specify an Action, select an **Action Type** and details of the action from the **Action** group. Click on the **Add** button to add the Action to the list.

When complete, click on the **Finish** button.

3.1.7 Statement Wizard

The Statement Wizard is very similar to the [Module Wizard](#), except that it only creates sections of code, and does not create a new Module. It can be used to create conditional statements (a condition and some actions) or plain statements (actions).

To use the Statement Wizard, click on the [Code Window](#) where the code is to be inserted. Select the **Statement Wizard** item from the pop-up menu (right click on the Code Window). Follow the instruction for the Module Wizard, except that it is not necessary to enter a Module name (as one is not being created).

3.1.8 Resource Window

The Logic Engine has finite resources, primarily memory. The Resources Window on the [Logic Editor](#) show the following data :

- Stack Size : the stack is where all Static (regular) variables are stored. The Stack size is how much memory has been used.
- Heap Size : the heap is where [Dynamic Variables](#) are stored. The Heap size is how much memory has been used.
- Spare : this is the amount of data memory spare
- Program : this is the amount of program memory used
- Int : this is how much of the Integer constants memory has been used
- Real : this is how much of the Real constants memory has been used
- String : this is how much of the String constants memory has been used
- Instructions : this is the number of Interpreter instructions executed in the last [scan](#)
- Scans : this is the number of scans run since the Logic Engine was started
- Time : this is the time / average time taken by the Logic Engine scan(s). This should be less than 100ms for stable operation. For PAC and C-Touch Mark II projects, this shows a rough estimate of the time and maximum time as a percentage of the unit capacity. This should not exceed 75% to allow some margin for error. See [How Much Logic Is Possible](#) topic for more information.

The Resources Window can be hidden using the [Logic Engine Options](#).

Note that the Resource Data are only updated when the Logic Engine is [Run](#).

3.1.9 Keyboard Shortcuts

There are various Logic Editor actions which can be performed using the keyboard which can be much faster than using a mouse to select a menu item. These Keyboard Shortcuts are listed below :

Shortcut	Action
TAB	Insert a Tab (spaces)
CTRL + C	Copy the selected text
CTRL + X	Cut the selected text
CTRL + V	Paste the selected text
Del	Delete the selected text
CTRL + Del	Delete to the end of the word
CTRL + Z	Undo the last action
CTRL + F	Find text
F3	Find next
CTRL + R	Replace text
F1	Help
CTRL + I	Indent the selected text by two spaces
CTRL + SHIFT + →	Select to end of next word
CTRL + SHIFT + ←	Select to start of previous word
SHIFT + HOME	Select to start of line
SHIFT + END	Select to end of line
CTRL + SHIFT + HOME	Select to start of page
CTRL + SHIFT + END	Select to end of page
ALT + →	Move to end of next word
ALT + ←	Move to start of previous word
ALT + HOME	Move to start of page
ALT + END	Move to end of page
INSERT	Toggle between insert and overwrite

3.1.10 Logic Report

A report containing the details of the user [Program](#) can be generated by clicking on the Logic Report button on the [Tool Bar](#). A report will be generated containing :

- A full listing of the user program
- A list of which Group Addresses are used where
- A list of which [System IO](#) variables are used where

3.2 Compiling

To compile a program, click on the **Compile** button on the [Logic Editor](#) toolbar.

Any [Compilation Errors](#) will appear in the window at the bottom. To find where in the code an error is, just double click on the error message, and you will be taken to the correct place in the code.

Notes

Clicking on the **Run** or **Run Once** buttons will compile the code before proceeding.

After compiling, you will not be able to compile again until the program has been changed.

3.3 Running Logic

The Logic can be [Compiled](#), run, paused or stopped using the [Tool Bar](#) :



To run the Logic program, click on the **Run Once** or **Run** buttons on the [Logic Editor](#) tool bar.

The **Run Once** button will make the Logic Program run through one [Scan](#), and then stop. This is generally only used when [Debugging Programs](#).

The **Run** button will make the Logic run continuously, and is the normal mode of operation.

To pause the Logic Engine (when it is running), either :

- Click on the **Run** button again; or
- Click on the **Pause** Button

To Stop the Logic Engine, click on the **Stop** button. When the Logic Engine has been stopped (as opposed to paused), re-starting it will cause the [Initialisation](#) to be run again.

Notes :

- Clicking on the **Run** or **Run Once** buttons will compile the code if it hasn't already been done.
- If a [Run Time Error](#) occurs, the Logic will stop and an error message will appear in the window at the bottom.
- The Logic Program can not be [Edited](#) while the Logic Engine is running.

3.4 Logic Engine Options

To set the Logic Engine options, select the Options button on the Logic Editor [Tool Bar](#). The Logic Options form will appear, allowing you to select the following options. All options are restored when PICED is re-started, with the exception of **Allow use of all Functions for Testing** which is always off when the software starts.

Editor Options

Resource Usage

The **Show Resource Usage** check box selects whether the Logic Engine [resources](#) window is shown or not.

Enable Range Checking

The **Enable Range Checking** check box selects whether the compilation inserts commands to check the range of parameters. This is normally used when a program is first being written and tested. Once it has been thoroughly tested, this option may be switched off, as it slows the Logic Engine slightly.

Show Function Parameters

When the pop-up menu is used in the [Code Window](#), code is automatically generated. If the Show Function Parameters option is selected, details of the parameters are written to the code window to save having to look up the details in the help file. For example, if the RoundRect function is selected

then the following code will be generated :

```
RoundRect(_left_, _top_, _right_, _bottom_, _radius_); { with Show Function
Parameters set }
RoundRect( , , , , ); { without Show Function Parameters set }
```

Use Short C-Bus Functions

When the [C-Bus functions](#) are selected with the logic [Wizard](#) or by using the right click menu in the [Code Window](#), if the selected network is the Default Network and the Application is Lighting, Trigger Control or Enable Control, the short version of the functions can be used. For example, the following two functions are the same :

```
SetCBusState("Local Network", "Lighting", "Kitchen", ON);
SetLightingState("Kitchen", ON);
```

The **Use Short C-Bus Functions where Possible** check box controls whether to automatically generate code using the short C-Bus functions or the long ones.

Allow use of all Functions

There are occasions where it is desirable to be able to use logic functions which are not applicable to the project type. This is generally only useful for testing purposes. For example, if you are testing a PAC project in PICED, you may wish to use logging functions. To enable the use of all functions temporarily, select the **Allow use of all Functions for Testing** check box. With this option selected, you will receive [compiler warning](#) W006 instead of compiler error C179 if using an invalid function.

WriteLn Output

The output of the [WriteLn Procedure](#) is normally written to just the [Output Window](#). By selecting the **Send WriteLn output to Log** option, the output of the WriteLn procedure is also written to the Log as for the [LogMessage Procedure](#).

Fonts Options

Syntax Highlighting

The Highlight Syntax option selects whether the different components of the [code](#) are highlighted in different colours or not. To enable Syntax Highlighting, select the **Show Syntax Highlighting** check box.

Fonts

The [Code Window](#) and [Output Window](#) can both have their font set. Click on the **Edit** buttons to select a new font. A fixed width font, such as Courier New, generally works best.

Errors Options

Auto Re-start

The **Auto Re-start Logic Following an Error** check box selects whether the Logic Engine will automatically start again following a [run-time error](#).

Critical Errors

The **Treat all Errors as Critical Errors** selects whether the Logic Engine should continue operating when a [non-critical run-time error](#) occurs.

Warn of excessive PAC usage

For PAC projects, this controls whether you get warnings if the logic is too much for a PAC. You will get a logged message if it exceeds 75% of the PAC capacity, and a [Run Time Error](#) if it exceeds 100%. You can see the current PAC usage in the [Resource Window](#).

Maximum Instructions

The **Maximum Instructions** box allows the maximum number of instructions in one [scan](#) to be set to prevent the Logic Engine from being caught in an infinite loop if there is an error in the user Program. The [Resource Window](#) shows how many instructions were executed in the last scan. The maximum number of instructions needs to be at least a bit bigger than the number of instructions which are executed when things are operating correctly.

Maximum Consecutive C-Bus Scans

One of the most common [Logical Errors](#) is to have an [If Statement](#) instead of a [Once Statement](#), resulting in C-Bus commands being sent on every [scan](#) under certain conditions. By setting a value in the **Maximum Consecutive C-Bus Scans** edit, an error will be raised when a C-Bus command has been sent on every one of the selected number of consecutive scans.

Other Options

Wait for C-Bus

If you want the logic engine to wait until C-Bus has connected before starting the logic engine, select the **Wait until C-Bus is on-line to start logic** check box. Note that in PICED this does not wait until C-Gate has synchronised all C-Bus networks.

In Colour C-Touch, if the option is selected, it waits until the C-Bus "discovery" process is complete, which generally takes less than one minute.

In other units, they always wait for the discovery process to be complete before starting logic (ie. the option is ignored). For these units, the option is there for testing purposes in PICED.

4 Logic Engine Language

The Logic Engine language is based on the [Pascal](#) language. Pascal was chosen because it is one of the easiest languages to learn, and is one of the most straight forward to read for someone with no experience with the language.

It is not necessary to learn the full language in order to do basic logic functions. To learn just enough to perform basic logic, read these sections :

- [Program Structure](#)
- [Identifiers](#)
- [Constants](#)
- [Variables](#)
- [Types](#)
- [Assignment](#)
- [Operators](#)
- [C-Bus Functions](#)
- [If Then Statement](#)
- [Once Statement](#)
- [Modules](#)
- [Using the Logic Engine](#)

Most people will not need the following sections, except for very specialised circumstances :

- [Complex Data Types](#)
- [Files](#)
- [Socket \(TCP/IP\) IO](#)

4.1 Program Structure



Note that the [Logic Editor](#) automatically builds the structure of the [program](#) for you, so it is not necessary to remember the details of the program structure. The basic structure of a Pascal program is:

```
program ProgramName (FileList);

const
  { Constant declarations }

type
  { Type declarations }

var
  { Variable declarations }

  { Procedure and Function definitions }

begin
  { statements }
end.
```

Note that the words in **bold** above are "[reserved words](#)", which have particular meanings in Pascal.

The words in between the { and } braces are [Comments](#), and are used to explain what the program is doing or how it works.

4.2 Code Formatting

All spaces and end-of-lines are ignored by the Pascal compiler unless they are inside a string. However, to make your program readable by human beings, you should indent your statements and put separate statements on separate lines.

Since Pascal ignores end-of-lines and spaces, punctuation is needed to tell the compiler when a statement ends. You MUST have a semicolon following:

- each [constant](#) definition
- each [variable](#) declaration
- each [type](#) definition (to be discussed later)
- almost all [statements](#)

The last statement in the [program](#) or [Block](#), the one immediately preceding the END, does not require a semicolon. However, it's recommended to add one, as it saves you from having to add a semicolon if you have to move the statement higher up, or add another statement after it.

Indenting is not required. However, it is of great use for the programmer, since it helps to make the program clearer. If you wanted to, you could have a program look like this:

```
if (GetLightingLevel("Porch") > 50%) and (time = "9:00PM") then begin
SetLightingLevel("Porch", 50, 8); Delay("1:00:00"); if (GetLightingLevel("Porch")
> 0%) then SetLightingLevel("Porch", 0%, 8); end;
```

But it's much better for it to look like this:

```
if (GetLightingLevel("Porch") > 50%) and (time = "9:00PM") then
begin
    SetLightingLevel("Porch", 50, 8);
    Delay("1:00:00");
    if (GetLightingLevel("Porch") > 0%) then
        SetLightingLevel("Porch", 0%, 8);
end;
```

In general, indent two spaces for each block. Skip a line between blocks of code. Most importantly, use [comments](#) liberally. If you ever return to a program that you wrote a year ago, you probably wouldn't remember the logic unless you documented it.

A block of code can be indented by selecting the code then using the CTRL + | [Keyboard Shortcut](#).

4.3 Identifiers

Identifiers are names that allow you to reference stored values, such as [variables](#) and [constants](#).

Rules for identifiers:

- Must begin with a letter from the English alphabet.
- Can be followed by alphanumeric characters (alphabetic characters and numerals) and the underscore (_).
- May not contain any special characters including ~ ! @ # \$ % ^ & * () + ` - = { } [] : " ; ' < > ? , . / | \

Many identifiers are reserved in Pascal - you cannot use them as your own identifiers. These "keywords" are:

- [and](#)

- [array](#)
- [begin](#)
- [case](#)
- [const](#)
- [div](#)
- [do](#)
- [downto](#)
- [else](#)
- [end](#)
- [file](#)
- [for](#)
- [forward](#)
- [function](#)
- goto (not implemented in the logic engine)
- [if](#)
- [in](#)
- label (not implemented in the logic engine)
- [mod](#)
- nil (not implemented in the logic engine)
- [not](#)
- [of](#)
- [or](#)
- packed (not implemented in the logic engine)
- [procedure](#)
- [program](#)
- [record](#)
- [repeat](#)
- [set](#)
- [then](#)
- [to](#)
- [type](#)
- [until](#)
- [var](#)
- [while](#)
- [with](#)

Also, Pascal has many pre-defined identifiers. You can replace them with your own definitions, but this would be unwise as you would change part of the functionality of Pascal :

- [abs](#)
- [arctan](#)
- [boolean](#)
- [char](#)
- [cos](#)
- [dispose](#)
- [eof](#)
- [eoln](#)
- [exp](#)
- [false](#)
- input (not implemented in the logic engine)
- [integer](#)
- [new](#)
- [odd](#)
- [ord](#)
- output (not implemented in the logic engine)
- pack (not implemented in the logic engine)
- [pred](#)
- [read](#)

- [readln](#)
- [real](#)
- [reset](#)
- [rewrite](#)
- [round](#)
- [sin](#)
- [sqr](#)
- [sqrt](#)
- [succ](#)
- text (not implemented in the logic engine)
- [true](#)
- [trunc](#)
- [write](#)
- [writeln](#)

Pascal is not case sensitive. "MyProgram", "MYPROGRAM", and "mYpRoGrAm" are equivalent. But for readability purposes, it is a good idea to use meaningful capitalization.

Note that identifiers are not the same as [Tags](#).

4.4 Comments

Pascal comments either :

- start with a `(*` and end with a `*)` or
- start with a `{` and end with a `}`
- have `//` at the start of the comment and continue to the end of the line

You can nest the `{ }` comments within the `(* *)` comments or vice versa :

```
(* comment { nested comment } comment *)
{ comment (* nested comment *) comment }
```

You can also nest the `//` comments within the `{ }` or `(* *)` comments and vice versa, but note that everything after `//` on a line is ignored, including `}` and `*)`.

You can't nest the same type of comments :

```
{ comment { this won't work } comment }
```

Commenting has two purposes: first, it makes your code easier to understand. If you write your code without comments, you may come back to it a year later and have a lot of difficulty figuring out what you've done or why you did it that way. Another use of commenting is to figure out errors in your program. When you don't know what is causing an error in your code, you can comment out any suspect code segments. It is recommended using brace comments `{ }` in general, leaving the `(* *)` comments for debugging (which will also comment out any other comments in the code).

Examples

```
(* this
   is a
   multi-line
   comment *)

{ this
  is also a
  multi-line
  comment }
```

```

{ this is a single line comment }

// this is also a single line comment

SetEnableLevel(32 {this is a comment in the middle of some code}, 255);

SetEnableLevel(32, 255); // this is a comment at the end of some code

```

4.5 Constants

Constants are referenced by [identifiers](#), and can be assigned a fixed value at the beginning of the program. The value stored in a constant cannot be changed. Constants are defined in the constant section of a [Program](#).

The format of a constant definition is :

```
Identifier = value;
```

Examples:

```

KitchenLightAddress = 32;
pi = 3.1416;
LetterA = 'a';

```

Constants are useful when you want to use a number in your programs that you may wish to change in the future. If you are writing a program which controls the Kitchen Light (where the Kitchen Light Group Address is defined as above), then you can change the address of the Kitchen Light at any stage in the future by just changing the constant definition. That way, the rest of your code can remain unchanged because it refers to the constant.

See also [Integer](#), [Real](#), [Boolean](#), [Char](#) and [String](#) constant formats.

4.6 Variables

Variables store a value and their values can be changed as the program runs. Global variables (used by [Modules](#)) must be declared in the [Global Variables](#) section of the code before they can be used. The format is :

```
IdentifierList : Type;
```

IdentifierList is a series of [identifiers](#), separated by commas (,). All identifiers in the list are declared as being of the same data type.

For example :

```

Temperature, SetPoint : Real;
CounterValue : integer;
IrrigationRunning : boolean;
Name : string;

```

The above example will create :

- two [Real](#) variables - one called Temperature and the other called SetPoint
- an [Integer](#) variable called CounterValue
- a [Boolean](#) variable called IrrigationRunning
- a [String](#) variable called Name

It is recommended that variables be initialised before they are first used. This can be done in the [Initialisation](#) section.

See also [Using Counters](#).

4.7 Types

The basic data types in Pascal are:

- [integer](#)
- [real](#)
- [char](#)
- [boolean](#)

The Logic Engine also supports the [string](#) type.

The user can also create [Complex Data Types](#).

4.7.1 Integer Type

The Integer data type can contain integers (whole numbers) from -2147483648 to +2147483647.

In the Logic Engine, [dates](#) and [times](#) are expressed as integers.

Integer Constants

An integer constant is written as a number with an optional minus sign at the front.

In many cases, [Tags](#) can be used as integer constants.

Integer constant values can also use the Hexadecimal notation or be expressed as a percent, as described below.

The largest value which can be entered as a constant is 2147483599 or \$7FFFFFFF.

Hexadecimal Constants

[Hexadecimal](#) numbers can be represented by placing a dollar (\$) sign in front of a number. For example :

Hexadecimal Constant	Value
\$00	0
\$FF	255
\$0B00	2816
\$FFFFFF	16777215

The range is from \$0 to \$FFFFFF.

Percentage Constants

[C-Bus](#) Levels (from 0 to 255) can also be expressed as a percent (0 to 100). To express a level as a percent, place a percent sign (%) after the number. For example :

Percent Constant	Value
0%	0
50%	127
100%	255

Note that there are conversion functions to convert between C-Bus Levels and Percent :

- [LevelToPercent Function](#)
- [PercentToLevel Function](#)

4.7.2 Real Type

The real data type has a range from $\pm 1.5 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$.

Real Constants

Real values can be written in either fixed-point notation or in scientific notation, with the character E separating the mantissa from the exponent. Hence :

452.13 is the same as 4.5213e2
 0.001 is the same as 1E-3

4.7.3 Boolean Type

The Boolean data type can have only two values - true and false. Boolean variables are often called "flags".

Boolean Constants

The only valid Pascal Boolean constant values are TRUE and FALSE. The Logic Engine also allows the use of ON and OFF respectively.

4.7.4 Char Type

The char data type stores a single character.

Char Constants

The character is enclosed in single quotes, or apostrophes, such as :

'a' 'B' '+'

This data type can hold any [ASCII](#) or [Unicode](#) characters, including characters which can not be printed, such as Carriage Return (ASCII number 13). These are written with a hash (#) followed by the ASCII value in decimal or [hexadecimal](#).

To use the quote character, two quotes are written together within the outer quotes (as for [Strings](#)). This results in four quotes in a row (as shown below).

Examples

Character	Constant Format
A	'A'
%	'%'
Line Feed	#10 or #10A
Carriage Return	#13 or #13D
' (quote)	''
space	' '

4.7.5 String Type

A string represents a sequence of characters. The reserved word **string** defines a default, 50 character string. For example,

var

```
S: string;
```

creates a variable S that holds a string.

You can index a string variable just as you would an [Array](#). If S is a string variable and i an integer expression, S[i] represents the ith character in S. Be careful indexing strings in this way, since overwriting the end of a string can cause errors.

You can assign the value of a string constant, or any other expression that returns a string, to a string variable. The length of the string changes dynamically when the assignment is made.

Examples:

```
MyString := s;
MyString := 'Hello world!';
MyString := ' ';           { space }
MyString := "";           { empty string }
```

A string is actually a zero based [Array](#) of [characters](#). The standard declaration of a string (as above) is equivalent to :

```
var
  S: array[0..50] of char;
```

The first character of the string (s[0]) stores the length of the string. To define strings of other lengths, define a zero based array of the appropriate length. For example, for a string of 20 characters :

```
var
  S: array[0..20] of char;
```

String Constants

A character string, also called a string literal or string constant, consists of a quoted string, a control string, or a combination of quoted and control strings. Separators can occur only within quoted strings.

A quoted string is a sequence of up to 255 characters from the [ASCII](#) or [Unicode](#) character set, written on one line and enclosed by apostrophes. A quoted string with nothing between the apostrophes is a null string. Two sequential apostrophes in a quoted string denote a single character, namely an apostrophe (as shown in the examples below).

A control string is a sequence of one or more control characters, each of which consists of the # symbol followed by an unsigned integer constant from 0 to 255 and denotes the corresponding ASCII character. The control string

```
#89#111#117
```

is equivalent to the quoted string

```
'You'
```

You can combine quoted strings with control strings to form larger character strings. For example, you could use

```
'Line 1'#13#10'Line 2'
```

to put a carriage-return-line-feed between "Line 1" and "Line 2". However, you cannot concatenate two quoted strings in this way, since a pair of sequential apostrophes is interpreted as a single character. (To concatenate quoted strings, use the [Append](#) function, or simply combine them into a single quoted string.)

Examples

string	interpretation
'hello'	{ hello }
'You'll see'	{ You'll see }
''''	{ ' } }
"	{ null string }
' '	{ a space }
#89#111#117	{ You }
'hell'#111	{ hello }

4.8 Assignment

Once you have [declared](#) a variable, you can store values in it. This is called assignment.

To assign a value to a variable, the syntax is:

```
variable_name := expression;
```

The expression can either be a single value, such as :

```
SetPoint := 38.5;
```

or it can be a complex expression containing [Operators](#), such as :

```
SetPoint := 73.5 * 37 + 35.8 / 67.1;
```

Each variable can only be assigned a value that is of the same data type. Thus, you cannot assign a real value to an integer variable. However, certain data types are compatible with others. In these cases, you can assign a value of a lower data type to a variable of a higher data type. This is most often done when assigning integer values to real variables. Suppose you had this variable declaration section:

```
var
  some_int : integer;
  some_real : real;
```

When the following block of statements executes,

```
some_int := 375;
some_real := some_int;
```

some_real will have a value of 375.0.

4.9 Displaying Data

For writing data to the Logic Engine [Output Window](#), there are two statements which can be used:

```
Write(Argument_List);
WriteLn(Argument_List);
```

These statements write the values of the parameters in the argument list to the screen. The WriteLn (Write Line) statement skips to the next line when done.

Example

For example, to write the text 'Counter =' followed by the value of the variable called MyCounter :

```
WriteLn('Counter = ', MyCounter);
```

If the value of MyCounter was 123, then the Output Window would have the following line written to it :

```
Counter =      123
```

Formatting the Output

Formatting the output is quite easy. For each identifier or literal value on the argument list, use:

```
Value : field_width
```

The output is right-justified in a field of the specified integer width. If the width is not long enough for the data, the width specification will be ignored and the data will be displayed in its entirety (except for real values - see below).

Suppose we had:

```
write('Hi':10, 5:4, 5673:2);
```

The output (with a dash simulating the space) would be:

```
-----Hi---55673
```

For real values, you can use the aforementioned syntax to display scientific notation in a specified field width, or you can convert to fixed notation using the format :

```
Value : field_width : decimal_field_width
```

The field width is the total field width, including the decimal part. The whole number part is always displayed fully, so if you have not allocated enough space, it will be displayed anyway. However, if the number of decimal digits exceeds the specified decimal field width, the output will be rounded to the specified number of places (though the variable itself is not changed). So :

```
write(573549.56792:20:2);
```

would look like:

```
-----573549.57
```

Pascal also supports the Read and ReadLn functions for the purpose of reading data from the keyboard. This is not supported by the Logic Engine, since it is not relevant.

4.9.1 Tutorial 1

In the tutorial questions, the sections that the code segments belong in are shown in comments. For example, a snippet of code starting with { var } belongs in the variable declaration section of a program. Where irrelevant bits of code have been skipped, the "..." symbol is used.

Question 1

Which of the following are valid Identifiers ?

```
name
case
```

light level
light_level
group#

Question 2

Write a statement to display the text 'Level = ', followed by the value of the variable Level.

Question 3

What are the numerical values of the following constants :

1.2E3
\$12
100%

Question 4

Which of the basic [Types](#) are the following constants :

123.45
100
true
'a'
'hello'

Question 5

Write [Variable](#) Declarations for the following :

An integer called "total".
A string called "message".
A variable to hold currency called "cost".
A variable to store whether an error has occurred, called "error".

Question 6

Find the 6 errors in the code below.

```
{ var }  
number1, number 2; integer;  
...  
{ main program }  
number1 = 12;  
number2 := 2.3  
WriteLn('number2 =, number2);
```

Question 7

A real variable called Level contains the level of a light bulb (from 0 to 255). The light bulb is 100W.
Write a statement to display the power level of the light, with one decimal place.

Question 8

Write a statement to add 1 to the value of a variable called Count.

[Tutorial Answers](#)

4.10 Operators

Operators are used for performing [mathematical operations](#) (such as addition or multiplication) or [comparisons](#) (for example to check if two numbers are equal or not).

When combining operators, it is important to understand [Operator Precedence](#).

An operator performs an operation on two or more operands. The type of operand depends on the operator [type](#).

4.10.1 Arithmetic Operators

The arithmetic operators in Pascal are:

Operator	Operation	Operands	Result
+	Addition or unary positive	real or integer	real or integer
-	Subtraction or unary negative	real or integer	real or integer
*	Multiplication	real or integer	real or integer
/	Real division	real or integer	real
div	Integer division	integer	integer
mod	Modulus (remainder division)	integer	integer

The div and mod operators only work on integers.

The / works on both reals and integers but will always yield a real answer.

The other operations work on both reals and integers.

For operators that accept both reals and integers, the resulting data type will be integer only if all the operands are integer. It will be real if any of the operands are real. Therefore,

`3857 + 68348 * 38 div 56834` will be integer, but

`38573 div 34739 mod 372 + 35730 - 38834 + 1.1` will be real because 1.1 is a real value.

In Pascal, the minus sign can be used to make a value negative. For example :

```
some_real := -15;
```

will result in the some_real variable having a value of minus 15.

Do not attempt to use two operators adjacent to each other, such as :

```
some_real := 37.5 * -2;
```

This may make perfect sense to you, since you're trying to multiply by negative 2. However, Pascal will be confused -- it won't know whether to multiply or subtract. You can avoid this by using parentheses:

```
some_real := 37.5 * (-2);
```

to make it clearer.

Note : [Bitwise Operators](#) can also be applied to Integers.

Example

To do something every 5 seconds:

```
once RunTime mod 5 = 0 then
begin
...
end;
```

4.10.2 Relational Operators

Relational operators are used to compare two values. The syntax of a Boolean expression is :

```
value1 RelationalOperator value2
```

The result of a boolean expression is a boolean value (TRUE or FALSE).

The following relational operators are used:

Operator	Meaning
<	less than
>	greater than
=	equal to
<=	less than or equal to
>=	greater than or equal to
<>	not equal to

There are also other boolean operators which are used for [Sets](#).

You can assign Boolean expressions to Boolean variables:

```
SomeBool := 3 < 5;
```

In this case, the value of SomeBool becomes TRUE.

Examples

To perform an action if x is greater than 5 :

```
if x > 5 then ...
```

Whenever possible, don't compare two real values with the equals sign. Small round-off errors may cause two equivalent expressions to differ. If you want to determine whether two variables x1 and x2 are within 0.001 of each other, use :

```
if abs(x1 - x2) < 0.001 then ...
```

4.10.3 Char and String Operators

When [Character](#) variables are compared against each other (using [Relational Operators](#)), they are converted to the [ASCII](#) value of the character before the comparison is made. So the expression :

```
'A' < 'B'
```

is TRUE.

[Strings](#) can also be compared. When two strings are compared, they are compared character by character until either :

- the characters do not match
- the end of one or both strings is reached

Examples

The following expressions are all TRUE :

```
'abcd' < 'abcx'
'abc' < 'abcd'
'abc' <> 'ABC'
'abc' = #97#98#99 (see String Constants)
```

4.10.4 Boolean Operators

Complex Boolean expressions are formed by using the Boolean operators:

Operator	Meaning
not	negation / inverse
and	conjunction
or	disjunction
xor	exclusive-OR

NOT Operator

The NOT operator is a unary operator - it is applied to only one value and inverts it. So for example, with a boolean expression "A", the value of "not A" can be found from :

A	not A
true	false
false	true

AND Operator

The AND operator yields TRUE only if both expressions are TRUE. So for example, with boolean expressions "A" and "B", the value of "A and B" can be found from :

A	B	A and B
false	false	false
false	true	false
true	false	false
true	true	true

Where a series of operands are used with AND, the result is true if all of the operands are true.

OR Operator

The OR operator yields TRUE if either expression is TRUE, or if both are. So for example, with boolean expressions "A" and "B", the value of "A or B" can be found from :

A	B	A or B
false	false	false

false	true	true
true	false	true
true	true	true

Where a series of operands are used with OR, the result is true if any of the operands are true.

XOR Operator

The XOR operator yields TRUE if one expression is TRUE and the other is FALSE. So for example, with boolean expressions "A" and "B", the value of "A xor B" can be found from :

A	B	A xor B
false	false	false
false	true	true
true	false	true
true	true	false

Examples

If you want to do something when both group 1 and group 2 are both on, the code would be:

```
if (GetLightingState(1) = ON) and (GetLightingState(2) = ON)
then...
```

If you want to do something when both group 1 and group 2 are both off, the code would be:

```
if (GetLightingState(1) = OFF) and (GetLightingState(2) = OFF)
then...
```

If you want to do something when either group 1 or group 2 is on, the code would be:

```
if (GetLightingState(1) = ON) or (GetLightingState(2) = ON)
then...
```

If you want to do something when either group 1 is on or group 2 is on, but not both, the code would be:

```
if (GetLightingState(1) = ON) xor (GetLightingState(2) = ON)
then...
```

If you want to do something except when both group 1 and group 2 are both on, the code would be:

```
if not ((GetLightingState(1) = ON) and (GetLightingState(2) = ON))
then...
```

Note that if you apply [DeMorgan's Rule](#), the above is equivalent to:

```
if (GetLightingState(1) = OFF) or (GetLightingState(2) = OFF)
then...
```

See also [Operator Precedence](#) and [Simplifying Logic Conditions](#)

4.10.5 Bitwise Operators



The bitwise arithmetic operators in Pascal are:

Operator	Operation	Operands	Result
shl	Bitwise Shift Left	integer	integer
shr	Bitwise Shift Right	integer	integer
or	Bitwise logical OR	integer	integer
and	Bitwise logical AND	integer	integer
xor	Bitwise logical XOR	integer	integer
not	Bitwise logical NOT	integer	integer

The "bitwise" operators operate on the individual bits (binary digits) of an integer. To understand the usage of these operators, it is necessary to understand [binary](#) numbers. The examples below all show values in binary to clarify what is happening.

Shift Left Operator

Syntax

Value shl NumberOfBits

The Shift Left operation moves all of the bits in the Value left by a specified NumberOfBits. The result is equal to multiplying Value by $2^{\text{NumberOfBits}}$.

Example

```
A          00000011
A shl 2    00001100
```

Shift Right Operator

Syntax

Value shr NumberOfBits

The Shift Right operation moves all of the bits in the Value right by a specified NumberOfBits. The result is equal to dividing Value by $2^{\text{NumberOfBits}}$.

Example

```
A          00010000
A shr 2    00000100
```

OR Operator

Syntax

A or B

The logical [OR](#) operation when applied to integers A and B results in a value where each bit is the

logical OR of the corresponding bits in A and B.

Example

A	00110000
B	00000011
A OR B	00110011

AND Operator

Syntax

A and B

The logical **AND** operation when applied to integers A and B results in a value where each bit is the logical AND of the corresponding bits in A and B.

Example

A	00111111
B	00000011
A AND B	00000011

XOR Operator

Syntax

A xor B

The logical **XOR** operation when applied to integers A and B results in a value where each bit is the logical XOR of the corresponding bits in A and B.

Example

A	00111100
B	00001111
A XOR B	00110011

NOT Operator

Syntax

not A

The logical **NOT** operation when applied to an integer A results in a value where each bit is the logical inverse (NOT) of the corresponding bits in A.

Example

A	00110000
NOT A	11001111

4.10.6 Operator Precedence

The Logic Engine follows an order of operations the same as used for algebra. The computer looks at each expression according to these rules, in this order :

- Evaluate all expressions in parentheses, starting from the innermost set of parentheses and proceeding to the outermost.
- Evaluate all multiplication and division from left to right.
- Evaluate all addition and subtraction from left to right.

For example, the expression :

```
some_int := 1 + 2 * 3;
```

is interpreted as :

```
some_int := 1 + (2 * 3);
```

and will result in some_int being 7. The expression :

```
some_int := (1 + 2) * 3;
```

will result in some_int being 9.

The priority given to the various operators, from highest to lowest, are:

Priority	Operators
Highest Priority	NOT, Negation
High Priority	*, /, DIV, MOD, AND, SHL, SHR
Low Priority	+, -, OR, XOR
Lowest Priority	=, <>, <, <=, >, >=, IN

Important Note

When combining two Boolean expressions using [relational](#) and [Boolean operators](#), be careful to use parentheses. For example:

```
(x > 5) or (y < 1)
```

This is because the Boolean operators are higher on the order of operations than the relational operators. The expression :

```
x > 5 or y < 1
```

would be evaluated as

```
x > (5 or y) < 1
```

which makes no sense.

Example 1

To perform an action if the variables x and y are both greater than 0 :

```
if (x > 0) and (y > 0) then ...
```

To perform an action if either of the variables x and y are greater than 0 :

```
if (x > 0) or (y > 0) then ...
```

Example 2

If you want something to happen at 9PM if either group 1 or group 2 is on, the code would be :

```
once ((GetLightingState(1) = ON) or (GetLightingState(2) = ON))
and (Time = "9:00PM") then...
```

If you wrote it as :

```
once (GetLightingState(1) = ON) or (GetLightingState(2) = ON) and
(Time = "9:00PM") then...
```

then the action would occur under either of the following conditions:

- Group 1 is on
- Group 2 is on AND the time is 9PM

which is not what was required.

This is because the order of evaluation is:

- Contents of brackets first
- "and" is evaluated next
- "or" is evaluated last

4.10.7 Tutorial 2

Question 1

Given that

```
A := 1;      B := 2;      C := 4;
```

What does X equal after each of the following statements,

```
X := A / B / C;
```

```
X := A + B / C;
```

```
X := A * B * C;
```

```
X := A * B - C;
```

```
X := A + B + C;
```

```
X := A / B * C;
```

```
X := A * B / C;
```

```
X := A + B - C;
```

Question 2

Write statements in Pascal which correctly express each of the following mathematical expressions.

$$Z = X + Y^2$$

$$Z = (X + Y)^2$$

$$Z = \frac{A + B + E}{D + E}$$

$$Z = A + \frac{B}{C}$$

$$Z = \frac{A + B}{C}$$

$$Z = A + \frac{B}{D - C}$$

Question 3

Which of the following statements is wrong and why ?

```

Y := 2X + A;
4 := X - Y;
A := 1 / ( X + ( Y - 2 ) );
-J := K + 1;
S := T / * 3;
Z + 1 := A;

```

Question 4

If the following integer variables are assigned :

```

a := 1;
b := 2;
c := 3;
d := 3;

```

What are the boolean values of the following expressions ?

- 1 a > b
- 2 a = c
- 3 b <> c
- 4 c = d
- 5 not (a < d)
- 6 (a = b) or (c = d)
- 7 (a = b) and (c = d)
- 8 (a < 10) and not (c < d)
- 9 (a > 0) or (b > 0) and (c = 0)

[Tutorial Answers](#)**4.11 Standard Functions**

The Pascal language provides a range of functions to perform data transformation and calculations. The following sections provides an explanation of the standard functions.

The Logic Engine adds a lot of other functions which are useful for automation and control. These are described in subsequent sections.

4.11.1 Mathematical Functions

The following mathematical functions are included with the Logic Engine :

- [Abs Function](#)
- [Exp Function](#)
- [Ln Function](#)
- [Odd Function](#)
- [Random Function](#)
- [Round Function](#)
- [Sqr Function](#)

- [Sqrt Function](#)
- [Trunc Function](#)

4.11.1.1 Abs Function

The ABSolute function returns the absolute value of either an integer or real variable

Syntax

`abs(x)`

Description

The absolute function makes a negative number positive, but does not affect positive values.

The result of the function will be of the same type as the argument. Hence the Abs value of an integer will be an integer, and the Abs value of a real expression will be a real value.

Example

`Abs(-21)` returns 21
`Abs(-3.5)` returns 3.5

To assign the absolute value of variable x to a variable n :

`n := abs(x);`

4.11.1.2 Exp Function

The exponential function calculates e (the base of natural logarithms) raised to the power of the argument.

Syntax

`exp(x)`

Where the argument (x) is a [real](#) expression and the result is real.

Example

`Exp(10)` returns e to the power of 10

To assign the value of e^x to a variable n :

`n := exp(x);`

There is no function in Pascal to calculate expressions such as a^x . These are calculated by using the formula

`ax = exp(x * ln(a))`

To assign the value of a^x to a variable n :

`n := exp(x * ln(a));`

See also [Power Function](#)

4.11.1.3 Ln Function

The logarithm function calculates the natural log of a number greater than zero.

Syntax

```
ln(x)
```

Where the argument (x) is a real expression and the result is real.

Example

To assign the logarithm of a variable x to a variable n :

```
n := ln(x);
```

4.11.1.4 Odd Function

The Odd function determines when a specified number is odd.

Syntax

```
odd(x)
```

Where the argument (x) is an integer expression and the result is [Boolean](#).

Description

The Odd function returns a result of true when the argument is odd (ie. 1, 3, 5, 7, 9, ...) and false when it is not (ie. it is even).

Example

To perform an action if a variable n is odd :

```
if odd(n) then ...
```

To perform an action if a variable n is even :

```
if not odd(n) then ...
```

4.11.1.5 Random Function

The Random function generates a random number.

Syntax

```
random(x)
```

Where the argument (x) is an integer expression and the result is an integer.

Description

The Random function generates a pseudo-random number between 0 and the value of the argument (including 0, but not including the value of the argument).

Example

To assign a random number between 0 and 99 to a variable n :

```
n := random(100);
```

To assign a random number between 1 and 100 to a variable n :

```
n := random(100) + 1;
```

See also [Random Event Times](#).

4.11.1.6 Round Function

This function returns the whole part (ie no decimal places) of a real number.

Syntax

```
round(x)
```

Where the argument (x) is a [real](#) expression and the result is [integer](#).

Description

The round function rounds its argument to the nearest integer. If the argument is positive :

- rounding is up for fractions greater than or equal to .5
- rounding is down for fractions less than .5

If the number is negative :

- rounding is down (away from zero) for fractions greater than or equal to .5
- rounding is up (towards zero) for fractions less than .5

Example

```
round(4.87)    returns 5  
round(-3.4)   returns -3
```

To assign the rounded value of variable x to a variable n :

```
n := round(x);
```

4.11.1.7 Sqr Function

The square function returns the square (ie the argument multiplied by itself) of its argument.

Syntax

```
sqr(x)
```

Description

The result of the function will be of the same type as the argument. Hence the Sqr value of an integer will be an integer, and the Sqr value of a real expression will be a real value.

Example

```
Sqr(2)         returns the value 4
```

To assign the square of a variable x to a variable n :

```
n := sqr(x);
```

See also [Exp Function](#) and [Power Function](#)

4.11.1.8 Sqrt Function

This function returns the square root of its argument.

Syntax

`sqrt(x)`

Where the argument (x) is an [integer](#) or real expression and the result is [real](#).

Example

`Sqrt(4)` returns 2.0

To assign the square root of a variable x to a variable n :

```
n := sqrt(x);
```

4.11.1.9 Trunc Function

This function returns the whole part (ie no decimal places) of a real number.

Syntax

`trunc(x)`

Where the argument (x) is a [real](#) expression and the result is [integer](#).

Example

`Trunc(4.87)` returns 4

`Trunc(-3.4)` returns -3

To assign the truncated part of variable x to a variable n :

```
n := trunc(x);
```

4.11.1.10 Power Function

The power function returns the value of a number raised to the power of another number.

Syntax

`power(X, Y)`

x and y are [Real](#)

Description

The result of the function is a real value equal to X^Y .

Example

`power(2, 3)` returns the value 8

`y := power(x, 4);` assigns the fourth power of x to the variable y

See also [Exp Function](#) and [Sqr Function](#)

4.11.2 Trigonometric Functions

Pascal has several standard trigonometric functions that you can utilize. For example, to find the value of sin of π radians,

```
value := sin(3.14159);
```

For all trigonometry functions, the angular measurements are always in radians. There are 2π radians in 360° .

The trigonometry functions in the Logic Engine are :

- [Sin Function](#)
- [Cos Function](#)
- [ArcTan Function](#)

To use other trigonometric functions, it is necessary to combine the above functions. For example, to find the tangent of a variable x :

```
tan := sin(x) / cos(x);
```

To convert from degrees to radians, divide by 180 then multiply by π . For example, to calculate the cosine of 270 degrees :

```
value := cos(270 / 180 * Pi);
```

Note that there is a [Constant](#) called Pi.

4.11.2.1 Sin Function

The SINE function returns the sine of its argument (in radians).

Syntax

```
sin(x)
```

Where the argument (x) is a real expression and the result is real.

Example

To assign the sine of a variable x to a variable n :

```
n := sin(x);
```

4.11.2.2 Cos Function

The COSINE function returns the cosine value, of its argument (in radians).

Syntax

```
cos(x)
```

Where the argument (x) is a real expression and the result is real.

Example

To assign the cosine of a variable x to a variable n :

```
n := cos(x);
```

4.11.2.3 ArcTan Function

The ARCTANGENT function returns the arc tangent value, in radians, of its argument.

Applicability

Colour C-Touch only.

Syntax

```
ArcTan(x)
```

Where the argument (x) is a real expression and the result is real.

Example

To assign the arc tangent of a variable x to a variable n :

```
n := arctan(x);
```

4.11.3 Ordinal Functions

For ordinal data types ([integer](#), [char](#) or [Enumerated Types](#)), which have a distinct predecessor and successor, you can use these functions:

- [Chr Function](#)
- [ChrW Function](#)
- [Ord Function](#)
- [ChrW Function](#)
- [Pred Function](#)
- [Succ Function](#)

Note that Real is not an ordinal data type. That's because it has no distinct successor or predecessor. What is the successor of 56.0? It could be 56.1, 56.01, 56.001, 56.0001, 56.00001, 56.000001 or any other value.

However, for an integer 56, there is a distinct predecessor (55) and a distinct successor (57).

The same is true of characters. 'b' has a successor ('c') and a predecessor ('a').

4.11.3.1 Chr Function

The chr function returns the character for a specified [ASCII](#) value.

Syntax

```
chr(x)
```

Where the argument (x) is an [integer](#) expression and the result is a [char](#).

Description

Chr returns the character with the ordinal value (ASCII value) of the integer expression, x. x must be between 0 and 255.

Example

```
chr(65) equals 'A'
```

To assign a char with the ordinal value of variable x to a variable char1 :

```
char1 := chr(x);
```

See also [ChrW Function](#)

4.11.3.2 ChrW Function

The ChrW function returns the character for a specified [Unicode](#) value.

Applicability

Colour C-Touch only.

Syntax

```
ChrW(x)
```

Where the argument (x) is an [integer](#) expression and the result is a [char](#).

Description

ChrW returns the character with the ordinal value of the integer expression, x. x must be between 0 and 65535.

Example

```
chrw(65)      equals 'A'
chrw(8364)   equals '€'
```

To assign a char with the ordinal value of variable x to a variable char1 :

```
char1 := chrw(x);
```

See also [Chr Function](#)

4.11.3.3 Ord Function

The ord function returns the ordinal value of an ordinal-type expression.

Syntax

```
ord(c)
```

Description

c is an [ordinal type](#) expression. The result is an integer, and its value is the ordinal position of c. If c is a [Char Type](#), then ord returns the [ASCII](#) value.

Example

```
ord('A')      equals 65 (ASCII value)
ord(Tuesday)  equals 1 - where the enumerated type (Monday, Tuesday, Wednesday,
Thursday, Friday) is defined
```

To determine whether char c is an [ASCII](#) carriage return character :

```
if ord(c) = 13 then ...
```

See also [OrdW Function](#)

4.11.3.4 OrdW Function

The OrdW function returns the ordinal value of a [Unicode](#) character expression.

Applicability

Colour C-Touch only.

Syntax

`ordw(c)`

Description

`c` is a [Unicode](#) character. The result is an integer, and its value is the Unicode value of `c`.

Example

```
ordw('A')    equals 65 (ASCII / Unicode value)
ordw('€')    equals 8364 (Unicode value)
```

To determine whether char `c` is a Euro character (€) :

```
if ordw(c) = 8364 then ...
```

See also [Ord Function](#)

4.11.3.5 Pred Function

The `pred` function returns the predecessor of the argument.

Syntax

`pred(n)`

Description

`n` is an expression of an [ordinal type](#). The result, of the same type as `n`, is the predecessor of `n`.

Example

```
pred(20)      equals 19
pred('C')     equals 'B'
pred(Tuesday) equals Monday - where the enumerated type (Monday, Tuesday,
Wednesday, Thursday, Friday) is defined
```

4.11.3.6 Succ Function

The `succ` function returns the successor of the argument.

Syntax

`succ(n)`

Description

`n` is an expression of an [ordinal type](#). The result, of the same type as `n`, is the successor of `n`.

Example

```
succ(20)      equals 21
```

```

succ('C')      equals 'D'
succ(Tuesday) equals Wednesday - where the enumerated type (Monday, Tuesday,
Wednesday, Thursday, Friday) is defined

```

4.11.4 Tutorial 3

Question 1

What is the value assigned to the variables in the following :

```

1.   int1 := abs(-10);
2.   bool1 := odd(5);
3.   int1 := random(10) + 5;
4.   int1 := round(3.5);
5.   int1 := trunc(3.5);
6.   char1 := chr( ord('A') + 2);
7.   int1 := succ(4);

```

Tutorial Answers

4.12 Tags

For many Logic Engine functions (not standard Pascal functions), it is possible to use a name in place of an integer index. For example, instead of

```
SetScene(23);
```

you can use

```
SetScene("All Off");
```

which makes the purpose much more clear.

This name is called a "tag", and is some text within double quotes. When the project is [Compiled](#), the tag is turned into an [integer](#), and when the project is [Run](#), the integer value is used.

Note that tags are very different from [Strings](#), and the two are not interchangeable. A string is some text which is used in the program. A tag represents a number.

Tags (other than [C-Bus Tags](#)) are case sensitive, hence the following tags are not equivalent : "Schedule 1", "schedule 1", "SCHEDULE 1".

Tags may have spaces or any other character (unlike [Identifiers](#), which can only have specific characters). Hence, the following are valid tags : "Kitchen Light", "Light #3", "Lounge & Dining".

Tags can be used for :

- [C-Bus](#) Networks, Applications, Group Addresses, Levels and Ramp Rates
- Page Names
- Scene Names
- [System IO Variable](#) Names
- [Module](#) Names
- [Dates](#)
- [Time](#)
- [Images](#)
- [Special Day Names](#)
- Special Functions
- Page Transition Effects

- [Pulse Power Meter Names](#)
- [Energy Tariff Names](#)
- [Profile Name](#)

4.13 Date Functions

The Logic Engine represents dates with [Integers](#). The value is the number of days that have passed since December 30 1899.

Following are some examples of Date numeric values and their corresponding dates:

Value	Date
0	Dec 30 1899
2	Jan 1 1900
-1	Dec 29 1899
35065	Jan 1 1996

To find the number of days between two dates, simply subtract the two values. Likewise, to increment a date value by a certain number of days, simply add the number to the date value.

The Logic Engine provides the following functions for manipulating Dates :

- [Date Function](#)
- [DateToString Procedure](#)
- [Day Function](#)
- [DayOfWeek Function](#)
- [DayOfYear Function](#)
- [DecodeDate Procedure](#)
- [EncodeDate Function](#)
- [Month Function](#)
- [Year Function](#)

[Tags](#) can be used for dates, month names and days of the week. **The interpretation of the date tags depends upon the date format selected in the Windows Control Panel.** Example date tags in this document are given in Day/Month/Year format.

The Month and Day names can be complete, or just the first three letters.

For example :

Tag	Value
"1/1/1996"	35065
"1 Jan 2003"	35065
"January"	1
"Feb"	2
"Monday"	2
"Tue"	3
"2 Jan"	2

Note: PAC and Black and White C-Touch will only support dates since 1 Jan 2000.

See also [Time Functions](#)

4.13.1 Date Function

The Date function returns an integer which represents the current [Date](#).

Syntax

date

Description

The value of the Date function is the number of days which have passed since 30 Dec 1899. Note that the PAC can not use dates prior to 1 Jan 2000.

Example

To assign the current date to a variable n :

```
n := date;
```

To perform an action if the date is Jan 1 2010 :

```
if date = "1/1/2010" then ...
```

See also [Date Tags](#).

4.13.2 Day Function

The Day function returns an integer which represents the current day of the month (1 to 31).

Syntax

day

Description

The value of the Day function is the day of the month. See also [DayOfYear Function](#).

Example

To assign the current day of the month to a variable n :

```
n := day;
```

To perform an action if the date is July 14th :

```
if (day = 14) and (month = "July") then ...
```

4.13.3 DayOfWeek Function

The DayOfWeek function returns an integer which represents the current day of the week.

Syntax

DayOfWeek

Description

The values representing the days of the week are :

Sunday = 1 ... Saturday = 7

Example

To assign the current day of the week to a variable n :

```
n := DayOfWeek;
```

To perform an action if the day of the week is Tuesday :

```
if DayOfWeek = 3 then ...
```

or

```
if DayOfWeek = "Tuesday" then ...
```

See also [Date Tags](#).

4.13.4 DayOfYear Function

The DayOfYear function returns an integer which represents the current day of the year (1 to 366).

Syntax

```
DayOfYear
```

Description

The value of the DayOfYear function is the number of the day of the year. The function treats every year as if it is a leap year (ie. has a Feb 29th) in order for the date to have the same day number each year. This simplifies comparisons.

Example

To assign the current day of the year to a variable n :

```
n := DayOfYear;
```

To perform an action if the day of the year is April 18th :

```
if DayOfYear = "18 Apr" then
```

```
...
```

4.13.5 DecodeDate Procedure

The DecodeDate procedure decodes a [Date](#) value to give the day, month and year.

Syntax

```
DecodeDate(Date1, Year1, Month1, Day1);
```

Date1 is an [Integer](#) expression

Day1, Month1, Year1 are integer variables

Description

The DecodeDate procedure is used to extract the day, month and year from a numerical date.

Example

The code :

```
Date1 := 35065; { 1/1/1996 }
```

```
DecodeDate(Date1, y, m, d);
```

results in y = 1996, m = 1, d = 1

4.13.6 EncodeDate Function

The EncodeDate function encodes a [Date](#) value given the day, month and year.

Syntax

```
EncodeDate(Year1, Month1, Day1)
```

Day1, Month1, Year1 are integer expressions

Description

The EncodeDate procedure is used to convert the day, month and year to a numerical date.

Example

To encode the date 1 Jan 1996 an assign it to a variable called Date1 :

```
Date1 := EncodeDate(1996, 1, 1);
```

This results in Date1 becoming 35065.

4.13.7 Month Function

The Month function returns an integer which represents the current month of the year (1 to 12).

Syntax

```
month
```

Description

The value of the Month function is the month of the year. See also [DayOfYear Function](#).

Example

To assign the current month to a variable n :

```
n := month;
```

To perform an action if the month is February :

```
if Month = "Feb" then
  ...
```

See also [Date Tags](#).

4.13.8 Year Function

The Year function returns an integer which represents the current year.

Syntax

```
year
```

Example

To assign the current year to a variable n :

```
n := year;
```

4.14 Time Functions

The Logic Engine represents times with [Integers](#). The value is the number of seconds that have passed since midnight.

Following are some examples of Time numeric values and their corresponding times:

Value	Time (24 hour)	Time (AM/PM)
0	00:00:00	12:00:00 AM
1	00:00:01	12:00:01 AM
43200	12:00:00	12:00:00 PM
86399	23:59:59	11:59:59 PM

To find the number of seconds between two times, simply subtract the two values. Likewise, to increment a time value by a certain number of seconds, simply add the number to the time value.

The Logic Engine provides the following functions for manipulating Times :

- [DecodeTime Procedure](#)
- [EncodeTime Function](#)
- [Hour Function](#)
- [Minute Function](#)
- [Second Function](#)
- [Sunrise Function](#)
- [Sunset Function](#)
- [Time Function](#)
- [TimeToString Procedure](#)

See also [Timer Functions](#) and [Date Functions](#)

[Tags](#) can be used for the time, instead of using a number. **The interpretation of the time tags depends upon the time format selected in the Windows Control Panel.** Example time tags in this document generally use AM/PM format.

Examples :

"7:00PM"
"23:00:00"

4.14.1 DecodeTime Procedure

The DecodeTime procedure decodes a [Time](#) value to give the hour, minute and second.

Syntax

```
DecodeTime(Time1, Hour1, Min1, Sec1);
```

Time1 is an [Integer](#) expression

Hour1, Min1 and Sec1 are integer variables

Example

The code :

```
Time1 := 43200; { 12:00:00 noon }  
DecodeTime(Time1, h, m, s);
```

results in h = 12, m = 0, s = 0

4.14.2 EncodeTime Function

The EncodeTime function encodes a [Time](#) value to given the hour, minute and second.

Syntax

```
EncodeTime(Hour1, Min1, Sec1)
```

Hour1, Min1 and Sec1 are integer expressions

Example

To encode the time 12:00 noon and assign in to a variable called Time1 :

```
Time1 := EncodeTime(12, 0, 0);
```

This results in Time1 becoming 43200.

4.14.3 Hour Function

The Hour function returns an integer which represents the current hour of the day (0 to 23).

Syntax

```
hour
```

Example

To assign the current hour of the day to a variable n :

```
n := hour;
```

4.14.4 Minute Function

The Minute function returns an integer which represents the current minute of the hour (0 to 59).

Syntax

```
minute
```

Example

To assign the current minute of the hour to a variable n :

```
n := minute;
```

4.14.5 RunTime Function

The RunTime function returns an integer which represents the number of seconds which have elapsed since the logic started running.

Syntax

```
RunTime
```

Example

To do something 10 seconds after start-up :

```
once RunTime = 10 then...
```

The RunTime function can be used a bit like another timer. If we wanted to determine the time which a Group Address has been on :

```
once GetLightingState("Main Light") = ON then
  StartTime = RunTime;
```

```

once GetLightingState("Main Light") = OFF then
begin
  ElapsedTime = RunTime - StartTime;
  ...
end;

```

4.14.6 Second Function

The Second function returns an integer which represents the current second of the minute (0 to 59).

Syntax

```
second
```

Example

To assign the current second of the minute to a variable n :

```
n := second;
```

To perform an action at the start of every minute :

```
if Second = 0 then ...
```

4.14.7 Sunrise Function

The Sunrise function returns an integer which represents the [Time](#) of sunrise.

Syntax

```
sunrise
```

Description

The sunrise time depends on :

- The location (longitude and latitude) - this can be set in the Project Details
- The date
- Daylight Savings - this can be set in the Project Details

Example

To assign today's sunrise time to a variable n :

```
n := sunrise;
```

To do something is the time is an hour before sunrise :

```
if time = Sunrise - "1:00:00" then ...
```

To do something if the time is after sunset but before sunrise (i.e. it is dark outside) :

```
if (time > sunset) or (time < sunrise) then...
```

Note that the following will not work, because it is not possible for a time to be both greater than sunset and less than sunrise :

```
if (time > sunset) and (time < sunrise) then... { do NOT do this }
```

To do something if the time is after sunrise but before sunset (i.e. it is light outside) :

```
if (time > sunrise) and (time < sunset) then...
```

4.14.8 Sunset Function

The Sunset function returns an integer which represents the [Time](#) of sunset.

Syntax

```
sunset
```

Description

The sunset time depends on :

- The location (longitude and latitude) - this can be set in the Project Details
- The date
- Daylight Savings - this can be set in the Project Details

Example

To assign today's sunset time to a variable n :

```
n := sunset;
```

To do something is the time is an hour after sunset :

```
if time = Sunset + "1:00:00" then ...
```

See also [Sunrise Function](#) examples

4.14.9 Time Function

The Time function returns an integer which represents the current [Time](#) (0 to 86399).

Syntax

```
time
```

Examples

To store the current time in integer variable n :

```
n := time;
```

To do something if the time is between 9PM and midnight :

```
if (time >= "9:00PM") and (time <= "11:59:59PM") then ...
```

Note that the following will not work :

```
if (time >= "9:00PM") and (time <= "12:00AM") then ...
```

because the time "12:00AM" is a value of 0, and hence the expression will never be true.

Note that it is actually not even necessary to compare the time with midnight in this case. You could just write :

```
if (time >= "9:00PM") then ...
```

since if the time is after 9PM, it must, by definition, be before midnight.

Note that if you want to check to see if a time is between 9PM and 7AM, the following code will not work :

```
if (time >= "9:00PM") and (time <= "7:00AM") then ...
```

The reason is that a time can not be both after 9PM and before 7AM. The correct logic code is :

```
if (time >= "9:00PM") or (time <= "7:00AM") then ...
```

If you needed to do it the other way around (between 7AM and 9PM), the code would be :

```
if (time >= "7:00AM") and (time <= "9:00PM") then ...
```

If you wanted to perform an action every hour, on the hour, you could write code like this :

```
if (time mod 3600) = 0 then ...
```

or

```
if (Minute = 0) and (Second = 0) then ...
```

4.15 C-Bus Functions

One of the primary purposes of the Logic Engine is to provide control and monitoring of C-Bus.

There are a series of functions provided for access to C-Bus Group Address [levels and states](#) :

- [GetCBusLevel Function](#)
- [GetCBusRampRate Function](#)
- [GetCBusState Function](#)
- [GetCBusTargetLevel Function](#)
- [GetEnableLevel Function](#)
- [GetEnableState Function](#)
- [GetLightingLevel Function](#)
- [GetLightingState Function](#)
- [GetTriggerLevel Function](#)
- [PulseCBusLevel Procedure](#)
- [SetCBusLevel Procedure](#)
- [SetCBusState Procedure](#)
- [SetEnableLevel Procedure](#)
- [SetEnableState Procedure](#)
- [SetLightingLevel Procedure](#)
- [SetLightingState Procedure](#)
- [SetTriggerLevel Procedure](#)
- [TrackGroup Procedure](#)
- [TrackGroup2 Procedure](#)

There are also functions which enable the control and monitoring of C-Bus Scenes. These Scenes are created with the PICED scene editor. The groups in a Scene can be controlled together as a "set" of group addresses.

The functions for C-Bus Scenes are :

- [CrossFadeScene Procedure](#)
- [GetSceneLevel Function](#)
- [GetSceneMaxLevel Function](#)
- [GetSceneMinLevel Function](#)
- [NudgeSceneLevel Procedure](#)
- [SceneIsSet Function](#)
- [SetScene Procedure](#)
- [SetSceneLevel Procedure](#)
- [SetSceneOffset Procedure](#)
- [StoreScene Procedure](#)

Some C-Bus Units such as Temperature Sensors and Light Level Sensors do not broadcast their data onto C-Bus. They must be interrogated to determine their parameters. The PICED software provides Monitor Components which enable the unit parameters to be displayed. If a Monitor Component is monitoring a unit parameter, then the Logic Engine has access to the parameter values. The following functions are used for accessing unit parameters :

- [GetUnitStatus Function](#)

- [GetUnitParameter Function](#)
- [GetUnitParamStatus Function](#)

4.15.1 C-Bus Level and State

C-Bus Group Address levels can be expressed in several ways, as follows.

Levels

On C-Bus, Group Addresses have a level from 0 to 255. To set a lighting Group Address number 4 to a level of 255 (100%), you would use the function :

```
SetLightingLevel(4, 255, 0);
```

Generally you only use the Level of a Group Address if its exact value is important, otherwise, use its state.

Percentage

Levels can also be expressed as a [Percentage](#) (from 0% to 100%). To set a lighting Group Address number 4 to a level of 100%, you would use the function :

```
SetLightingLevel(4, 100%, 0);
```

In the above example, the compiler converts the value 100% to its corresponding level (255).

If you have a variable which contains a level in percent, you must [convert](#) it to a level before being used by a C-Bus Function. For example :

```
SetLightingLevel(4, PercentToLevel(NewLevel), 0);
```

State

The state of a Group Address is a [Boolean](#) value (true/false or on/off). If the level is 0, then the state is false/off. For a level of 1 - 255, the state is true/on.

The State of a Group Address is used where the exact level is not important. For example, if you want to perform an action if a Group Address state is ON, but you are not concerned with its exact level :

```
if GetLightingState(4) then ...
```

or

```
if GetLightingState(4) = ON then ...
```

In the above cases, the condition will be true if the level is anywhere between level 1 and level 255. This condition is equivalent to :

```
if GetLightingLevel(4) > 0 then ...
```

A comparison of the alternatives for expressing a C-Bus level is given below :

Percentage	Level	State
0%	0	False / Off
1%	2	True / On

2%	5	True / On
50%	127	True / On
100%	255	True / On

4.15.2 Tags

C-Bus [Tags](#) are used to make the code easier to read and understand. For example:

```
GetCBusLevel("Local Network", "Lighting", "Kitchen")
```

is easier to read than

```
GetCBusLevel(254, 56, 32)
```

C-Bus Tags give names to the following C-Bus properties :

- Network Names
- Application Names
- Group Address Names
- Level Names

The tags are created with the C-Bus Installation Software, and can be read from the database to avoid the need to use numbers for the above properties when using the [C-Bus Functions](#).

Tags can also be used for :

- Ramp Rates
- Scene Names

Ramp rate tags are of the format :

```
"number s"    for a number of seconds; or
"number m"    for a number of minutes
```

C-Bus only supports ramp rates of 0, 4, 8, 12, 20, 30, 40, 60, 90, 120, 180, 300, 420, 600, 900 and 1020 seconds. If a ramp rate other than these is requested, then the closest one will be used.

Examples :

```
SetLightingLevel("Kitchen", 50%, "4s");
SetScene("All Off");
```

If logic code is intended to be re-usable, it is better to use constants for group addresses. Refer to the [Logic Templates](#) topic for details.

To insert a group address tag in the logic code, select **C-Bus/Insert Group Tag** from the [pop-up menu](#).

See also [C-Bus Tag Functions](#)

4.15.3 CrossFadeScene Procedure

The CrossFadeScene procedure sets the level of a C-Bus Scene Group Addresses to their default levels, but using Ramp Rates selected such that all Groups get to their final level at approximately the selected time.

Syntax

```
CrossFadeScene(SceneNumber, Duration);
```

SceneNumber is an Integer or Scene [Tag](#).
Duration is a time, in seconds.

Description

The Scene Groups are set to their default levels. Each group address is ramped with a rate which depends on the duration and the amount of level change required. Due to the limited choice of [ramp rates](#), the Group Addresses do arrive at their target levels at slightly different times. Note that the index of the first Scene is [0, not 1](#).

Example

To cross fade to Scene "Party" over 8 seconds :

```
CrossFadeScene("Party", 8);
```

4.15.4 GetCBusLevel Function

The GetCBusLevel function returns the [level](#) of a C-Bus Group Address.

Syntax

```
GetCBusLevel(Network, Application, GroupAddress)
```

Network is an [Integer](#) or [Network Tag](#).
Application is an Integer or Application Tag.
GroupAddress is an Integer or Group Address Tag.

Description

The integer result is the [level](#) of the Group Address.

Example

To assign the value of Group Address 32 on Application 56 (lighting) on Network 254 to variable Level :

```
Level := GetCBusLevel(254, 56, 32);
```

To perform an action if the value of Group Address called "Kitchen" on the "Lighting" Application on the "Local Network" is 100% :

```
if GetCBusLevel("Local Network", "Lighting", "Kitchen") = 100% then ...
```

4.15.5 GetCBusRampRate Function

The GetCBusRampRate function returns the ramp rate of a ramping C-Bus Group Address.

Syntax

```
GetCBusRampRate(Network, Application, GroupAddress)
```

Network is an [Integer](#) or [Network Tag](#).
Application is an Integer or Application Tag.
GroupAddress is an Integer or Group Address Tag.

Description

The integer result is the ramp rate of the Group Address in seconds.

Example

To assign the ramp rate of Group Address 32 on Application 56 (lighting) on Network 254 to variable RampRate :

```
RampRate := GetCBusRampRate(254, 56, 32);
```

4.15.6 GetCBusState Function

The GetCBusState function returns the [state](#) of a C-Bus Group Address.

Syntax

```
GetCBusState(Network, Application, GroupAddress)
```

Network is an [Integer](#) or [Network Tag](#).

Application is an Integer or Application Tag.

GroupAddress is an Integer or Group Address Tag.

Description

The boolean result is the state of the Group Address (true/false or on/off) .

Example

To assign the state of Group Address 32 on Application 56 (lighting) on Network 254 to variable State :

```
State := GetCBusLevel(254, 56, 32);
```

To perform an action if Group Address called "Kitchen" on the "Lighting" Application on the "Local Network" is on :

```
if GetCBusState("Local Network", "Lighting", "Kitchen") then ...
```

or

```
if GetCBusState("Local Network", "Lighting", "Kitchen") = ON then ...
```

4.15.7 GetCBusTargetLevel Function

The GetCBusTargetLevel function returns the target (final) [level](#) of a ramping C-Bus Group Address.

Syntax

```
GetCBusTargetLevel(Network, Application, GroupAddress)
```

Network is an [Integer](#) or [Network Tag](#).

Application is an Integer or Application Tag.

GroupAddress is an Integer or Group Address Tag.

Description

The integer result is the [level](#) which the Group Address is ramping towards.

Example

To assign the target value of Group Address 32 on Application 56 (lighting) on Network 254 to variable Level :

```
Level := GetCBusTargetLevel(254, 56, 32);
```

To perform an action if the Group Address called "Kitchen" on the "Lighting" Application on the "Local Network" is ramping :

```
if GetCbusTargetLevel("Local Network", "Lighting", "Kitchen") <>
GetCbusLevel("Local Network", "Lighting", "Kitchen") then ...
```

4.15.8 GetCbusTimer Function

The GetCbusTimer function returns the value of the timer for a C-Bus Group Address.

Syntax

```
GetCbusTimer(Network, Application, GroupAddress)
```

Network is an [Integer](#) or [Network Tag](#).

Application is an Integer or Application Tag.

GroupAddress is an Integer or Group Address Tag.

Description

The integer result is the timer value for the Group Address. This is the time in seconds before the timer "expires". A value of -1 means that the timer has already expired. Note that this will only give the value of the timer running locally. If the timer is in a different unit (for example, a key input switch), it is not possible to know the value of the timer.

Example

To perform an action if the Group Address called "Kitchen" on the "Lighting" Application on the "Local Network" has just timed out :

```
once GetCbusTimer("Local Network", "Lighting", "Kitchen") = -1 then ...
```

4.15.9 GetEnableLevel Function

The GetEnableLevel function returns the [level](#) of a C-Bus Enable Control Group.

Syntax

```
GetEnableLevel(EnableGroup)
```

EnableGroup is an Integer or Group Address [Tag](#).

Description

The integer result is the value (0 to 255) of the Enable Group on the Enable Control Application on the Local Network. Note that the result is the same as using the GetCbusLevel function with the Local Network number and the Enable Control Application address (\$CB).

Example

To assign the value of Enable Group 32 to variable Level :

```
Level := GetEnableLevel(32);
```

To perform an action if the value of the Enable Control Group called "Enable Irrigation" is 100% :

```
if GetEnableLevel("Enable Irrigation") = 100% then ...
```

4.15.10 GetEnableState Function

The GetEnableState function returns the [state](#) of a C-Bus Enable Control Group.

Syntax

```
GetEnableState(EnableGroup)
```

EnableGroup is an Integer or Group Address [Tag](#).

Description

The boolean result is the state of the Enable Group (true/false or on/off) on the Enable Control Application on the Local Network. Note that the result is the same as using the GetCBusState function with the Local Network number and the Enable Control Application address (\$CB).

Example

To assign the state of Enable Group 32 to variable State :

```
State := GetEnableState(32);
```

To perform an action if the Enable Group called "Enable Irrigation" is on :

```
if GetEnableState("Enable Irrigation") then ...
```

or

```
if GetEnableState("Enable Irrigation") = ON then ...
```

4.15.11 GetLightingLevel Function

The GetLightingLevel function returns the [level](#) of a C-Bus Group Address.

Syntax

```
GetLightingLevel(GroupAddress)
```

GroupAddress is an Integer or Group Address [Tag](#).

Description

The integer result is the [level](#) of the Group Address on the Lighting Application on the Local Network. Note that the result is the same as using the GetCBusLevel function with the Local Network number and the default Lighting Application address (\$38).

Example

To assign the value of Lighting Group 32 to variable Level :

```
Level := GetLightingLevel(32);
```

To perform an action if the value of the Lighting Group called "Kitchen" is 100% :

```
if GetLightingLevel("Kitchen") = 100% then ...
```

4.15.12 GetLightingState Function

The GetLightingState function returns the [state](#) of a C-Bus Group Address.

Syntax

```
GetLightingState(GroupAddress)
```

GroupAddress is an Integer or Group Address [Tag](#).

Description

The boolean result is the state of the Group Address (true/false or on/off) on the Lighting Application on the Local Network. Note that the result is the same as using the GetCBusState function with the Local Network number and the default Lighting Application address (\$38).

Example

To assign the state of Lighting Group 32 to variable State :

```
State := GetLightingState(32);
```

To perform an action if the Lighting called "Kitchen" is on :

```
if GetLightingState("Kitchen") then ...
```

or

```
if GetLightingState("Kitchen") = ON then ...
```

4.15.13 GetSceneLevel Function

The GetSceneLevel function returns the level of a C-Bus Scene.

Syntax

```
GetSceneLevel(SceneNumber)
```

SceneNumber is an Integer or Scene [Tag](#).

Description

The integer result is the "value" of the Scene. There are three possible types of result :

Value	Meaning
-2	The Scene group levels are all different and do not match the default scene setting
-1	The Scene group are set to their default level
0 to 255	The Scene groups are all set to this value

Scenes can be used as a collection of Group Addresses which are controlled or monitored together.

Note that the index of the first Scene is [0, not 1](#).

Example

To assign the level of Scene 32 to variable Level:

```
Level := GetSceneLevel(32);
```

To perform an action if the value of the Scene called "Upstairs" is 100%:

```
if GetSceneLevel("Upstairs") = 100% then ...
```

To check to see if a Scene is set:

```
if GetSceneLevel("Away From Home") = -1 then ...
```

Note that the last example checks that all of the Scene Components are at their target level. If the Scene is in the process of being set and Groups are ramping towards their final levels, the value of GetSceneLevel will not be -1. To check if a Scene is set, or in the process of being set, use the [ScenelsSet Function](#).

4.15.14 GetSceneMaxLevel Function

The GetSceneMaxLevel function returns the maximum level of the groups in a C-Bus Scene.

Syntax

```
GetSceneMaxLevel(SceneNumber)
```

SceneNumber is an Integer or Scene [Tag](#).

Description

The integer result is maximum of the levels of the groups in Scene SceneNumber. Scenes can be used as a collection of Group Addresses which are controlled or monitored together. Note that the index of the first Scene is [0, not 1](#).

Example

To determine whether all of the groups in Scene 32 are off :

```
if GetSceneMaxLevel(32) = 0 then ...
```

To determine whether any of the groups in the Scene called "Upstairs" are on :

```
if GetSceneMaxLevel("Upstairs") > 0 then ...
```

4.15.15 GetSceneMinLevel Function

The GetSceneMinLevel function returns the minimum level of the groups in a C-Bus Scene.

Syntax

```
GetSceneMinLevel(SceneNumber)
```

SceneNumber is an Integer or Scene [Tag](#).

Description

The integer result is minimum of the levels of the groups in Scene SceneNumber. Scenes can be used as a collection of Group Addresses which are controlled or monitored together. Note that the index of the first Scene is [0, not 1](#).

Example

To determine whether all of the groups in Scene 32 are on :

```
if GetSceneMinLevel(32) > 0 then ...
```

To determine whether any of the groups in the Scene called "Upstairs" are off :

```
if GetSceneMinLevel("Upstairs") = 0 then ...
```

4.15.16 GetTriggerLevel Function

The GetTriggerLevel function returns the [level](#) (Action Selector) of a C-Bus Trigger Control Group.

Syntax

```
GetTriggerLevel(TriggerGroup)
```

TriggerGroup is an Integer or Group Address [Tag](#).

Description

The integer result is the value of the Trigger Group (0 to 255) on the Trigger Control Application on the Local Network. Note that the result is the same as using the GetCBusLevel function with the Local Network number and the Trigger Control Application address (\$CA).

Note that there is no GetTriggerState function, as the concept of a state is meaningless within the Trigger Control Application.

Example

To assign the value of Trigger Group 32 to variable Level :

```
Level := GetTriggerLevel(32);
```

To perform an action if the value of the Trigger Control Group called "Scenes" is 100% :

```
if GetTriggerLevel("Scenes") = 100% then ...
```

4.15.17 GetUnitStatus Function

The GetUnitStatus function returns whether the C-Bus unit is operating.

Syntax

```
GetUnitStatus(Network, UnitAddress)
```

Network is an [Integer](#) or [Network Tag](#).

UnitAddress is an Integer.

Description

The [Boolean](#) result is whether the unit is operating or not. Note that there needs to be a Monitor monitoring the value of a parameter on the unit for the software to be able to determine if the unit is replying to messages or not.

Example

To determine whether the C-Bus unit with unit address 5 on the Local Network is operating :

```
if GetUnitStatus("Local Network", 5) then ...
```

4.15.18 GetUnitParameter Function

The GetUnitParameter function returns the value of a C-Bus unit parameter.

Syntax

```
GetUnitParameter(Network, UnitAddress, ParameterType)
```

Network is an [Integer](#) or [Network Tag](#).

UnitAddress is an Integer.

ParameterType is an Integer.

Description

The [real](#) result is the value of the selected parameter on the selected unit. The options for the ParameterType are :

Value	Constant	Parameter Returned	Applicable Unit Types
1	ptTemperature	Temperature (°C or °F)	Temperature Sensor
2	ptLightLevel	Light Level (Lux)	Light Level Sensor
3	ptVoltage	Voltage (Volts)	C-Bus 2 Output Units

Notes :

- The unit address must correspond to a unit of the correct type to get meaningful data.
- A Monitor component with the matching unit address and parameter must be used in the Project in order to have the data available.
- The values returned are only as accurate as the unit sensors. Without calibration, the accuracy is not guaranteed.
- The units of temperature (°C or °F) are set in the PICED Project Details.
- The unit parameter [status](#) should be checked before the value is used
- See also the Monitor Value [System IO variable](#)

Example

To assign the light level of unit address 5 on the Local Network to the variable Level :

```
Level := GetUnitParameter("Local Network", 5, ptLightLevel);
```

4.15.19 GetUnitParamStatus Function

The GetUnitParamStatus function returns whether the value returned by the [GetUnitParameter Function](#) is valid.

Syntax

```
GetUnitParamStatus(Network, UnitAddress, ParameterType)
```

Network is an [Integer](#) or [Network Tag](#).

UnitAddress is an Integer.

[ParameterType](#) is an Integer.

Description

The [Boolean](#) result is whether the value of the selected parameter on the selected unit is valid or not. The parameter value will only be valid once the value has been successfully read from the C-Bus unit. Possible reasons for the status being false are :

- Incorrect Network
- Incorrect UnitAddress
- Incorrect ParameterNo
- The C-Bus connection has not yet fully synchronised all units
- There is no PICED Monitor Component monitoring this value

Example

To determine whether the light level of unit address 5 on the Local Network is valid before using it :

```
if GetUnitParamStatus("Local Network", 5, ptLightLevel) then ...
```

4.15.20 NudgeSceneLevel Procedure

The NudgeSceneLevel procedure adjusts the level of a C-Bus Scene Group Addresses by a particular offset.

Syntax

```
NudgeSceneLevel(SceneNumber, Offset, RampRate);
```

SceneNumber is an Integer or Scene [Tag](#).

Offset is an Integer or Percent

RampRate is an integer (number of seconds) or Ramp Rate Tag

Description

The Scene Groups are adjusted by a particular offset at a particular ramp rate. Scenes can be used as a collection of Group Addresses which are controlled or monitored together. Note that the index of the first Scene is [0, not 1](#).

If you have selected **Nudge/Ramp only On Groups in Scenes** in the Project Details, then only the Scene Groups which are already on will be nudged.

Example

To adjust the values of the Group Addresses in Scene 32 by 10% instantaneously :

```
NudgeSceneLevel(32, 10%, 0);
```

To adjust the value of the Group Addresses in Scene called "Upstairs" by -20% over 4 seconds :

```
NudgeSceneLevel("Upstairs", -20%, "4s");
```

4.15.21 PulseCBusLevel Procedure

The PulseCBusLevel procedure pulses the [level](#) of a C-Bus Group Address.

Syntax

```
PulseCBusLevel(Network, Application, GroupAddress, NewLevel, RampRate, Duration, FinalLevel);
```

Network is an [Integer](#) or [Network Tag](#).

Application is an Integer or Application Tag.

GroupAddress is an Integer or Group Address Tag.

NewLevel is an Integer, Percent or Level Tag

RampRate is an integer (number of seconds) or Ramp Rate Tag

Duration is an integer (number of seconds) or Duration Tag

FinalLevel is an Integer, Percent or Level Tag

Description

The Group Address on the selected Application and Network gets set to the NewLevel, with a specified Ramp Rate. After a delay of Duration seconds, the level is set to the FinalLevel. If the level is to return to the original level before the pulse commenced, use a value of -1 as the FinalLevel.

Example

To set the value of Group Address 32 on Application 56 (lighting) on Network 254 to level 255 immediately, and set the level back to 0 following a delay of 10 seconds :

```
PulseCbusLevel(254, 56, 32, 255, 0, 10, 0);
```

To set the value of Group Address called "Kitchen" on the "Lighting" Application on the "Local Network" to 50% over 4 seconds, then set back to the original level after one minute :

```
PulseCbusLevel("Local Network", "Lighting", "Kitchen", 50%, "4s", "0:01:00",
-1);
```

4.15.22 ScenelsSet Function

The ScenelsSet function returns whether a Scene is set.

Syntax

```
SceneIsSet(SceneNumber)
```

SceneNumber is an Integer or Scene [Tag](#).

Description

The ScenelsSet function returns a boolean value corresponding to the state of a Scene indicator. It shows if the Scene Groups are at their target levels, or are ramping towards them.

Example

To switch off a set of lights if they are not already off (or ramping towards off):

```
if not SceneIsSet("All Off") then
begin
SetScene("All Off");
end;
```

See also [GetSceneLevel Function](#)

4.15.23 SetCbusLevel Procedure

The SetCbusLevel procedure sets the [level](#) of a C-Bus Group Address.

Syntax

```
SetCbusLevel(Network, Application, GroupAddress, NewLevel, RampRate);
```

Network is an [Integer](#) or [Network Tag](#).

Application is an Integer or Application Tag.

GroupAddress is an Integer or Group Address Tag.

NewLevel is an Integer, Percent or Level Tag

RampRate is an integer (number of seconds) or [Ramp Rate Tag](#)

Description

The Group Address on the selected Application and Network gets set to the NewLevel, with a specified Ramp Rate. If you select a ramp rate other than the [standard ramp rates](#), it will choose the closest one.

Example

To set the value of Group Address 32 on Application 56 (lighting) on Network 254 to level 255 immediately :

```
SetCbusLevel(254, 56, 32, 255, 0);
```

To set the value of Group Address called "Kitchen" on the "Lighting" Application on the "Local Network" to 50% over 4 seconds :

```
SetCbusLevel("Local Network", "Lighting", "Kitchen", 50%, "4s");
```

4.15.24 SetCbusState Procedure

The SetCbusState procedure sets the [state](#) of a C-Bus Group Address.

Syntax

```
SetCbusState(Network, Application, GroupAddress, NewState);
```

Network is an [Integer](#) or [Network Tag](#).

Application is an Integer or Application Tag.

GroupAddress is an Integer or Group Address Tag.

NewState is a Boolean value

Description

The Group Address on the selected Application and Network gets set to the NewState immediately (zero ramp rate).

Example

To set the value of Group Address 32 on Application 56 (lighting) on Network 254 to level 255 immediately :

```
SetCbusState(254, 56, 32, ON);
```

To set the value of Group Address called "Kitchen" on the "Lighting" Application on the "Local Network" to 0% immediately :

```
SetCbusState("Local Network", "Lighting", "Kitchen", OFF)
```

4.15.25 SetEnableLevel Procedure

The SetEnableLevel procedure sets the [level](#) of a C-Bus Enable Group.

Syntax

```
SetEnableLevel(EnableGroup, NewLevel);
```

EnableGroup is an Integer or Group Address [Tag](#).

NewLevel is an Integer, Percent or Level Tag

Description

The Enable Group on the Local Network gets set to the NewLevel, instantaneously. Note that the result is the same as using the SetCbusLevel function with the Local Network number, the Enable Control Application address (\$CB) and zero ramp rate.

Example

To set the value of Enable Group 32 to level 255 :

```
SetEnableLevel(32, 255);
```

To set the value of the Enable Group called "Enable Irrigation" to level 255 :

```
SetEnableLevel("Enable Irrigation", 100%);
```

4.15.26 SetEnableState Procedure

The SetEnableState procedure sets the [state](#) of a C-Bus Enable Group.

Syntax

```
SetEnableState(EnableGroup, NewState);
```

EnableGroup is an Integer or Group Address [Tag](#).

NewState is a [Boolean](#) value

Description

The Enable Group on the Local Network gets set to the NewState, instantaneously. Note that the result is the same as using the SetCBusState function with the Local Network number, the Enable Control Application address (\$CB).

Example

To set the value of Enable Group 32 to on (level 255) :

```
SetEnableState(32, ON);
```

To set the value of the Enable Group called "Enable Irrigation" to on (level 255) :

```
SetEnableState("Enable Irrigation", ON);
```

4.15.27 SetLightingLevel Procedure

The SetLightingLevel procedure sets the [level](#) of a C-Bus Lighting Group Address.

Syntax

```
SetLightingLevel(GroupAddress, NewLevel, RampRate);
```

GroupAddress is an Integer or Group Address [Tag](#).

NewLevel is an Integer, Percent or Level Tag

RampRate is an integer (number of seconds) or [Ramp Rate Tag](#)

Description

The Group Address on the Local Network gets set to the NewLevel, instantaneously. Note that the result is the same as using the SetCBusLevel function with the Local Network number and the Lighting Application address (\$38).

Example

To set the value of Group Address 32 to level 255 instantaneously :

```
SetLightingLevel(32, 255, 0);
```

To set the value of the Group Address called "Kitchen" to level 255 over 4 seconds :

```
SetLightingLevel("Kitchen", 100%, "4s");
```

4.15.28 SetLightingState Procedure

The SetLightingState procedure sets the [state](#) of a C-Bus Lighting Group Address.

Syntax

```
SetLightingState(GroupAddress, NewState);
```

GroupAddress is an Integer or Group Address [Tag](#).

NewState is a [Boolean](#) value

Description

The Group Address on the Local Network gets set to the NewState, instantaneously. Note that the result is the same as using the SetCBusState function with the Local Network number and the Lighting Application address (\$38).

Example

To set the value of Group Address 32 to on (level 255) :

```
SetLightingState(32, ON);
```

To sets the value of the Group Address called "Kitchen" to on (level 255) :

```
SetLightingState("Kitchen", ON);
```

4.15.29 SetScene Procedure

The SetScene procedure sets the level of a C-Bus Scene Group Addresses to their default levels.

Syntax

```
SetScene(SceneNumber);
```

SceneNumber is an Integer or Scene [Tag](#).

Description

The Scene Groups are set to their default levels with their default ramp rate. Note that the index of the first Scene is [0, not 1](#).

Example

To set the value of the Group Addresses in Scene 32 to their default levels :

```
SetScene(32);
```

To set the value of the Group Addresses in Scene called "Upstairs" to their default levels :

```
SetScene("Upstairs");
```

See also [CrossFadeScene Procedure](#)

4.15.30 SetSceneLevel Procedure

The SetSceneLevel procedure sets the level of a C-Bus Scene Group Addresses to particular levels.

Syntax

```
SetSceneLevel(SceneNumber, NewLevel, RampRate);
```

SceneNumber is an Integer or Scene [Tag](#).

NewLevel is an Integer or Percent

RampRate is an integer (number of seconds) or Ramp Rate Tag

Description

The Scene Groups are set to particular levels with a particular ramp rate. If you select a ramp rate other than the [standard ramp rates](#), it will choose the closest one.

There are two alternatives for the New Level :

New Level	Meaning
-1	The Scene groups are set to their default level
0 to 255	The Scene groups are all set to this value

Scenes can be used as a collection of Group Addresses which are controlled or monitored together.

Note that the index of the first Scene is [0, not 1](#).

Example

To set the value of the Group Addresses in Scene 32 to 255 instantaneously :

```
SetSceneLevel(32, 255, 0);
```

To set the value of the Group Addresses in Scene called "Upstairs" to 50% over 4 seconds :

```
SetSceneLevel("Upstairs", 50%, "4s");
```

4.15.31 SetSceneOffset Procedure

The SetSceneOffset procedure sets the level of a C-Bus Scene Group Addresses to their default level plus an offset.

Syntax

```
SetSceneOffset(SceneNumber, Offset, RampRate);
```

SceneNumber is an Integer or Scene [Tag](#).

Offset is an Integer or Percent (-100% to +100%)

RampRate is an integer (number of seconds) or Ramp Rate Tag

Description

The Scene Groups are set to particular levels with a particular ramp rate. If you select a ramp rate other than the [standard ramp rates](#), it will choose the closest one. The level for each Group Address is its default value plus the offset value. Note that the index of the first Scene is [0, not 1](#).

If you have selected **Nudge/Ramp only On Groups in Scenes** in the Project Details, then only the Scene Groups which are already on will be adjusted.

Example

To set the value of the Group Addresses in Scene 32 to 10% higher than the default level instantaneously :

```
SetSceneOffset(32, 10%, 0);
```

To set the value of the Group Addresses in Scene called "Upstairs" to 10% lower than the default over 4 seconds :

```
SetSceneOffset("Upstairs", -10%, "4s");
```

4.15.32 SetTriggerLevel Procedure

The SetTriggerLevel procedure sets the [level](#) (Action Selector) of a C-Bus Trigger Group.

Syntax

```
SetTriggerLevel(TriggerGroup, NewLevel);
```

TriggerGroup is an Integer or Group Address [Tag](#).

NewLevel is an Integer, Percent or Level Tag

Description

The Trigger Group on the Local Network gets set to the NewLevel, instantaneously. Note that the result is the same as using the SetCBusLevel function with the Local Network number, the Trigger Control Application address (\$CA) and zero ramp rate.

Note that there is no SetTriggerState function, as the concept of a state is meaningless within the Trigger Control Application.

Example

To set the value of Trigger Group 32 to level 255 :

```
SetTriggerLevel(32, 255);
```

To set the value of the Trigger Group called "Scenes" to level 255 :

```
SetTriggerLevel("Scenes", 100%);
```

4.15.33 StoreScene Procedure

The StoreScene procedure stores the level of a C-Bus Scene Group Addresses.

Syntax

```
StoreScene(SceneNumber);
```

SceneNumber is an Integer or Scene [Tag](#).

Description

The current levels of the Scene Group Addresses are stored in the Scene. Note that the index of the first Scene is [0, not 1](#).

Example

To store the value of the Group Addresses in Scene 32 at their current levels :

```
StoreScene(32);
```

To store the value of the Group Addresses in Scene called "Upstairs" at their current levels :

```
StoreScene("Upstairs");
```

4.15.34 TrackGroup Procedure

The TrackGroup procedure sets the [level](#) of a C-Bus Group Address to match another Group Address.

Syntax

```
TrackGroup(Network, Application, GroupAddress1, GroupAddress2);
```

Network is an [Integer](#) or [Network Tag](#).

Application is an Integer or Application Tag.

GroupAddress1 and GroupAddress2 are Integers or Group Address Tags.

Description

The TrackGroup procedure makes GroupAddress2 on the selected Application and Network get set to be the same level (and ramp rate if ramping) as GroupAddress1.

Note that if GroupAddress2 gets changed, the TrackGroup procedure will change it back to match GroupAddress1.

See also [TrackGroup2 Procedure](#)

Example

See the FAQ topic [Tracking a Group Address](#) for examples.

4.15.35 TrackGroup2 Procedure

The TrackGroup2 procedure sets the [level](#) of two C-Bus Group Addresses to match each other.

Applicability

Colour C-Touch only.

Syntax

```
TrackGroup2(Network1, Application1, GroupAddress1, Network2, Application2,  
GroupAddress2);
```

Network1 and Network 2 are [Integers](#) or [Network Tags](#).

Application1 and Application 2 are Integers or Application Tags.

GroupAddress1 and GroupAddress2 are Integers or Group Address Tags.

Description

The TrackGroup2 procedure makes Group Addresses 1 and 2 track each other. If Group Address 1 (on Network 1, Application 1) changes, then Group Address 2 (on Network 2, Application 2) will be changed to match and vice versa..

See also [TrackGroup Procedure](#)

Example

See the FAQ topic [Tracking a Group Address](#) for examples.

4.15.36 C-Bus Tag Functions



[C-Bus Tags](#) are generally used in [C-Bus Functions](#) to make the code easier to read. The tag is interpreted as a number by the logic engine.

In rare circumstances, it is useful to know or use the actual text of a tag. The following functions can be used with [C-Bus Tags](#):

- [GetCBusNetworkCount Function](#)
- [GetCBusNetworkFromIndex Function](#)
- [GetCBusNetworkAddress Function](#)
- [GetCBusNetworkTag Procedure](#)
- [GetCBusApplicationCount Function](#)
- [GetCBusApplicationFromIndex Function](#)
- [GetCBusApplicationAddress Function](#)
- [GetCBusApplicationTag Procedure](#)
- [GetCBusGroupCount Function](#)
- [GetCBusGroupFromIndex Function](#)
- [GetCBusGroupAddress Function](#)
- [GetCBusGroupTag Procedure](#)
- [GetCBusLevelCount Function](#)
- [GetCBusLevelFromIndex Function](#)
- [GetCBusLevelAddress Function](#)
- [GetCBusLevelTag Procedure](#)

⚠ If you are considering using these functions, be sure that they are really needed and that there is not an alternative method you can use. These use considerable processor time and should be used sparingly. It is recommended that they only be used in the [Initialisation](#) section if possible.

4.15.36.1 GetCBusNetworkCount Function

The GetCBusNetworkCount function returns the number of C-Bus Networks.

Applicability

Colour C-Touch only.

Syntax

```
GetCBusNetworkCount
```

Description

This function returns the number of C-Bus networks in the project.

Example

See [GetCBusNetworkTag Example](#)

4.15.36.2 GetCBusNetworkFromIndex Function

The GetCBusNetworkFromIndex function returns the number/address of a C-Bus Network.

Applicability

Colour C-Touch only.

Syntax

```
GetCBusNetworkFromIndex(index)
```

Where index is an integer from 0 to the number of networks - 1 (see [GetCBusNetworkCount](#)).

Description

This function returns the number/address of network from its index in the list of C-Bus Networks. Note that the list of C-Bus Networks is not sorted.

Example

See [GetCBusNetworkTag Example](#)

4.15.36.3 GetCBusNetworkAddress Function

The GetCBusNetworkAddress function returns the number/address of a C-Bus Network from its name ([Tag](#)).

Syntax

```
GetCBusNetworkAddress(Name)
```

Where Name is a [String](#) variable or a C-Bus Network [Tag](#)

Description

This function returns the number/address of network from its name ([Tag](#)). If the tag name does not exist, the function result will be -1.

If the Name parameter is a string, then the name is looked up in the C-Bus Tag Database each time the function is executed. This is relatively demanding on processor time. **This is only possible in Colour C-Touch.**

If the Name parameter is a Network Tag, then the name is looked up in the C-Bus Tag Database at compile time only. This is not at all demanding on processor time.

Example

To get the number of a network called "Local" (at compile time) and store it in variable n:

```
n := GetCBusNetworkAddress("Local");
```

To get the number of a network which is stored in a variable NetName and store it in variable n:

```
n := GetCBusNetworkAddress(NetName);
```

4.15.36.4 GetCbusNetworkTag Procedure

The GetCbusNetworkTag procedure returns the name ([Tag](#)) of a C-Bus Network.

Applicability

Colour C-Touch only.

Syntax

```
GetCbusNetworkTag(Address, Name);
```

Where:

Address is an [integer](#); the Network address/number.

Name is a [String](#) variable

Description

This procedure gets the name (Tag) of a network from its address/number and stores it in the Name variable.

Example

To display a list of all networks in the project:

```
Count := GetCbusNetworkCount;
WriteLn(Count, ' Networks');
for i := 1 to Count do
begin
    Net := GetCbusNetworkFromIndex(i);
    GetCbusNetworkTag(Net, NetName);
    WriteLn(i, Net, ' ', NetName);
end;
```

4.15.36.5 GetCbusApplicationCount Function

The GetCbusApplicationCount function returns the number of C-Bus Applications.

Applicability

Colour C-Touch only.

Syntax

```
GetCbusApplicationCount
```

Description

This function returns the number of C-Bus Applications in the project.

Example

See [GetCbusApplicationTag Example](#)

4.15.36.6 GetCbusApplicationFromIndex Function

The GetCbusApplicationFromIndex function returns the number/address of a C-Bus Application.

Applicability

Colour C-Touch only.

Syntax

```
GetCbusApplicationFromIndex(index)
```

Where index is an integer from 0 to the number of Applications - 1 (see [GetCbusApplicationCount Function](#)).

Description

This function returns the number/address of Application from its index in the list of C-Bus Applications. Note that the list of C-Bus Applications is not sorted.

Example

See [GetCbusApplicationTag Example](#)

4.15.36.7 GetCbusApplicationAddress Function

The GetCbusApplicationAddress function returns the number/address of a C-Bus Application from its name ([Tag](#)).

Syntax

```
GetCbusApplicationAddress(Name)
```

Where Name is a [String](#) variable or a C-Bus Application [Tag](#)

Description

This function returns the number/address of Application from its name ([Tag](#)). If the tag name does not exist, the function result will be -1.

If the Name parameter is a string, then the name is looked up in the C-Bus Tag Database each time the function is executed. This is relatively demanding on processor time. **This is only possible in Colour C-Touch.**

If the Name parameter is an Application Tag, then the name is looked up in the C-Bus Tag Database at compile time only. This is not at all demanding on processor time.

Example

To get the number of a Application called "Lighting" (at compile time) and store it in variable n:

```
n := GetCbusApplicationAddress("Lighting");
```

To get the number of a Application which is stored in a variable AppName and store it in variable n:

```
n := GetCbusApplicationAddress(AppName);
```

4.15.36.8 GetCbusApplicationTag Procedure

The GetCbusApplicationTag procedure returns the name ([Tag](#)) of a C-Bus Application.

Applicability

Colour C-Touch only.

Syntax

```
GetCbusApplicationTag(Address, Name);
```

Where:

Address is an [integer](#); the Application address/number.

Name is a [String](#) variable

Description

This procedure gets the name (Tag) of a Application from its address/number and stores it in the Name variable.

Example

To display a list of all Applications in the project:

```
Count := GetCbusApplicationCount;
WriteLn(Count, ' Applications');
for i := 1 to Count do
begin
  App := GetCbusApplicationFromIndex(i);
  GetCbusApplicationTag(App, AppName);
  WriteLn(i, App, ' ', AppName);
end;
```

4.15.36.9 GetCbusGroupCount Function

The GetCbusGroupCount function returns the number of C-Bus Groups.

Applicability

Colour C-Touch only.

Syntax

```
GetCbusGroupCount(Network, Application)
```

Where

Network is an [Integer](#) or [Network Tag](#).

Application is an Integer or Application Tag.

Description

This function returns the number of C-Bus Groups in the Network and Application.

Example

See [GetCBusGroupTag Example](#)

4.15.36.1 GetCBusGroupFromIndex Function

The GetCBusGroupFromIndex function returns the number/address of a C-Bus Group.

Applicability

Colour C-Touch only.

Syntax

```
GetCBusGroupFromIndex(Network, Application, index)
```

Where

Network is an [Integer](#) or [Network Tag](#).

Application is an Integer or Application Tag.

index is an integer from 0 to the number of Groups - 1 (see [GetCBusGroupCount Function](#)).

Description

This function returns the number/address of Group from its index in the list of C-Bus Groups. Note that the list of C-Bus Groups is not sorted.

Example

See [GetCBusGroupTag Example](#)

4.15.36.11 GetCBusGroupAddress Function

The GetCBusGroupAddress function returns the number/address of a C-Bus Group from its name ([Tag](#)).

Syntax

```
GetCBusGroupAddress(Network, Application, Name)
```

Where

Network is an [Integer](#) or [Network Tag](#).

Application is an Integer or Application Tag.

Name is a [String](#) variable or a Group Address [Tag](#)

Description

This function returns the number/address of a Group from its name ([Tag](#)). If the tag name does not exist, the function result will be -1.

If the Name parameter is a string, then the name is looked up in the C-Bus Tag Database each time the function is executed. This is relatively demanding on processor time. **This is only possible in Colour C-Touch.**

If the Name parameter is a Group Address Tag, then the name is looked up in the C-Bus Tag Database at compile time only. This is not at all demanding on processor time.

Example

To get the number of a Group called "Kitchen" (at compile time) which is on the "Local" network and "Lighting" application, and store it in variable n:

```
n := GetCbusGroupAddress("Local", "Lighting", "Kitchen");
```

To get the number of a Group which is stored in a variable GroupName which is on the "Local" network and "Lighting" application, and store it in variable n:

```
n := GetCbusGroupAddress("Local", "Lighting", GroupName);
```

4.15.36.1: GetCbusGroupTag Procedure

The GetCbusGroupTag procedure returns the name ([Tag](#)) of a C-Bus Group.

Applicability

Colour C-Touch only.

Syntax

```
GetCbusGroupTag(Network, Application, Address, Name);
```

Where:

Network is an [Integer](#) or [Network Tag](#).

Application is an Integer or Application Tag.

Address is an [integer](#); the Group address/number.

Name is a [String](#) variable

Description

This procedure gets the name (Tag) of a Group from its address/number and stores it in the Name variable.

Example

To display a list of all Groups in the "Local" network and "Lighting" application:

```
Count := GetCbusGroupCount("Local", "Lighting");
WriteLn(Count, ' Lighting Groups');
for i := 1 to Count do
begin
    Group := GetCbusGroupFromIndex("Local", "Lighting", i);
    GetCbusGroupTag("Local", "Lighting", Group, GroupName);
    WriteLn(i, Group, ' ', GroupName);
end;
```

4.15.36.1: GetCbusLevelCount Function

The GetCbusLevelCount function returns the number of C-Bus Levels.

Applicability

Colour C-Touch only.

Syntax

```
GetCbusGroupCount(Network, Application, GroupAddress)
```

Where

Network is an [Integer](#) or [Network Tag](#).

Application is an Integer or Application Tag.

GroupAddress is an Integer or Group Address Tag.

Description

This function returns the number of C-Bus Levels in the Group Address.

Example

See [GetCbusLevelTag Example](#)

4.15.36.14 GetCbusLevelFromIndex Function

The GetCbusLevelFromIndex function returns the number/address of a C-Bus Level.

Applicability

Colour C-Touch only.

Syntax

```
GetCbusLevelFromIndex(Network, Application, GroupAddress, index)
```

Where

Network is an [Integer](#) or [Network Tag](#).

Application is an Integer or Application Tag.

GroupAddress is an Integer or Group Address Tag.

index is an integer from 0 to the number of Levels - 1 (see [GetCbusLevelCount Function](#)).

Description

This function returns the number/address of Level from its index in the list of C-Bus Levels. Note that the list of C-Bus Levels is not sorted.

Example

See [GetCbusLevelTag Example](#)

4.15.36.15 GetCbusLevelAddress Function

The GetCbusLevelAddress function returns the number/address of a C-Bus Level from its name ([Tag](#)).

Syntax

```
GetCbusLevelAddress(Network, Application, GroupAddress, Name)
```

Where

Network is an [Integer](#) or [Network Tag](#).

Application is an Integer or Application Tag.

GroupAddress is an Integer or Group Address Tag.

Name is a [String](#) variable or a Level [Tag](#)

Description

This function returns the number/address of a Level from its name ([Tag](#)). If the tag name does not exist, the function result will be -1.

If the Name parameter is a string, then the name is looked up in the C-Bus Tag Database each time the function is executed. This is relatively demanding on processor time. **This is only possible in Colour C-Touch.**

If the Name parameter is a Level Tag, then the name is looked up in the C-Bus Tag Database at compile time only. This is not at all demanding on processor time.

Example

To get the number of a Level called "Preset" (at compile time) which is on the "Local" network, "Lighting" application, "Kitchen" group, and store it in variable n:

```
n := GetCbusLevelAddress("Local", "Lighting", "Kitchen", "Preset");
```

To get the number of a Level which is stored in a variable LevelName which is on the "Local" network, "Lighting" application, "Kitchen" group, and store it in variable n:

```
n := GetCbusLevelAddress("Local", "Lighting", "Kitchen", LevelName);
```

4.15.36.1 GetCbusLevelTag Procedure

The GetCbusLevelTag procedure returns the name ([Tag](#)) of a C-Bus Level.

Applicability

Colour C-Touch only.

Syntax

```
GetCbusLevelTag(Network, Application, GroupAddress, Address, Name);
```

Where:

Network is an [Integer](#) or [Network Tag](#).

Application is an Integer or Application Tag.

GroupAddress is an Integer or Group Address Tag.

Address is an [integer](#); the Level address/number.

Name is a [String](#) variable

Description

This procedure gets the name (Tag) of a Level from its address/number and stores it in the Name variable.

Example

To display a list of all Levels (Action Selectors) in the "Local" network, "Trigger" application, "Scenes" group:

```
Count := GetCbusLevelCount("Local", "Trigger Control", "Scenes");
WriteLn(Count, ' Levels in Scenes');
for i := 1 to Count do
```

```
begin
  Level := GetCBusLevelFromIndex("Local", "Trigger Control", "Scenes", i -
1);
  GetCBusLevelTag("Local", "Trigger Control", "Scenes", Level, LevelName);
  WriteLn(i, Level, ' ', LevelName);
end;
```

4.15.37 Tutorial 4

Question 1

Write a [Once Statement](#) to set C-Bus Lighting Group Address "Porch Light" on at sunset + 1/2 hour every week night (Monday to Friday).

Question 2

Write a Once statement to set the scene "Party" at 7PM on the first Friday of each month.

Question 3

Write a statement to assign the current time plus two hours to a variable called "OffTime".

Question 4

Write a statement to increment (ie. add one to) a variable called Counter every 20 seconds. Do the same for every 45 seconds (this has less alternative solutions).

Question 5

Write a statement to nudge a Scene called "Living Area" up by 10% when a Trigger Group called "Nudge Up" is set to 100%.

[Tutorial Answers](#)

4.16 Timer Functions

The Logic Engine has timers that can be used for determining the amount of time that has elapsed since some event. Each timer is either disabled or running. The timers which are running have their value incremented automatically every second.

There are several functions which can be used with timers :

- [TimerRunning Function](#)
- [TimerSet Procedure](#)
- [TimerStart Procedure](#)
- [TimerStop Procedure](#)
- [TimerTime Function](#)

Notes

The value of a Logic Timer can be displayed with a Clock component.

Counters count upwards (not down).

If a timer value is -1, this means that the timer is not running. Once it gets set to 0, it will start counting upwards automatically.

For the PAC, the maximum timer value is 9 hours (32767 seconds). If a timer is allowed to count beyond this time, it will roll over to -9 hours then continue counting upwards.

See also [Time Functions](#) and Software Limits.

4.16.1 TimerRunning Function

The TimerRunning function returns whether the specified Logic Timer is running or not.

Syntax

```
TimerRunning(n)
```

Description

The boolean result is whether timer number n is running or not.

Example

To perform an action if Timer 2 is running :

```
if TimerRunning(2) then ...
```

4.16.2 TimerSet Procedure

The TimerSet procedure sets a Logic Timer to a particular value.

Syntax

```
TimerSet(n, t)
```

Description

This sets the value of timer number n to a value of t. If the timer is not already running, this will start the timer.

Example

To set Timer 2 to 10 seconds :

```
TimerSet(2, 10);
```

To set Timer 2 to 0 seconds - this is the same as TimerStart(2) :

```
TimerSet(2, 0);
```

To set Timer n to -1 seconds - this is the same as TimerStop(n) :

```
TimerSet(n, -1);
```

4.16.3 TimerStart Procedure

The TimerStart procedure starts a Logic Timer.

Syntax

```
TimerStart(n);
```

Description

This starts timer number n running. The time will be set to 0. If the timer is already running, it will

just set the time to 0.

Example

To start Timer 2 :

```
TimerStart(2);
```

4.16.4 TimerStop Procedure

The TimerStop procedure stops a Logic Timer.

Syntax

```
TimerStop(n);
```

Description

This stops timer number n running. The time will be set to -1.

Example

To stop Timer 2 :

```
TimerStop(2);
```

4.16.5 TimerTime Function

The TimerTime function returns the time of a specified Logic Timer.

Syntax

```
TimerTime(n)
```

Description

The integer result is the value of Logic Timer number n. If the result is -1, then the timer is not running. The Timer Time becomes 1 greater each second while the Timer is running.

Example

To perform an action if the value of Timer 2 is 60 seconds :

```
if TimerTime(2) = 60 then ...
```

To display the value of timer 3 on the screen :

```
TextPos(100, 100);
DrawText('Time = ', TimerTime(3));
```

To display the value of timer 3 on the screen, as a value counting down from 60 seconds :

```
TextPos(100, 100);
DrawText('Time = ', 60 - TimerTime(3));
```

4.17 System IO Functions

System Input/Output (IO) Variables are provided for additional system control and interaction with the logic engine. There are two types of System IO Variables :

- [In-Built System IO Variables](#) : these are pre-defined and provide access to various system functions
- [User System IO Variables](#) : these are defined by the user as required

User System IO variables can be [Integer](#), [Real](#), [Boolean](#) or [Strings](#).

4.17.1 Using SystemIO Variables with Components

A PICED component (such as a button or slider) can control or monitor the state of a System IO variable. To get a Component to control or monitor a System IO variable :

- Place a Component on a Page
- Edit the Component
- Click on the **System IO** tab
- Select the **Key Function**
- Select whether a User or In-Built System IO Variable is required
- Select the System IO **Variable** from the list
- Select the **Value** if applicable

To copy the User System IO Variable from one Component to another :

- Select the component to have its System IO Variable copied
- Select the other components to have the System IO Variable copied to
- Select the **Edit | Copy Group** menu item

4.17.2 User System IO Variables

User System IO (Input / Output) Variables are used to provide a means of user input and output to the Logic Engine. Components can be placed on the PICED page to allow the user to set or read the User System IO variable value.

The values of User System IO variables are saved when the project is saved. This provides a means of achieving non-volatile storage for the Logic Engine. The present value of the User System IO variables can be seen and set with the [System IO Editor](#). These values also get changed by Components and logic functions.

The [System IO Editor](#) can be used to create and edit User System IO variables.

The following functions can be used with User System IO variables :

- [GetBoolSystemIO Function](#)
- [GetIntSystemIO Function](#)
- [GetRealSystemIO Function](#)
- [GetStringSystemIO Procedure](#)
- [SetBoolSystemIO Procedure](#)
- [SetIntSystemIO Procedure](#)
- [SetRealSystemIO Procedure](#)
- [SetStringSystemIO Procedure](#)

Note : the differences between a User System IO Variable and a regular Logic [Variable](#) are :

- the user can not directly monitor or control a regular variable.
- with User System IO variables, the "get" and "set" functions must be used to access them
- the values of User System IO variables are not initialised when the Logic Engine runs (they stay at their previous value). The values can be reset in the logic [Initialisation](#) section if required.
- the values of User System IO variables are saved when the Project is saved

A User System IO Variable should not be used where a regular Logic variable will suffice. There are several problems with using a System IO variable instead of a regular variable :

- User System IO variables slow down the logic execution
- User System IO variables use more resources
- User System IO variables make larger project files
- There are a limited number of User System IO variables available

A User System IO Variable should be used when :

- the value needs to be restored following a power failure
- the user needs to be able to control the value easily

See also Software Limits.

4.17.2.1 System IO Manager

The System IO Manager is used to :

- Add new [System IO Variables](#)
- Edit System IO Variables
- Change System IO Variable values

To open the System IO Manager, click on the **System IO** button on the Logic Editor [Tool Bar](#), or on the PICED tool bar.

To Add a new System IO variable, click on the **Add** button. The [System IO Variable Editor](#) will open.

To delete an existing System IO variable, select the System IO variable and click on the **Delete** button.

To edit an existing System IO variable, select the System IO variable and click on the **Edit** button. The [System IO Variable Editor](#) will open.

To make a copy of an existing System IO variable, select the System IO variable and click on the **Duplicate** button.

Note : Special Functions can also be used to allow the user to open the System IO Manager or to open a particular System IO Variable.

See also Software Limits.

4.17.2.1.1 System IO Variable Editor

The System IO Variable Editor can be accessed from the [System IO Manager](#) or by using Special Functions.

The System IO Variable Editor allows the properties of a System IO Variable to be set :

- **Name** : this is the name of the System IO Variable
- **Type** : this is the [Type](#) of the System IO Variable (note that dates and times are just [Integers](#))
- **Minimum** : this is the minimum value that the System IO Variable can have (not applicable for [Boolean Types](#) or [String Types](#))
- **Maximum** : this is the maximum value that the System IO Variable can have (not applicable for [Boolean Types](#) or [String Types](#))
- **Value** : this is the current value of the System IO Variable

4.17.2.2 System IO Tags

A System IO Tag is a [Tag](#) used to refer to a System IO Variable number. The tag corresponds to the System IO Variable [name](#).

The format is :

"System IO Variable Name"

4.17.2.3 GetBoolSystemIO Function

The GetBoolSystemIO function returns the value of a [Boolean System IO](#) variable.

Syntax

```
GetBoolSystemIO(n)
```

n is an [Integer](#) or System IO Variable [System IO Tag](#).

Description

The boolean result is the value of System IO variable number n. If System IO variable number n is not of the correct type, a [Compilation Error](#) or [Run Time Error](#) will occur. Note that the index of the first System IO Variable is [0, not 1](#).

Example

To assign the value of System IO variable number 2 to a variable called State :

```
State := GetBoolSystemIO(2);
```

To perform an action if the value of System IO variable called "Enable State" is true :

```
if GetBoolSystemIO("Enable State") then ...
```

4.17.2.4 GetIntSystemIO Function

The GetIntSystemIO function returns the value of an [integer System IO](#) variable.

Syntax

```
GetIntSystemIO(n)
```

n is an [Integer](#) or System IO Variable [System IO Tag](#).

Description

The integer result is the value of System IO variable number n. This can also be used to get the value of a [Date](#) or [Time](#) System IO variable, since dates and times are just integers. If System IO variable number n is not of the correct type, a [Compilation Error](#) or [Run Time Error](#) will occur. Note that the index of the first System IO Variable is [0, not 1](#).

Example

To assign the value of System IO variable number 2 to a variable called Date1 :

```
Date1 := GetIntSystemIO(2);
```

To perform an action if the value of System IO variable called "Start Time" is equal to the current time :

```
if GetIntSystemIO("Start Time") = Time then ...
```

4.17.2.5 GetRealSystemIO Function

The GetRealSystemIO function returns the value of a [Real System IO](#) variable.

Syntax

```
GetRealSystemIO(n)
```

n is an [Integer](#) or System IO Variable [System IO Tag](#).

Description

The real result is the value of System IO variable number n. If System IO variable number n is not of the correct type, a [Compilation Error](#) or [Run Time Error](#) will occur. Note that the index of the first System IO Variable is [0, not 1](#).

Example

To assign the value of System IO variable number 2 to a variable called x :

```
x := GetRealSystemIO(2);
```

To perform an action if the value of System IO variable called "Setting" is equal to a variable called x :

```
if GetRealSystemIO("Setting") = x then ...
```

4.17.2.6 GetStringSystemIO Procedure

The GetStringSystemIO procedure obtains the value of an [String System IO](#) variable.

Syntax

```
GetStringSystemIO(n, v);
```

n is an [Integer](#) or System IO Variable [System IO Tag](#).
v is a string variable

Description

The value of System IO variable number n is stored in v. If System IO variable number n is not of the correct type, a [Compilation Error](#) or [Run Time Error](#) will occur. Note that the index of the first System IO Variable is [0, not 1](#).

Example

To store the value of System IO variable number 2 in variable s :

```
GetStringSystemIO(2, s);
```

To store the value of System IO variable called "Alarm State" in variable State :

```
GetStringSystemIO("Alarm State", State);
```

To compare a system IO variable called "My String" with the text 'kitchen' (using string variable called s) :

```
GetStringSystemIO("My String", s);  
if s = 'kitchen' then ...
```

4.17.2.7 SetBoolSystemIO Procedure

The SetBoolSystemIO procedure sets the value of an [boolean System IO](#) variable.

Syntax

```
SetBoolSystemIO(n, v);
```

n is an [Integer](#) or System IO Variable [System IO Tag](#).
v is a boolean expression

Description

The value of System IO variable number n is set to v. If System IO variable number n is not of the correct type, a [Compilation Error](#) or [Run Time Error](#) will occur. Note that the index of the first System IO Variable is [0, not 1](#).

Example

To set the value of System IO variable number 2 to TRUE :

```
SetBoolSystemIO(2, true);
```

To set the value of System IO variable called "Enable State" to ON (TRUE) :

```
SetBoolSystemIO("Enable State", ON);
```

4.17.2.8 SetIntSystemIO Procedure

The SetIntSystemIO procedure sets the value of an [integer System IO](#) variable.

Syntax

```
SetIntSystemIO(n, v);
```

n is an [Integer](#) or System IO Variable [System IO Tag](#).
v is an integer expression

Description

The value of System IO variable number n is set to v. This can also be used to set the value of a [Date](#) or [Time](#) System IO variable, since dates and times are just integers. If System IO variable number n is not of the correct type, a [Compilation Error](#) or [Run Time Error](#) will occur. Note that the index of the first System IO Variable is [0, not 1](#).

Example

To set the value of System IO variable number 2 to 5 :

```
SetIntSystemIO(2, 5);
```

To set the value of System IO variable called "Start Time" to 46800 (1PM) :

```
SetIntSystemIO("Start Time", "1:00 PM");
```

4.17.2.9 SetRealSystemIO Procedure

The SetRealSystemIO procedure sets the value of an [real System IO](#) variable.

Syntax

```
SetRealSystemIO(n, v);
```

n is an [Integer](#) or System IO Variable [System IO Tag](#).
v is a real expression

Description

The value of System IO variable number n is set to v. If System IO variable number n is not of the correct type, a [Compilation Error](#) or [Run Time Error](#) will occur. Note that the index of the first System IO Variable is [0, not 1](#).

Example

To set the value of System IO variable number 2 to 1.23 :

```
SetRealSystemIO(2, 1.23);
```

To set the value of System IO variable called "Setting" to the value of variable x :

```
SetRealSystemIO("Setting", x);
```

4.17.2.10 SetStringSystemIO Procedure

The SetStringSystemIO procedure sets the value of an [String System IO](#) variable.

Syntax

```
SetStringSystemIO(n, v);
```

n is an [Integer](#) or System IO Variable [System IO Tag](#).

v is a string expression

Description

The value of System IO variable number n is set to v. If System IO variable number n is not of the correct type, a [Compilation Error](#) or [Run Time Error](#) will occur. Note that the index of the first System IO Variable is [0, not 1](#).

Example

To set the value of System IO variable number 2 to 'stop' :

```
SetStringSystemIO(2, 'stop');
```

To set the value of System IO variable called "Alarm State" to 'Armed' :

```
SetStringSystemIO("Alarm State", 'Armed');
```

4.17.3 In-Built System IO Variables

In-built System Input/Output (IO) Variables provide access to system functions like :

- Scenes
- Schedules
- Irrigation
- Other C-Bus Applications (Measurement, Security, Telephony, HVAC etc)

For a full list of In-Built System IO Variables, their functions and their values, refer to the main help file.

The easiest way to use the In-built System IO variables in logic is:

- Click in the [Code Window](#)
- Right click to show the [pop-up menu](#)
- Select **System IO**
- Select either **Set IB System IO** or **Get IB System IO**
- Select the desired variable and its properties
- Click on **OK**

Below is a table showing the list of In-Built System IO Variables, their type, whether they can be set and their parameters. **Note that although logic has access to all In-built System IO Variables, it is not necessarily sensible to use some of them from logic.**

Name	Type	Settable	Parameters
		?	
Access Level Name	String	no	None
Access Level	Integer ¹	no	None
Access Logged In	Boolean	no	None
Access Password	String	no	None
Access User Name	String	no	None
Alarm Sounding	Boolean	yes	None
Alarm, Cancel Next	Boolean	yes	None
Audio Balance	Integer	yes	Matrix Switcher, Zone
Audio Bass	Integer	yes	Matrix Switcher, Zone
Audio High Priority	Boolean	yes	Matrix Switcher, Source, Level
Audio Last Error Code	Integer	no	Matrix Switcher, Zone
Audio Mute	Boolean	yes	Matrix Switcher, Zone
Audio Timer	Integer	yes	Matrix Switcher, Zone
Audio Treble	Integer	yes	Matrix Switcher, Zone
Audio Volume	Integer	yes	Matrix Switcher, Zone
Audio Zone Source	Integer ¹	yes	Matrix Switcher, Zone
C-Bus State	Boolean	no	None
Date to be set	Date (Integer)	yes	None
Daylight Savings	Boolean	no	None
E-Mail Account Count	Integer	no	None
E-Mail Account Name	String	no	None
E-Mail Account Number	Integer ¹	yes	None
E-Mail Count	Integer	no	Account Number ⁰
E-Mail Message Body	String	no	None
E-Mail Message Count	Integer	no	None
E-Mail Message Number	Integer ¹	no	None
E-Mail Message Present	Boolean	no	None
E-Mail Message Sender	String	no	None
E-Mail Message Subject	String	no	None
E-Mail Present	Boolean	no	Account Number ⁰
Energy Tariff	Real	yes	Tariff number ⁰
Error App Project Most Recent Error Name	String	no	None
Error App Project Most Recent Error Severity	Integer	no	None
Error App Project Most Recent Error Status	Boolean	no	None
Error App Project Most Severe Error Name	String	no	None
Error App Project Most Severe Error Severity	Integer	no	None
Error App Project Most Severe Error Status	Boolean	no	None
Error App Network Most Recent Error Name	String	no	Network Number
Error App Network Most Recent Error Severity	Integer	no	Network Number
Error App Network Most Recent Error Status	Boolean	no	Network Number
Error App Network Most Severe Error Name	String	no	Network Number

Error App Network Most Severe Error Severity	Integer	no	Network Number
Error App Network Most Severe Error Status	Boolean	no	Network Number
Error App Unit Most Recent Error Name	String	no	Network Number, Category, Unit Number
Error App Unit Most Recent Error Severity	Integer	no	Network Number, Category, Unit Number
Error App Unit Most Recent Error Status	Boolean	no	Network Number, Category, Unit Number
Error App Unit Most Severe Error Name	String	no	Network Number, Category, Unit Number
Error App Unit Most Severe Error Severity	Integer	no	Network Number, Category, Unit Number
Error App Unit Most Severe Error Status	Boolean	no	Network Number, Category, Unit Number
HVAC Data Valid	Boolean	no	Zone Group, Zone Numbers ²
HVAC Error Number	Integer	no	Zone Group, Zone Numbers ²
HVAC Error Name	String	no	Zone Group, Zone Numbers ²
HVAC Error State	Boolean	no	Zone Group, Zone Numbers ²
HVAC Fan Speed	Integer	yes	Zone Group, Zone Numbers ²
HVAC Fan Speed Text	String	no	Zone Group, Zone Numbers ²
HVAC Mode	Integer	yes	Zone Group, Zone Numbers ²
HVAC Mode Name	String	no	Zone Group, Zone Numbers ²
HVAC Operating Type	Integer	yes	Zone Group, Zone Numbers ²
HVAC Operating Type Name	String	no	Zone Group, Zone Numbers ²
HVAC Plant Busy	Boolean	no	Zone Group, Zone Numbers ²
HVAC Plant Cooling	Boolean	no	Zone Group, Zone Numbers ²
HVAC Plant Equipment	Integer	no	Zone Group, Zone Numbers ²
HVAC Plant Equipment Name	String	no	Zone Group, Zone Numbers ²
HVAC Plant Fan	Boolean	no	Zone Group, Zone Numbers ²
HVAC Plant Heating	Boolean	no	Zone Group, Zone Numbers ²
HVAC Sensor Status	Integer	no	Zone Group, Zone Numbers ²
HVAC Sensor Status Name	String	no	Zone Group, Zone Numbers ²
HVAC Set-Level	Real	yes	Zone Group, Zone Numbers ²
HVAC Set-Level Text	String	no	Zone Group, Zone Numbers ²
HVAC Set-Level Type	Integer	no	Zone Group, Zone Numbers ²
HVAC Set-Level Type Name	String	no	Zone Group, Zone Numbers ²
HVAC Setback Enabled	Boolean	yes	Zone Group, Zone Numbers ²
HVAC Temperature	Real	no	Zone Group, Zone Numbers ²
HVAC Zone Damper Open	Boolean	no	Zone Group, Zone Numbers ²
HVAC Zone Enabled	Boolean	yes	Zone Group, Zone Numbers ²
HVAC Zone Group State	Boolean	yes	Zone Group
Irrigation Program Count	Integer	no	None
Irrigation Program Name	String	no	None
Irrigation Program Number	Integer ¹	yes	None
Irrigation Running Zone Name	String	no	None
Irrigation Running	Boolean	no	None
Irrigation Time Remaining	Time (Integer)	no	None
Irrigation Zone Duration	Time (Integer)	yes	Offset from selected zone. Leave as 0.
Irrigation Zone Count	Integer	no	None
Irrigation Zone Name	String	no	None
Irrigation Zone Number	Integer ¹	yes	None
Irrigation Zone Running	Boolean	no	Offset from selected zone. Leave as 0.
Label Action Selector Text	String	yes	Network, Application, Group, Action Selector, Variant ⁰

Label Group Text	String	yes	Network, Application, Group, Variant ⁰
Label Language Name	String	no	None
Label Language	Integer	yes	None
Load Monitor Load Power	Real	no	Network, Application, Group
Load Monitor Total Power	Real	no	None
Logic Module Enabled	Boolean	yes	Module Number ⁰
Logic Running	Boolean	no	None
Logic Timer Time	Time (Integer)	no	Timer Number
Master Unit	Boolean	yes	None
Measurement App Boolean Value	Boolean	yes	Network, Device Id, Channel ⁰
Measurement App Energy Value	Real	yes	Network, Device Id, Channel ⁰ , Period Type, Period Quantity, Offset, Unit Type, Data Type
Measurement App Integer Value	Integer	yes	Network, Device Id, Channel ⁰
Measurement App Power Maximum	Real	no	Network, Device Id, Channel ⁰ , Period Type, Period Quantity, Offset, Unit Type
Measurement App Power Value	Real	yes	Network, Device Id, Channel ⁰ , Unit Type
Measurement App Real Value	Real	yes	Network, Device Id, Channel ⁰
Measurement App Valid Value	Boolean	no	Network, Device Id, Channel ⁰
Media Transport Control Category	Integer	yes	Media Link Group , Offset ³
Media Transport Control Category Name	String	yes	Media Link Group
Media Transport Control Fast Forward	Integer	yes	Media Link Group
Media Transport Control Pause	Boolean	yes	Media Link Group
Media Transport Control Play	Boolean	yes	Media Link Group
Media Transport Control Power	Boolean	yes	Media Link Group
Media Transport Control Repeat	Integer	yes	Media Link Group
Media Transport Control Rewind	Integer	yes	Media Link Group
Media Transport Control Selection	Integer	yes	Media Link Group, Offset ³
Media Transport Control Selection Name	String	yes	Media Link Group
Media Transport Control Shuffle	Boolean	yes	Media Link Group
Media Transport Control Stop	Boolean	yes	Media Link Group
Media Transport Control Track	Integer	yes	Media Link Group, Offset ³
Media Transport Control Track Count	Integer	yes	Media Link Group
Media Transport Control Track Name	String	yes	Media Link Group
Metric Units	Boolean	yes	None
Monitor Value	Real	no	Network, Unit, Parameter ⁴
Page Name	String	no	None
Power Meter Maximum	Real	no	Period Type, Period Quantity, Offset, Unit Type, Tariff
Power Meter Total	Real	no	Unit Type, Tariff
Power Meter Total Energy	Real	no	Period Type, Period Quantity, Offset, Unit Type, Data Type, Tariff
Pulse Power Meter Energy			Meter Number ⁰ , Period Type, Period Quantity, Offset, Unit Type, Data Type
Pulse Power Meter Level	Real	no	Power Meter Number ⁰
Pulse Power Meter Maximum	Real	no	Meter Number ⁰ , Period Type, Period Quantity, Offset, Unit Type
Scene Component Count	Integer	no	None
Scene Component Current Level	Integer	yes	Offset from selected component. Leave as 0.
Scene Component Level	Integer	yes	Offset from selected component.

Scene Component Name	String	no	Leave as 0. Offset from selected component. Leave as 0.
Scene Component Number	Integer ¹	yes	None
Scene Component Ramp Rate	Integer	yes	Offset from selected component. Leave as 0.
Scene Component Ramp Rate Text	String	no	Offset from selected component. Leave as 0.
Schedule Controls C-Bus Group	Boolean	no	None
Scene Count	Integer	no	None
Scene Current Level	Integer	yes	None
Schedule Has Start & End Time	Boolean	no	None
Scene Name	String	no	None
Scene Nudge Level	Integer	yes	None
Scene Number	Integer ¹	yes	None
Schedule Also On	Boolean	yes	Special Day (0 = normal, 1 = public holiday, 2 = special day 1 etc)
Schedule Any Year	Boolean	yes	None
Schedule Count	Integer	no	None
Schedule Day of Month	Boolean	yes	Day of month
Schedule Day of Month Mask	Integer	yes	None
Schedule Day of Month Type	Time (Integer)	no	None
Schedule Day of Week	Boolean	yes	Day (1 = Sunday, 7 = Saturday)
Schedule Day of Week Mask	Integer	yes	None
Schedule Day of Week Type	Time (Integer)	no	None
Schedule Day Text	String	no	None
Schedule Day Type	Integer	yes	None
Schedule Enabled	Boolean	yes	None
Schedule End	Time (Integer)	yes	None
Schedule Event	String	no	None
Schedule Is Last Week of Month	Boolean	yes	None
Schedule Is Repeat	Boolean	yes	None
Schedule Level	Integer	yes	None
Schedule Month	Boolean	yes	Month (1 = January, 12 = December)
Schedule Month Mask	Integer	yes	None
Schedule Name	String	no	None
Schedule Next Due	Time (Integer)	no	None
Schedule Not On	Boolean	yes	Special Day (0 = normal, 1 = public holiday, 2 = special day 1 etc)
Schedule Number	Integer ¹	yes	None
Schedule Pulse Duration	Time (Integer)	yes	None
Schedule Repeat Due	Date (Integer)	yes	None
Schedule Repeat Interval	Integer	yes	None
Schedule Start	Time (Integer)	yes	None
Schedule Time	Time (Integer)	yes	None
Schedule Time Text	String	no	None
Schedule Time Type	Time (Integer)	no	None
Schedule Year	Integer	yes	None
Security Alarm Sounding	Boolean	no	None
Security All Zones Secure	Boolean	no	None
Security Arm Failed	Boolean	no	None
Security Arm Ready	Boolean	no	None
Security Armed Level Name	String	no	None
Security Armed Level	Integer	no	None
Security Armed State	Boolean	no	None
Security Battery Charging	Boolean	no	None
Security Entry Delay	Boolean	no	None
Security Exit Delay	Boolean	no	None
Security Fire Alarm Sounding	Boolean	no	None
Security Gas Alarm Sounding	Boolean	no	None

Security Line Cut Alarm Sounding	Boolean	no	None
Security Low Battery	Boolean	no	None
Security Mains Failure	Boolean	no	None
Security Normal Operation	Boolean	no	None
Security Other Alarm Sounding	Boolean	no	None
Security Panic	Boolean	no	None
Security Password OK	Boolean	no	None
Security Password Status Name	String	no	None
Security Password Status	Integer	no	None
Security Tamper	Boolean	no	None
Security Zone Isolated	Boolean	no	Zone number ⁰
Security Zone Name	String	no	Zone number ⁰
Security Zone Secure	Boolean	no	Zone number ⁰
Security Zone Status Name	String	no	Zone number ⁰
Security Zone Status	Integer	no	Zone number ⁰
Special Day Type	String	no	None
System Free Memory	Real	no	None
System Processor Usage	Real	no	None
Telephony Dial In Fail Reason Name	String	no	None
Telephony Dial In Fail Reason	Integer	no	None
Telephony Dial Out Fail Reason	String	no	None
Name			
Telephony Dial Out Fail Reason	Integer	no	None
Telephony Diverted Number	String	no	None
Telephony Diverted	Boolean	no	None
Telephony Last Incoming Number	String	no	None
Telephony Last Outgoing Number	String	no	None
Telephony Off Hook Number	String	no	None
Telephony Off Hook Reason Name	String	no	None
Telephony Off Hook Reason	Integer	no	None
Telephony Off Hook	Boolean	no	None
Telephony Ringing Number	String	no	None
Telephony Ringing	Boolean	no	None
Telephony Secondary Isolated	Boolean	no	None
Time to be set	Time (Integer)	yes	None
Time-out Page	Integer	yes	None

Notes

⁰ parameters which are [Zero Based](#) start from 0, not 1. So for example, the first Module number is 0, not 1. When possible, use [Tags](#) to avoid errors.

¹ All other values are one based. So, for example, to select the first Schedule, set the Schedule Number to 1.

² HVAC Zones are a "bit mask". See [Controlling HVAC](#)

³ An offset of 0 is the current Category, selection or track. Offset of 1 is the next one, and offset of 2 is the one after that. The names can only be set if the Media Link Group has the server property set.

⁴ The parameter types are listed in [GetUnitParameter Function](#)

The following functions can be used with User System IO variables :

- [GetBoolIBSystemIO Function](#)
- [GetIntIBSystemIO Function](#)
- [GetRealIBSystemIO Function](#)
- [GetStringIBSystemIO Procedure](#)
- [SetBoolIBSystemIO Procedure](#)
- [SetIntIBSystemIO Procedure](#)
- [SetRealIBSystemIO Procedure](#)
- [GetStringIBSystemIO Procedure](#)

See also [Controlling HVAC](#)

4.17.3.1 System IOTags

An In-Built System IO Tag is a [Tag](#) used to refer to an In-Built System IO Variable number. The tag corresponds to the In-Built System IO Variable name.

The format is :

"System IO Variable Name"

In-Built System IO variables can only be referred to by their tags, not by their number.

Where an in-built system IO procedure or function refers to a C-Bus Network, Application, Group Address or Level/Action Selector, a [C-Bus tag](#) can be used.

In addition, there are tags for use with [Pulse Power Meters](#) and [Energy Tariffs](#).

4.17.3.2 GetBoolIBSystemIO Function

The GetBoolIBSystemIO function returns the value of a [Boolean In-Built System IO Variable](#).

Syntax

```
GetBoolIBSystemIO(name [, parameter1] [, parameter2] [, parameter3])
```

name is an [In-Built System IO Variable Tag](#).

parameter1, parameter2 and parameter3 are optional integer parameters which depend on the selected In-Built System IO Variable.

Description

The boolean result is the value of the selected [In-Built System IO Variable](#). Refer to the main help file for details of what the In-built System IO variables do.

Examples

To assign the value of In-Built System IO variable called "Access Logged In" to a variable called State :

```
State := GetBoolIBSystemIO("Access Logged In");
```

To perform an action if someone is currently logged in :

```
if GetBoolIBSystemIO("Access Logged In") then ...
```

Other boolean System IO variable examples :

-
- Network 50 has an error reported on it :

```
GetBoolIBSystemIO("Error App Network Error", 50);
```
- Network 50, unit 23 (category 884) has an error reported on it :

```
GetBoolIBSystemIO("Error App Unit Error", 50, 884, 23)
```
- The data for [HVAC Zone Group 1, Zone 2](#) is valid :

```
GetBoolIBSystemIO("HVAC Data Valid", 1, HVACZone2)
```
- HVAC Zone Group 1 in on :

```
GetBoolIBSystemIO("HVAC Zone Group State", 1)
```
- The currently selected irrigation zone running :

```
GetBoolIBSystemIO("Irrigation Zone Running", 0)
```

-
- The second logic module is enabled (note [Zero Based](#)) :

```
GetBoolIBSystemIO("Logic Module Enabled", 1)
```
- Security Zone 4 is secure :

```
GetBoolIBSystemIO("Security Zone Secure", 4)
```

See also [Measurement Application System IO](#)

4.17.3.3 GetIntIBSystemIO Function

The GetIntIBSystemIO function returns the value of an [Integer In-Built System IO Variable](#).

Syntax

```
GetIntIBSystemIO(name [, parameter1] [, parameter2] [, parameter3])
```

name is an [In-Built System IO Variable Tag](#).

parameter1, parameter2 and parameter3 are optional integer parameters which depend on the selected In-Built System IO Variable.

Description

The integer result is the value of the selected [In-Built System IO Variable](#). Refer to the main help file for details of what the In-built System IO variables do.

Examples

To assign the value of In-Built System IO variable called "Access Level" to a variable called CurrentLevel :

```
CurrentLevel := GetIntIBSystemIO("Access Level");
```

To perform an action if someone is currently logged in :

```
if GetIntIBSystemIO("Access Level") > 0 then ...
```

Other integer System IO variable examples :

- Network 120 error severity :

```
GetIntIBSystemIO("Error App Network Error Severity", 120)
```
- Network 120, unit 23 (category 884) error severity :

```
GetIntIBSystemIO("Error App Unit Error Severity", 120, 884, 23)
```
- [HVAC](#) Zone Group 1, Zone 2 operational type :

```
GetIntIBSystemIO("HVAC Operating Type", 1, HVACZone2)
```
- Irrigation time for the selected program and zone :

```
GetIntIBSystemIO("Irrigation Time", 0)
```
- C-Bus Level for the selected scene component :

```
GetIntIBSystemIO("Scene Component Level", 0)
```
- Status of Security Zone 4 :

```
GetIntIBSystemIO("Security Zone Status", 4)
```

See also [Measurement Application System IO](#)

4.17.3.4 GetRealIBSystemIO Function

The GetRealIBSystemIO function returns the value of a [Real In-Built System IO Variable](#).

Syntax

```
GetRealIBSystemIO(name [, parameter1] [, parameter2] [, parameter3])
```

name is an [In-Built System IO Variable Tag](#).

parameter1, parameter2 and parameter3 are optional integer parameters which depend on the selected In-Built System IO Variable.

Description

The real result is the value of the selected [In-Built System IO Variable](#). Refer to the main help file for details of what the In-built System IO variables do.

Examples

To assign the value of In-Built System IO variable for the load energy for Group 1 on Application 56 (lighting) on network 254 to a variable called Energy :

```
Energy := GetRealIBSystemIO("Load Monitor Load Energy", 254, 56, 1);
```

To perform an action if the [HVAC](#) set-point level for Zone Group 1, Zone 2 is over 25C :

```
if GetRealIBSystemIO("HVAC Set-Level", 1, HVACZone2) > 25 then ...
```

To read the value of a temperature "monitor" on network 254, unit address 123 and assign it to a variable called OutsideTemp:

```
OutsideTemp := GetRealIBSystemIO("Monitor Value", 254, 123, 1);
```

(Note that the [GetUnitParameter Function](#) function is normally used to read monitor data values.)

See also [Using Power and Energy Data](#) and [Measurement Application System IO](#)

4.17.3.5 **GetStringIBSystemIO Procedure**

The GetStringIBSystemIO procedure returns the value of a [String In-Built System IO Variable](#).

Syntax

```
GetStringIBSystemIO(name [, parameter1] [, parameter2] [, parameter3] [, parameter4] [, parameter5], StringVar);
```

name is an [In-Built System IO Variable Tag](#).

parameter1...parameter5 are optional integer parameters which depend on the selected In-Built System IO Variable.

StringVar is a string variable

Description

The value of the selected [In-Built System IO Variable](#) is stored in the string variable. Refer to the main help file for details of what the In-built System IO variables do.

Examples

String System IO variable examples :

- Network 120 error severity :

```
GetIntIBSystemIO("Error App Network Error Severity Name", 120)
```
- Network 120, unit 23 (category 884) error severity name :

```
GetIntIBSystemIO("Error App Unit Error Severity Name", 120, 884, 23)
```
- [HVAC](#) Zone Group 1, Zone 2 operational type name :

```
GetIntIBSystemIO("HVAC Operating Type Name", 1, HVACZone2)
```
- The state of measurement application network 254, unit 5, channel 2 (note [Zero Based](#)) :

```
GetIntIBSystemIO("Measurement App Integer Value", 254, 5, 1)
```
- Name of the selected scene component :

```
GetIntIBSystemIO("Scene Component Name", 0)
```

- Name of Security Zone 4 :
`GetIntIBSystemIO("Security Zone Name", 4)`

See also [C-Bus Labels](#)

4.17.3.6 SetBoolIBSystemIO Procedure

The SetBoolSystemIO procedure sets the value of a [boolean In-Built System IO Variable](#).

Syntax

```
SetBoolIBSystemIO(name [, parameter1], v);
```

name is an [In-Built System IO Variable Tag](#).

parameter1 is an optional integer parameter which depends on the selected In-Built System IO Variable.

v is a boolean expression

Description

The value of the [In-Built System IO Variable](#) is set to v. Refer to the main help file for details of what the In-built System IO variables do.

Examples

To enable the first logic module :

```
SetBoolIBSystemIO("Logic Module Enabled", 0, true);
```

See also [additional examples](#)

4.17.3.7 SetIntIBSystemIO Procedure

The SetIntSystemIO procedure sets the value of a [Integer In-Built System IO Variable](#).

Syntax

```
SetIntIBSystemIO(name [, parameter1], v);
```

name is an [In-Built System IO Variable Tag](#).

parameter1 is an optional integer parameter which depends on the selected In-Built System IO Variable.

v is an integer expression

Description

The value of the [In-Built System IO Variable](#) is set to v. Refer to the main help file for details of what the In-built System IO variables do.

Examples

To set the label language to English (Australian):

```
SetIntIBSystemIO("Label Language", 2);
```

To select the first Schedule so that it can be used:

```
SetIntIBSystemIO("Schedule Number", 1);
```

See also [additional examples](#)

4.17.3.8 SetRealIBSystemIO Procedure

The SetRealSystemIO procedure sets the value of a [real In-Built System IO Variable](#).

Syntax

```
SetRealIBSystemIO(name [, parameter1], v);
```

name is an [In-Built System IO Variable Tag](#).

parameter1 is an optional integer parameter which depends on the selected In-Built System IO Variable.

v is a real expression

Description

The value of the [In-Built System IO Variable](#) is set to v. Refer to the main help file for details of what the In-built System IO variables do.

Examples

To set the [HVAC](#) set-point level for Zone Group 1, Zone 2 to 25C :

```
SetRealIBSystemIO("HVAC Set-Level", 1, HVACZone2, 25);
```

See also [additional examples](#) and [Using Power and Energy Data](#)

4.17.3.9 SetStringIBSystemIO Procedure

The SetStringSystemIO procedure sets the value of a [String In-Built System IO Variable](#).

Syntax

```
SetStringIBSystemIO(name [, parameter1] [, parameter2] [, parameter3] [, parameter4] [, parameter5], v);
```

name is an [In-Built System IO Variable Tag](#).

parameter1...parameter5 are optional integer parameters which depend on the selected In-Built System IO Variable.

v is a string expression

Description

The value of the [In-Built System IO Variable](#) is set to v. Refer to the main help file for details of what the In-built System IO variables do.

Examples

See [additional examples](#)

See [C-Bus Labels](#)

4.17.3.10 Controlling HVAC

When using in-built System IO variables to control the HVAC Application, the following in-built System IO variables can be used :

- HVAC Fan Speed
- HVAC Mode
- HVAC Operating Type

- HVAC Setback Enabled
- HVAC Set-Level
- HVAC Zone Enabled
- HVAC Zone Group State

Other in-built System IO variables can be used to monitor the state of various aspects of an HVAC system.

For all except the last two listed above, it is possible to control than more than one zone at once. To support this, the Zone Numbers parameter is actually a "bit mask". The values and logic constants used to control different combinations of zones are shown below.

Unswitched	Zone 1	Zone 2	Zone 3	Zone 4	Bit Mask	Constant
✓					1	HVACZoneU
	✓				2	HVACZone1
✓	✓				3	HVACZoneU1
		✓			4	HVACZone2
✓		✓			5	HVACZoneU2
	✓	✓			6	HVACZone12
✓	✓	✓			7	HVACZoneU12
			✓		8	HVACZone3
✓			✓		9	HVACZoneU3
	✓		✓		10	HVACZone13
✓	✓		✓		11	HVACZoneU13
		✓	✓		12	HVACZone23
✓		✓	✓		13	HVACZoneU23
	✓	✓	✓		14	HVACZone123
✓	✓	✓	✓		15	HVACZoneU123
				✓	16	HVACZone4
✓				✓	17	HVACZoneU4
	✓			✓	18	HVACZone14
✓	✓			✓	19	HVACZoneU14
		✓		✓	20	HVACZone24
✓		✓		✓	21	HVACZoneU24
	✓	✓		✓	22	HVACZone124
✓	✓	✓		✓	23	HVACZoneU124
			✓	✓	24	HVACZone34
✓			✓	✓	25	HVACZoneU34
	✓		✓	✓	26	HVACZone134
✓	✓		✓	✓	27	HVACZoneU134
		✓	✓	✓	28	HVACZone234
✓		✓	✓	✓	29	HVACZoneU234
	✓	✓	✓	✓	30	HVACZone1234
✓	✓	✓	✓	✓	31	HVACZoneU1234

To set the HVAC set-point level for Zone Group 1, Zone 2 to 25C :

```
SetRealIBSystemIO("HVAC Set-Level", 1, HVACZone2, 25);
```

To set the HVAC set-point level for Zone Group 2, Zone 2, 3 and 4 to 25C :

```
SetRealIBSystemIO("HVAC Set-Level", 2, HVACZone234, 25);
```

When using the value of a zone, it is only possible to get the value of one zone at a time, but the above constants are still used.

To perform an action if the HVAC set-point level for Zone Group 1, Zone 2 is over 25C :

```
if GetRealIBSystemIO("HVAC Set-Level", 1, HVACZone2) > 25 then ...
```

If you were to get the value of a HVAC in-built System IO variable and you selected multiple zones, it would just use the first enabled zone in the list. So, the code below would be the same as that above :

```
if GetRealIBSystemIO("HVAC Set-Level", 1, HVACZone234) > 25 then ...
```

The HVAC Set Level value varies depending on the mode of the HVAC plant. It generally is a temperature, but can also be a Comfort Level (if using an evaporative plant) or a fan speed (when in fan-only mode). The type of value can be found using the HVAC Set Level Type in-built System IO variable.

Before HVAC data is used in logic, you should check to make sure that the data is valid. There will be a short period following start-up when the data is not yet available. For example :

```
if GetBoolIBSystemIO("HVAC Data Valid", 1, HVACZone2) then
begin
  if GetRealIBSystemIO("HVAC Set Level", 1, HVACZone2) > 25 then
    ...
end;
```

4.17.3.11 Measurement Application

There are three in-built System IO variables for the values of C-Bus Measurement Application channels:

- Measurement App Boolean Value
- Measurement App Integer Value
- Measurement App Real Value

Using the Channel Data

Measurement Application channels need to be created in the Measurement Application Manager before they can be used.

The following values will be reported for system IO variables prior to a value being received via C-Bus:

- Integer values: 0
- Real Values: 0.0
- Boolean Values: false

During this time, the "Measurement App Valid Value" In-built System IO variable will have a value of false. The value of the channel data should be ignored during this time, as the correct value is unknown.

Once a C-Bus message has been received containing a value for the channel, the corresponding system IO variable will have the correct value and the "Measurement App Valid Value" will have a value of true.

 Note that the channel numbers of the General Input Unit are [Zero Based](#). The first channel is actually 0, not 1.

Examples

In the following examples, a General Input Unit is assumed to be on network 254, Device Id 5.

To perform an action if the state of the first channel is on:

```
if GetBoolIBSystemIO("Measurement App Valid Value", 254, 5, 0) and
```

```
GetBoolIBSystemIO("Measurement App Boolean Value", 254, 5, 0) then ...
```

To assign the value of the second channel to a variable called OutsideTemp:

```
if GetBoolIBSystemIO("Measurement App Valid Value", 254, 5, 1) then
begin
  OutsideTemp := GetRealIBSystemIO("Measurement App Real Value", 254, 5, 1);
  ...
end;
```

To perform an action if the value of the third channel is more than 100:

```
if GetBoolIBSystemIO("Measurement App Valid Value", 254, 5, 2) then
begin
  if GetIntIBSystemIO("Measurement App Integer Value", 254, 5, 2) > 100 then
  ...
  ...
end;
```

Controlling the Channel Data

Setting a Measurement App in-built System IO variable is only possible if the Controllable property is set. In this case, the value will also be broadcast onto C-Bus. Avoid changing these too often as it can create excessive C-Bus Traffic.

Examples

To set the value of measurement channel 0 on network 254, Device Id 20 to 100 and delay for 30 seconds:

```
SetIntSystemIO("Measurement App Integer Value", 254, 20, 0, 100);
delay(30);
```

To set the value of measurement channel 0 on network 254, Device Id 20 to the value of variable OutsideTemp if it has changed:

```
if HasChanged(OutsideTemp) then
  SetIntSystemIO("Measurement App Integer Value", 254, 20, 0, OutsideTemp);
```

See also [Using Power and Energy Data](#)

4.17.3.12 C-Bus Labels

C-Bus Group Addresses and levels can be "labelled" to display text on a DLT switch or elsewhere.

The In-built System IO Variables which can be used for this are:

- Label Group Text
- Label Action Selector Text

Group Labels

To assign the label for Group Address 2 (for "Variant 1") to a variable called Name :

```
GetStringIBSystemIO("Label Group Text", 254, 56, 2, Variant1, Name);
```

or you can use [C-Bus tags](#) :

```
GetStringIBSystemIO("Label Group Text", "Local", "Lighting", "Load 2",
Variant1, Name);
```

To compare the label with the text 'kitchen' (using string variable called s) :

```
GetStringIBSystemIO("Label Group Text", 254, 56, 2, Variant1, s);
if s = 'kitchen' then ...
```

To send a label for group address 2, Variant 1 :

```
SetStringIBSystemIO("Label Group Text", 254, 56, 2, Variant1, 'New label');
```

or you can use [C-Bus tags](#) :

```
SetStringIBSystemIO("Label Group Text", "Local", "Lighting", "Load 2",
Variant1, 'New Label');
```

Action Selector Labels

To read the label for Trigger Group 1, Action Selector number 255 into a variable s:

```
GetStringIBSystemIO("Label Action Selector Text", "Local", "Trigger
Control", "Trigger Group 1", 255, Variant1, s);
```

To set the label for Trigger Group 1, Action Selector 0, Variant 1:

```
SetStringIBSystemIO("Label Action Selector Text", "Local", "Trigger Control", "Trigger C
```

Variants

Variant numbers are [Zero Based](#), so Variant 1 is actually the number 0 and so forth. To minimise the possibility of confusion, it is recommended that the variant constants be used:

Constant	Value
Variant1	0
Variant2	1
Variant3	2
Variant4	3

Language

Labels will be read and sent using the language most recently set. The default language is English (language number 1).

The language can be set using the Label Language In-built System IO Variable. To set the language to German (language number 50 [hexadecimal](#)):

```
SetIntIBSystemIO("Label Language", $50);
```

4.17.3.13 Using Power and Energy Data

Real-time Power Level

Several in-built System IO Variables can be used with power data:

- Pulse Power Meter Level - Measured power for an individual Pulse Power Meter
- Measurement App Power Value - Current power level for an Analogue Power Meter
- Power Meter Total - Total power of all meters

To get the the power level for a pulse power meter:

```
GetRealIBSystemIO("Pulse Power Meter Level", MeterNumber, UnitType);
```

where

MeterNumber is the index of the Pulse Power Meter (0 is the first one) or tag

UnitType is the units the data is required in:

- 0 = Watts
- 1 = kg of CO₂ per hour (uses the Carbon Footprint value)
- 2 = Cost per hour (uses the energy Tariff)

For example, to store the power level (Watts) for a pulse power meter called "Meter A" into a variable called "Power":

```
Power := GetRealIBSystemIO("Pulse Power Meter Level", "Meter A", 0);
```

To get the the power level for an analogue power meter:

```
GetRealIBSystemIO("Measurement App Power Value", Network, DeviceId, Channel,
UnitType);
```

where

Network is the C-Bus Network number

DeviceId is the Measurement Application Device Identifier

Channel is the Measurement Application channel number (0 is the first channel)

UnitType is the units the data is required in (as above)

For example, to get the power level for network 254, Device Id 2, channel 3 (the fourth channel) in kg of CO₂ per hour:

```
Power := GetRealIBSystemIO("Measurement App Power Value", 254, 2, 3, 1);
```

To get the the total power level for all power meters:

```
GetRealIBSystemIO("Power Meter Total", UnitType, TariffNo);
```

Where TariffNo is the number of the tariff of the meters to be included in the total. Use a value of -1 to get the total for all meters.

For example, to get the total cost per hour of all power meters:

```
Cost := GetRealIBSystemIO("Power Meter Total", 2, -1);
```

For example, to get the total cost per hour of all power meters using the first tariff (in this case, this tariff is used for off-peak power):

```
Cost := GetRealIBSystemIO("Power Meter Total", 2, 0);
```

Peak Power

Several in-built System IO Variables can be used to find the peak (maximum) power data:

- Pulse Power Meter Maximum - Recorded peak power for an individual Pulse Power Meter
- Measurement App Power Maximum - Recorded peak power for an Analogue Power Meter
- Power Meter Maximum - Recorded peak power for total of all meters

To get the maximum power level over a period of time for a pulse power meter:

```
GetRealIBSystemIO("Pulse Power Meter Maximum", MeterNumber, PeriodType, PeriodQuantity, Offset,
```

where

MeterNumber is the index of the Pulse Power Meter (0 is the first one) or tag

PeriodType is the units of the duration:

- 0 = Hours
- 1 = Days
- 2 = Weeks
- 3 = Months

PeriodQuantity is the number of periods

Offset is the offset back from today (in multiples of PeriodQuantity x PeriodType)

UnitType is the units the data is required in:

- 0 = Watts
- 1 = kg of CO₂ per hour (uses the Carbon Footprint value)
- 2 = Cost per hour (uses the energy Tariff)

For example, to find the maximum power (in Watts) over the past 7 days for pulse power meter "Meter A":

```
Power := GetRealIBSystemIO("Pulse Power Meter Maximum", "Meter A", 1, 7, 0, 0);
```

To find the maximum for the 7 days prior to that, the offset will be 1:

```
Power := GetRealIBSystemIO("Pulse Power Meter Maximum", "Meter A", 1, 7, 1, 0);
```

To get the maximum power level over a period of time for an analogue power meter:

```
GetRealIBSystemIO("Measurement App Power Maximum", Network, DeviceId, Channel, PeriodType,
```

where

Network is the C-Bus Network number

DeviceId is the Measurement Application Device Identifier

Channel is the Measurement Application channel number (0 is the first channel)

PeriodType is the units of the duration (as above)

PeriodQuantity is the number of periods

Offset is the offset back from today (in multiples of PeriodQuantity x PeriodType)

UnitType is the units the data is required in (as above)

For example, to get the maximum power level for network 254, Device Id 2, channel 3 (the fourth channel) in kg of CO₂ per hour over the past 3 months:

```
Power := GetRealIBSystemIO("Measurement App Power Maximum", 254, 2, 3, 3, 3, 0, 1);
```

To get the the peak total power level for all power meters:

```
GetRealIBSystemIO("Power Meter Maximum", PeriodType, PeriodQuantity, Offset, UnitType, TariffNo);
```

Where TariffNo is the number of the tariff of the meters to be included in the total. Use a value of -1 to get the total for all meters.

For example, to get the peak total cost per hour of all power meters over the past week:

```
Cost := GetRealIBSystemIO("Power Meter Maximum", 2, 1, 0, 2, -1);
```

Energy

Several in-built System IO Variables can be used with energy data:

- Pulse Power Meter Energy - Recorded energy for a Pulse Power Meter
- Measurement App Energy Value - Recorded energy for an Analogue Power Meter
- Power Meter Total Energy - Total energy for all meters

To get the maximum power level over a period of time for a pulse power meter:

```
GetRealIBSystemIO("Pulse Power Meter Energy", MeterNumber, PeriodType, PeriodQuantity, Offset,
```

where

MeterNumber is the index of the Pulse Power Meter (0 is the first one) or tag

PeriodType is the units of the duration:

- 0 = Hours
- 1 = Days
- 2 = Weeks
- 3 = Months

PeriodQuantity is the number of periods

Offset is the offset back from today (in multiples of PeriodQuantity x PeriodType)

UnitType is the units the data is required in:

- 0 = kWh
- 1 = kg of CO₂ (uses the Carbon Footprint value)
- 2 = Cost (uses the energy Tariff)
- 3 = MJ

DataType is:

- 0 = To date (energy used during the period so far)
- 1 = Predicted (rough estimate for the period)
- 2 = Average (average for as far back as data is available)

For example, to find the energy (in Watt Hours) over the past 24 hours for pulse power meter "Meter A":

```
Power := GetRealIBSystemIO("Pulse Power Meter Energy", "Meter A", 0, 24, 0, 0, 0);
```

To find the energy for the 24 hours prior to that, the offset will be 1:

```
Power := GetRealIBSystemIO("Pulse Power Meter Energy", "Meter A", 0, 24, 1, 0, 0);
```

To find the predicted energy cost for this month:

```
Cost := GetRealIBSystemIO("Pulse Power Meter Energy", "Meter A", 3, 1, 0, 2, 1);
```

To get the maximum power level over a period of time for an analogue power meter:

```
GetRealIBSystemIO("Measurement App Energy Value", Network, DeviceId, Channel, PeriodType, P
```

where

Network is the C-Bus Network number

DeviceId is the Measurement Application Device Identifier

Channel is the Measurement Application channel number (0 is the first channel)

PeriodType is the units of the duration (as above)

PeriodQuantity is the number of periods

Offset is the offset back from today (in multiples of PeriodQuantity x PeriodType)

UnitType is the units the data is required in (as above)

DataType is as above

For example, to get the energy used by the analogue power meter on network 254, Device Id 2, channel 0 (the first channel) in kg of CO₂ over the past 12 months:

```
Power := GetRealIBSystemIO("Measurement App Energy Value", 254, 2, 0, 3, 12, 0,
1, 0);
```

To get the the total energy for all power meters:

```
GetRealIBSystemIO("Power Meter Total Energy", PeriodType, PeriodQuantity,
Offset, UnitType, DataType, TariffNo);
```

Where TariffNo is the number of the tariff of the meters to be included in the total. Use a value of -1 to get the total for all meters.

For example, to get the energy used by all power meters today:

```
Energy := GetRealIBSystemIO("Power Meter Total Energy", 1, 1, 0, 0, 0, -1);
```

To get the average daily energy used by all power meters:

```
Energy := GetRealIBSystemIO("Power Meter Total Energy", 1, 1, 0, 0, 2, -1);
```

Notes

When using a period which is a multiple of days, the data is from midnight to the current day and time. When using a period which is a multiple of weeks, the data is from midnight on a Monday to the current day and time. When using a period which is a multiple of months, the data is from midnight on the first of the month to the current day and time.

Predicted data is based on past history and/or the current power level. It may give unexpected results in unusual circumstances.

Energy Tariff

The energy (electricity) tariff can be used and controlled with the "Energy Tariff" In-built System IO Variable. This is the current energy tariff price (per kWh).

For example, to get the present cost for a tariff called "General":

```
TariffCost := GetRealIBSystemIO("Energy Tariff", "General");
```

Examples

The following examples show how to set the tariff for a variety of situations. These can be combined for more complex requirements.

Time of day based tariff:

```
// Adjust the energy tariff
// - midnight - 8AM = $0.12/kWh
// - 8AM - 6PM = $0.10/kWh
// - 6PM - 8PM = $0.15/kWh
// - 8PM - midnight = $0.10/kWh

case hour of
  0,1,2,3,4,5,6,7:
    Tariff := 0.12;
  8,9,10,11,12,13,14,15,16,17:
    Tariff := 0.10;
  18,19:
    Tariff := 0.15;
  20,21,22,23 :
    Tariff := 0.10;
end;

SetRealIBSystemIO("Energy Tariff", "General", Tariff);

delay("0:30:00"); // only recalculate every 1/2 hour
```

Consumption based tariff:

```
// Adjust the tariff based on the energy consumption for the day:
// - 0 to 3kWh = $0.11/kWh
// - 3 to 10kWh = $0.13/kWh
// - 10 to 30kWh = $0.14/kWh
// - 30 to 50kWh = $0.15/kWh
// - over 50kWh = $0.16/kWh

TodaysEnergy := GetRealIBSystemIO("Power Meter Total Energy", 1, 1, 0, 0,
0)/1000;

if TodaysEnergy < 3 then
  Tariff := 0.11
else if TodaysEnergy < 10 then
  Tariff := 0.13
else if TodaysEnergy < 30 then
  Tariff := 0.14
else if TodaysEnergy < 50 then
  Tariff := 0.15
else
  Tariff := 0.16;

SetRealIBSystemIO("Energy Tariff", "General", Tariff);

delay("0:30:00"); // only recalculate every 1/2 hour
```

Season based tariff:

```
// Adjust the tariff:
```

```
// - 15 Dec to 14 Mar = $0.15/kWh
// - 15 Mar to 14 Jun = $0.12/kWh
// - 15 Jun to 14 Sep = $0.14/kWh
// - 15 Sep to 14 Dec = $0.12/kWh

if DayOfYear <= "14 Mar" then
    Tariff := 0.15
else if DayOfYear <= "14 Jun" then
    Tariff := 0.12
else if DayOfYear <= "14 Sep" then
    Tariff := 0.14
else if DayOfYear <= "14 Dec" then
    Tariff := 0.12
else
    Tariff := 0.15;

SetRealIBSystemIO("Energy Tariff", "General", Tariff);

delay("0:30:00"); // only recalculate every 1/2 hour
```

Net Feed-in Tariff (used when solar panels are feeding power back into the grid):

```
// Adjust the tariff:
// - if using power from grid, tariff = $0.20/kWh
// - if supplying power to grid, tariff = $0.45/kWh

Power := GetRealIBSystemIO("Power Meter Total", 0, -1);

if Power >= 0 then
    Tariff := 0.20
else
    Tariff := 0.45

SetRealIBSystemIO("Energy Tariff", "General", Tariff);
SetRealIBSystemIO("Energy Tariff", "Solar Panel", Tariff);

delay("0:30:00"); // only recalculate every 1/2 hour
```

4.17.3.14 Schedules

Schedules can be controlled using a series of In-built System IO variables:

- Schedule Also On
- Schedule Any Year
- Schedule Count
- Schedule Day of Month
- Schedule Day of Month Mask
- Schedule Day of Month Type
- Schedule Day of Week
- Schedule Day of Week Mask
- Schedule Day of Week Type
- Schedule Day Text
- Schedule Day Type
- Schedule Enabled
- Schedule End
- Schedule Event
- Schedule Is Last Week of Month

- Schedule Is Repeat
- Schedule Level
- Schedule Month
- Schedule Month Mask
- Schedule Name
- Schedule Next Due
- Schedule Not On
- Schedule Number
- Schedule Pulse Duration
- Schedule Repeat Due
- Schedule Repeat Interval
- Schedule Start
- Schedule Time
- Schedule Time Text
- Schedule Time Type
- Schedule Year

To use or change a property of a Schedule, it is first necessary to set the value of the Schedule Number In-built System IO Variable. Note that if the logic code changes the selected Schedule, then anything on the user interface using the Schedule In-built System IO Variables will be changed too.

The properties of the selected Schedule are then available using the other In-built System IO Variables.

Bit-masks

The In-built System IO Variables with "mask" in their name use bit masks to represent the state of multiple bits of data.

To understand bit-masks, it is necessary to understand [binary numbers](#). These are related to [Hexadecimal Numbers](#), but are not explained here.

In a bit-mask, each bit of a binary number is used to represent a boolean value. In the case of the Day of The Week bit-mask, there are 7 days of the week, so there are 7 bits in the bit-mask. The least significant bit is Sunday and the most significant bit is Saturday. So the binary representation of a mask for the days Monday to Friday inclusive would be as follows:

0	0	1	1	1	1	1	0
U	S	F	T	W	T	M	S
N	A	R	H	E	U	O	U
U	T	I	U	D	E	N	N
S	U	D	R	N	S	D	D
E	R	A	S	E	D	A	A
D	D	Y	D	S	A	Y	Y
	A		A	D	Y		
	Y		Y	A			
				Y			

So the bit-mask is 0011 1110, which is 3E hexadecimal, or 62 decimal.

Similarly, the bit-mask for months of the year is a 12 bit number with the least significant bit being January. A bit mask for the months January to June inclusive would be:

0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1
U	U	U	U	D	N	O	S	A	J	J	M	A	M	F	J
N	N	N	N	E	O	C	E	U	U	U	A	P	A	E	A
U	U	U	U	C	V	T	P	G	L	N	Y	R	R	B	N
S	S	S	S												
E	E	E	E												
D	D	D	D												

which is a binary value of 0000 0000 0011 1111 which is 003F hexadecimal or 63 decimal.

The Day of the Month Mask is a 31 bit number with the least significant bit being the first day of the month. The most significant bit is not used.

Some useful values for bit-masks are given in the table below.

Bit-Mask	Use	Value (binary)	Value (hexadecimal)	Value (decimal)
Day of Week	No days	0000 0000	00	0
Day of Week	All Days	0111 1111	7F	127
Day of Week	Monday - Friday	0011 1110	3E	62
Day of Week	Saturday & Sunday	0100 0001	41	65
Day of Month	No Days	0000 0000 0000 0000 0000 0000 0000 0000	0000	0
Day of Month	All Days	0111 1111 1111 1111 1111 1111 1111 1111	7FFFFFFF	2147483647
Day of Month	First week	0000 0000 0000 0000 0000 0000 0111 1111	0000007F	127
Day of Month	Second week	0000 0000 0000 0000 0011 1111 1000 0000	00003F80	16256
Day of Month	Third Week	0000 0000 0001 1111 1100 0000 0000 0000	001FC000	2080768
Day of Month	Fourth Week	0000 1111 1110 0000 0000 0000 0000 0000	0FE00000	266338304
Day of Month	Odd Days	0101 0101 0101 0101 0101 0101 0101 0101	55555555	1431655765
Day of Month	Even Days	0010 1010 1010 1010 1010 1010 1010 1010	2AAAAAAA	715827882
Month	No Months	0000 0000 0000 0000	0000	0
Month	All Months	0000 1111 1111 1111	0FFF	4095

Examples

To perform some action if the time of the second Schedule (number 2) is due:

```
SetIntIBSystemIO("Schedule Number", 2);
once time = GetIntIBSystemIO("Schedule Time") then
begin
  { do something here }
end;
```

Note that it is generally a better idea to have the Schedule perform an action directly if possible (for example, controlling C-Bus or executing a [Special Function](#)) or alternatively, having the Schedule enabling a logic module which then performs some actions, then disables itself. See [Controlling Modules from Components or Schedules](#) for details.

To change the time the second Schedule is due to 7:00PM without disrupting the selection of the Schedule on the user interface:

```
OriginalSchedule := GetIntIBSystemIO("Schedule Number"); { record for later }
SetIntIBSystemIO("Schedule Number", 2);
SetIntIBSystemIO("Schedule Time", "7:00PM");
SetIntIBSystemIO("Schedule Number", OriginalSchedule); { set selected Schedule
back to what it was }
```

To change the third Schedule to be due on odd days of the month:

```
SetIntIBSystemIO("Schedule Number", 3);
SetIntIBSystemIO("Schedule Day of Month Mask", $55555555);
```

4.17.4 System IO Examples

Example 1

To increment the value of an integer System IO variable called "Door Count" when the Lighting Group Address "Door Press" goes on, the following methods could be used :

```
once GetLightingLevel("Door Press") then
begin
  { count is an integer variable }
  count := GetIntSystemIO("Door Count") + 1;
  SetIntSystemIO("Door Count", count);
end;
```

OR

```
once GetLightingLevel("Door Press") then
  SetIntSystemIO("Door Count", GetIntSystemIO("Door Count") + 1);
```

OR

```
once GetLightingLevel("Door Press") then
  SetIntSystemIO("Door Count", succ(GetIntSystemIO("Door Count")));
```

Example 2

System IO Variables "Schedule Time" and "Schedule Date" are used to set when a relay (Group Address 1) is to be switched on. One possible solution to do this would be :

```
once (Time = GetIntSystemIO("Schedule Time")) and (Date = GetIntSystemIO
("Schedule Date")) then
begin
  SetLightingLevel(1, 100%, 0);
end;
```

Example 3

Normally a C-Bus Thermostat would be used to control an HVAC system. If you had a very simple requirement for a heating system, you could use:

- A heater controlled by a relay (group address 5)
- A C-Bus temperature sensor, broadcasting on the C-Bus HVAC Application (Zone Group 1, unswitched zone)
- A user System IO variable ("Set Point") to allow the user to set the temperature

The code could look something like this:

```

{ constants section }
HeatingGA = 5;

{ var section }
SetPoint, Temp : real;
Heating : boolean;

{ modules }
SetPoint := GetRealSystemIO("Set Point");
Temp := GetRealIBSystemIO("HVAC Temperature", 1, HVACZoneU);
Heating := GetLightingState(HeatingGA);

if (Temp >= SetPoint + 1) and Heating then
    SetLightingState(HeatingGA, off);
if (Temp <= SetPoint - 1) and (not Heating) then
    SetLightingState(HeatingGA, on);

```

Note that there is ± 1 degree of "hysteresis" so that the heater doesn't switch off and on too rapidly.

Example 4

To get the time of the first Schedule for use with logic:

```

{ var section }
ScheduleTime : integer;

{ modules }
SetIntIBSystemIO("Schedule Number", 1);
ScheduleTime := GetIntIBSystemIO("Schedule Time");

```

4.17.5 Tutorial 5

Question 1

What are the differences between a System I/O variable and a regular Pascal variable ?

Question 2

An Integer System I/O variable called "Counter" has been defined. What is wrong with the following statements :

1. if "Counter" = 10 then ...
2. "Counter" := 0;

Question 3

Write some code using a timer to switch off the "Bathroom Light" if it has been on for more than 30 minutes.

Question 4

Write a statement to set the "Switch On Time" System IO variable to the time that the "Spa Pump"

Group Address was switched on.

[Tutorial Answers](#)

4.18 Special Days

Special Days are used for providing more flexibility for Schedules and Access Control. Special Days are created using the PICED Special Day Manager.

Typical uses of Special Days are to enable the user to perform specific actions on Special Days like public holidays or school holidays.

Special Day Types

Each day has a particular Special Day type. The Special Day type indicates whether the day is a Public Holiday or one of several user-defined Special Day types.

The Special Day types are [Integers](#) as shown below. Special Days also have [Tags](#) which are the names allocated using the Special Day Manager.

Special Day Type Value	Meaning	Default Tag
1	Normal day	"Normal"
2	Public Holiday	"Public Holiday"
4	User Defined 1	"Special Day 1"
8	User Defined 2	"Special Day 2"
16	User Defined 3	"Special Day 3"
32	User Defined 4	"Special Day 4"
64	User Defined 5	"Special Day 5"
128	User Defined 6	"Special Day 6"

A given date may have more than one Special Day type. For example, it may be both a public holiday and a school holiday. In this case, the Special Day Type value for the day will be the sum of the values in the table above. So if Special Day 1 was defined as being "School Holidays", then a day that was a Public Holiday during school holidays will be both a Public Holiday and a School Holiday. In this case the day's Special Day Type value will be 6 (Public Holiday (2) + school holiday (4) = 6).

There are two functions related to Special Days :

- [IsSpecialDayType Function](#)
- [SpecialDayType Function](#)

4.18.1 IsSpecialDayType Function

The IsSpecialDayType function returns whether the [Special Day](#) Type of a given date is a particular value.

Applicability

Colour C-Touch, Black & White C-Touch and PAC only.

Syntax

IsSpecialDayType(d, SpecialDayType)

d is an [Integer](#) or [Date Tag](#)

SpecialDayType is an Integer or [Special Day Tag](#)

Description

The IsSpecialDayType function returns whether the [Special Day Type](#) of the date d is of type SpecialDayType. The result is a [Boolean](#) value (true or false).

[Tags](#) can also be used when referring to the SpecialDayType.

Example

To perform an action if today is a public holiday :

```
if IsSpecialDayType(date, 2) then ...
```

OR

```
if IsSpecialDayType(date, "Public Holiday") then ...
```

To perform an action if today is a Special Day 1 or a Special Day 2 :

```
if IsSpecialDayType(date, 12) then ... { 4 + 8 = 12 }
```

OR

```
if IsSpecialDayType(date, "Special Day 1" + "Special Day 2") then ...
```

OR

```
if IsSpecialDayType(date, "Special Day 1" or "Special Day 2") then ...
```

4.18.2 SpecialDayType Function

The SpecialDayType function returns the [Special Day](#) Type of a given date.

Applicability

Colour C-Touch, Black & White C-Touch and PAC only.

Syntax

```
SpecialDayType(d)
```

d is an [Integer](#) or [Date Tag](#)

Description

The SpecialDayType function returns the [Special Day Type](#) of the [Date](#) d. The Special Day Type returned is an integer.

[Tags](#) can also be used when referring to the value returned by the SpecialDayType function.

Example

If Special Day 1 corresponds to school holidays and 25th December 2004 has been defined as both a public holiday and a Special Day 1 (school holiday) then :

```
SpecialDayType("25 Dec 2004")
```

will return a value of 6 (Public Holiday (2) + school holiday (4) = 6).

To determine if a particular date is a particular Special Day type, use the [IsSpecialDayType Function](#). If you want to determine if today is a Public Holiday, do **NOT** use :

```
if SpecialDayType(date) = 2 then ... { this will not work }
```

OR

```
if SpecialDayType(date) = "Public Holiday" then ... { this will not work }
```

because these will not work if the given date is of more than one Special Day type.

Alternatively, you can use [Bitwise Operators](#). For the above example, you could use :

```
if (SpecialDayType(date) and 2) <> 0 then ...
```

OR

```
if (SpecialDayType(date) and "Public Holiday") <> 0 then ...
```

4.19 String Functions



The following functions can be used for the manipulation of [Strings](#) :

- [Append Procedure](#)
- [Copy Procedure](#)
- [DateToString Procedure](#)
- [Format Procedure](#)
- [Length Function](#)
- [LowerCase Procedure](#)
- [Pos Function](#)
- [SetLength Procedure](#)
- [StringToInt Function](#)
- [StringToReal Function](#)
- [TimeToString Procedure](#)
- [UpperCase Procedure](#)
- [IntToHexString Procedure](#)
- [HexStringToInt Function](#)
- [StringToUTF8 Procedure](#)
- [UTF8ToString Procedure](#)

4.19.1 Append Procedure

The append procedure concatenates two [strings](#).

Syntax

```
append(string1, string2);
```

string1 is a string variable

string2 is a string or char

Description

String2 is added onto the end of string1.

Example

The code :

```
string1 := 'abc';
append(string1, '123');
```

results in string1 becoming 'abc123'.

4.19.2 Copy Procedure

The copy procedure extracts a substring of a [string](#).

Syntax

```
Copy(string1, string2, Index, Count);
```

string1 is a string variable
string2 is a string
Index and Count are integers

Description

Copy extracts a substring containing Count characters starting at character number Index, and stores it in String1.

If Index is larger than the length of String2, Copy returns an empty string. If Count specifies more characters than are available, only the characters from number Index to the end of String2 are returned.

Example

The code :

```
Copy(string1, 'abcdefgh', 3, 2);
```

results in string1 becoming 'cd'.

4.19.3 DateToString Procedure

The DateToString procedure converts a [Date](#) to a [string](#).

Syntax

```
DateToString(date1, string1);
```

date1 is an [integer](#)
string1 is a [string](#) variable

Description

DateToString converts the date1 parameter to a string using the Windows date format and stores it in String1.

Example

The code :

```
DateToString(35065, string1);
```

results in string1 becoming something like '1/1/1996' (depending on the date format selected from the Windows Control Panel).

The code :

```
DateToString(date, string1);
```

results in string1 becoming the current date.

4.19.4 Format Procedure

The format procedure produces a [string](#) containing a list of values of parameters.

Syntax

```
Format(string, argument list);
```

string is a string variable

argument list is a list of arguments (integer, real, boolean, char or string)

Description

The format function operates in a similar way to the [WriteLn](#) function, except that the result is stored in a string. The argument list uses the same format as the WriteLn procedure.

Example

The code :

```
i := 5;
Format(string1, 'result =', i:3);
```

results in string1 becoming 'result = 5'.

4.19.5 Length Function

The length function returns an integer which represents the length (number of characters) of the [string](#).

Syntax

```
length(String1)
```

Description

Length returns the number of characters actually used in the string.

Example

```
length('abcdef') equals 6
```

To perform an action if the length of string s is zero :

```
if length(s) = 0 then ...
```

See also [SetLength Procedure](#).

4.19.6 LowerCase Procedure

The LowerCase procedure converts a [string](#) to lower case.

Syntax

```
LowerCase(string1);
```

string1 is a string variable

Description

LowerCase changes String1 so that all letters are converted to lower case. The conversion affects only [ASCII](#) characters between 'A' and 'Z'.

Example

The code :

```
String1 := 'ABCdef';  
LowerCase(string1);
```

results in string1 becoming 'abcdef'.

See also [UpperCase Procedure](#)

4.19.7 Pos Function

The pos function returns the position of a specified substring that occurs in a given [string](#).

Syntax

```
pos(Substr, S)
```

Where

Substr is a [string](#) expression and is the string to be found
s is a string expression and is the string being searched

Description

Pos searches for a substring, Substr, in a string, S.

Pos searches for Substr within S and returns an integer value that is the index of the first character of Substr within S. Pos is case-sensitive. If Substr is not found, Pos returns zero.

Example

```
pos('de', 'abcdefgh') equals 4
```

To perform an action if the string 'get' is in the string s :

```
if pos('get', s) <> 0 then ...
```

See also [Pos2 Function](#)

4.19.8 Pos2 Function

The pos2 function returns the position of the specified substring that occurs in a given [string](#).

Syntax

```
pos2(Substr, S, n)
```

Where

Substr is a [string](#) expression and is the string to be found
s is a string expression and is the string being searched
n is an integer expression and is the starting position for the search

Description

Pos2 searches for a substring, Substr, in a string, S starting from position n.

Pos2 searches for Substr within S, starting from position n, and returns an integer value that is the index of the first character of Substr within S. Pos2 is case-sensitive. If Substr is not found, Pos2 returns zero.

Note

pos(substr, s) is equivalent to pos2(substr, s, 1).

Example

```
pos2('de', 'abcdefghabcdefgh', 2) equals 4
pos2('de', 'abcdefghabcdefgh', 6) equals 12
```

See also [Pos Function](#)

4.19.9 SetLength Procedure

The SetLength procedure sets the length of a [string](#).

Syntax

```
SetLength(string1, NewLength);
```

string1 is a [string](#) variable
NewLength is an integer

Description

SetLength simply sets the [length](#)-indicator character (the character at string1[0]) to the given value. In this case, NewLength must be a value between 0 and the maximum length of the string. Existing characters in the string or elements in the array are preserved, but the content of newly allocated space is undefined.

Note that this does not affect the maximum length of a string - this is set by the string declaration.

Example

The code :

```
String1 := 'abcdefghij';
SetLength(string1, 3);
```

results in string1 becoming 'abc'.

The code :

```
String1 := 'abcdefghij';
SetLength(string1, 20);
```

results in the start of string1 becoming 'abcdefghij', but the next 10 characters are undefined, and could be anything.

4.19.10 StringToInt Function

The StringToInt function converts a [string](#) to an [Integer](#).

Syntax

```
StringToInt(string1)
```

string1 is a [string](#) expression

Description

StringToInt converts the string1 parameter to an integer value. String is a string-type expression; it must be a sequence of characters that form a signed integer number. If the string is not a valid number, then the result of the function will be zero.

To convert an integer to a string, use the [Format Procedure](#).

Example

The code :

```
int1 := StringToInt('23');
```

results in int1 becoming 23.

See also [StringToIntDef Function](#)

4.19.11 StringToIntDef Function

The StringToIntDef function converts a [string](#) to an [Integer](#).

Syntax

```
StringToIntDef(string1, default)
```

string1 is a [string](#) expression
default is an integer

Description

StringToIntDef converts the string1 parameter to an integer value. String is a string-type expression; it must be a sequence of characters that form a signed integer number. If the string is not a valid number, then the result of the function will be the default value.

To convert an integer to a string, use the [Format Procedure](#).

Example

The code :

```
int1 := StringToInt('23', -1);
```

results in int1 becoming 23.

The code :

```
int1 := StringToInt('ABC', -1);
```

results in int1 becoming -1.

See also [StringToInt Function](#)

4.19.12 StringToReal Function

The StringToReal function converts a [string](#) to a [Real](#) number.

Syntax

```
StringToReal(string1)
```

string1 is a [string](#) expression

Description

StringToReal converts the string1 parameter to a real value. String is a string-type expression; it must be a sequence of characters that form a signed real number. If the string is not a valid number, then the result of the function will be zero.

To convert a real number to a string, use the [Format Procedure](#).

Example

The code :

```
real1 := StringToReal('23.45');
```

results in real1 becoming 23.45

4.19.13 TimeToString Procedure

The TimeToString procedure converts a [Time](#) to a [string](#).

Syntax

```
TimeToString(time1, string1);
```

time1 is an [integer](#)

string1 is a [string](#) variable

Description

TimeToString converts the time1 parameter to a string using the Windows time format and stores it in String1.

Example

The code :

```
TimeToString(3600, string1);
```

results in string1 becoming something like '1:00:00AM' (depending on the time format selected from the Windows Control Panel).

The code :

```
TimeToString(time, string1);
```

results in string1 becoming the current time.

See also [DurationToString Procedure](#)

4.19.14 DurationToString Procedure

The DurationToString procedure converts a [Time](#) to a [string](#).

Applicability

Colour C-Touch only.

Syntax

```
DurationToString(time1, string1);
```

time1 is an [integer](#)

string1 is a [string](#) variable

Description

DurationToString converts the time1 parameter to a string and stores it in String1. It does not use any AM/PM indication and hence is useful for showing the text for a duration or elapsed time, rather than the current time. The Windows settings are used for the separators between the hours, minutes and seconds.

Example

The code :

```
TimeToString(3600, string1);
```

results in string1 becoming something like '1:00:00' (depending on the time format selected from the Windows Control Panel).

The code :

```
TimeToString(TimerTime(1), string1);
```

results in string1 becoming the time of [Timer](#) 1.

See also [TimeToString Procedure](#)

4.19.15 UpperCase Procedure

The UpperCase procedure converts a [string](#) to upper case.

Syntax

```
UpperCase(string1);
```

string1 is a string variable

Description

UpperCase changes String1 so that all letters are converted to upper case. The conversion affects only [ASCII](#) characters between 'a' and 'z'.

Example

The code :

```
String1 := 'ABCdef';  
UpperCase(string1);
```

results in string1 becoming 'ABCDEF'.

See also [LowerCase Procedure](#)

4.19.16 IntToHexString Procedure

The IntToHexString procedure converts an [integer](#) to a [Hexadecimal](#) number [string](#).

Syntax

```
IntToHexString(number, precision, string1);
```

number is an [integer](#)
precision is an integer
string1 is a [string](#) variable

Description

IntToHexString converts the number parameter to a string.

The precision parameter is the number of digits in the result. If there are more digits in the precision than in the number, then the string will be padded with 0s at the front. A precision of 0 makes the string exactly the right length. A negative precision, or one greater than 12, will result in a value of 12 being used for the precision.

Negative numbers are represented as two's complement numbers.

Example

The code :

```
IntToHexString(255, 4, string1);
```

results in string1 becoming '00FF'.

The code :

```
IntToHexString(255, 0, string1);
```

results in string1 becoming 'FF'.

See also [HexStringToInt Function](#)

4.19.17 HexStringToInt Function

The HexStringToInt function converts a [string](#) containing a [Hexadecimal Number](#) to an integer.

Syntax

```
HexStringToInt(string1)
```

string1 is a [string](#) variable

Description

HexStringToInt converts the hexadecimal number in the string parameter to an integer. If the string does not contain a valid hexadecimal number, the result will be 0.

The maximum value is limited by the range of [Integers](#) (2147483647 or 7FFFFFFF hexadecimal).

Example

The code :

```
n := HexStringToInt('FF');
```

results in n becoming 255.

See also [IntToHexString Procedure](#)

4.19.18 StringToUTF8 Procedure

The StringToUTF8 procedure converts a normal ([Unicode](#)) string to a [UTF-8](#) encoded string.

Applicability

Colour C-Touch only.

Syntax

```
StringToUTF8(string1)
```

string1 is a [string](#) variable

Description

StringToUTF8 converts the string to the UTF-8 format for use with writing to [files](#), [serial ports](#) and [sockets](#).

See [UTF-8 Example](#)

4.19.19 UTF8ToString Procedure

The UTF8ToString procedure converts a [string](#) from [UTF-8](#) encoding to a normal ([Unicode](#)) string.

Applicability

Colour C-Touch only.

Syntax

```
UTF8ToString(string1)
```

string1 is a [string](#) variable

Description

UTF8ToString converts the string from the UTF-8 format for use with reading from [files](#), [serial ports](#) and [sockets](#).

See [UTF-8 Example](#)

4.20 Other Functions

The following functions are related to the use of the Logic Engine, and are not standard Pascal functions :

- [Beep Procedure](#)
- [CurrentPage Function](#)
- [Execute Procedure](#)
- [GetAccessLevel Function](#)
- [LevelToPercent Function](#)
- [LogMessage Procedure](#)
- [PercentToLevel Function](#)
- [ShowPage Procedure](#)
- [ShowingPage Function](#)
- [Execute Special Function Procedure](#)

4.20.1 Beep Procedure

The Beep procedure makes a default beep sound.

Applicability

Colour C-Touch and Black & White C-Touch only.

Syntax

```
Beep;
```

Note : the [Execute Procedure](#) can be used to play particular sound files.

4.20.2 CurrentPage Function

The CurrentPage function returns the number of which page is currently showing.

Applicability

Colour C-Touch and Black & White C-Touch only.

Syntax

```
CurrentPage
```

Description

The CurrentPage function returns an integer which is page the PICED software is currently displaying. This is particularly useful for [Graphics](#). Note that the first page number is [0, not 1](#).

Example

To perform an action if the PICED page has just changed :

```
if CurrentPage <> OldPage then
begin
  OldPage := CurrentPage;
  ...
end;
```

See also [ShowingPage Function](#)

4.20.3 Execute Procedure

The Execute procedure is used to play a sound file.

Applicability

Colour C-Touch only.

Syntax

```
Execute(Command, CommandParameters);
```

Command is a string

CommandParameters is a string (unused)

Description

In HomeGate and Schedule Plus, the Execute procedure is used to run a program or open a file. For Colour C-Touch, this can only be used to play a sound (WAV) file. The function arguments are :

- Command : This is the file name of the WAV file
- CommandParameters : This is unused and should be an empty string (").

Example

To Play a sound file "test.wav" which is located in the project folder (the folder containing the current project):

```
Execute('test.wav', '');
```

 Note that all files used in logic (including WAV files) need to be included with the project archive when transferred to a Colour C-Touch. See the main help file topic Exporting to an Archive.

4.20.4 GetAccessLevel Function

The GetAccessLevel function returns the current user access level.

Applicability

Colour C-Touch and Black & White C-Touch only.

Syntax

```
GetAccessLevel
```

Description

GetAccessLevel returns the current user access level. The lowest level is index 0.

Example

To store the current user access level in the variable AccessLevel :

```
AccessLevel := GetAccessLevel;
```

4.20.5 LevelToPercent Function

The LevelToPercent function converts a C-Bus level to a percent.

Syntax

```
LevelToPercent(level)
```

Description

The LevelToPercent function converts a C-Bus level (0 to 255) to a percent (0 to 100). See percent

for more details.

Example

To store the percent equivalent of the variable Level in the variable PC :

```
PC := LevelToPercent(Level);
```

See also [PercentToLevel Function](#)

4.20.6 LogMessage Procedure

The LogMessage procedure writes a message to the log.

Applicability

Colour C-Touch only.

Syntax

```
LogMessage(s);
```

s is a [string](#)

Description

The LogMessage procedure writes the string s to the PICED log and to the Logic [Output Window](#). This is useful for [Debugging Programs](#).

The [WriteLn Procedure](#) will also write data to the log if the **Send WriteLn output to Log** option is selected in the [Logic Engine Options](#).

Example

To write the string 'start' to the log :

```
LogMessage('start');
```

4.20.7 ShowPage Procedure

The ShowPage procedure shows a PICED page.

Applicability

Colour C-Touch and Black & White C-Touch only.

Syntax

```
ShowPage(p);
```

p is an [Integer](#) or page name [Tag](#)

Description

The ShowPage procedure shows the selected PICED page. Note that the first page number is [0, not 1](#).

Example

To display a PICED page called "warning" :

```
ShowPage("warning");
```

See also [ShowingPage Function](#), [ShowSubPage Procedure](#)

4.20.8 ShowingPage Function

The ShowingPage function returns whether a page is showing.

Applicability

Colour C-Touch and Black & White C-Touch only.

Syntax

```
ShowingPage(p)
```

p is an [Integer](#) or page name [Tag](#)

Description

The ShowingPage function returns whether the selected PICED page is currently being displayed. This is particularly useful for [Graphics](#). Note that the first page number is [0, not 1](#).

Example

To perform an action if the PICED page called "warning" is being displayed :

```
if ShowingPage("warning") then ...
```

See also [CurrentPage Function](#), [ShowingSubPage Function](#)

4.20.9 Halt Statement

The Halt statement stops the Logic Engine from [Running](#).

Syntax

```
Halt
```

Example

To stop the Logic Engine if a variable called Counter reaches a value of 10 :

```
if Counter = 10 then
  Halt;
```

To stop the Logic Engine and then restart it, see the [Restart Statement](#).

4.20.10 Restart Statement

The Restart statement stops the Logic Engine from [Running](#) and then restarts it.

Syntax

```
Restart
```

Example

To stop the Logic Engine and then restart it if a boolean system IO variable called Restart is true :

```
if GetBoolSystemIO("Restart") then
  Restart;
```

To stop the Logic Engine without restarting, see the [Halt Statement](#).

4.20.11 Tutorial 6

In all of the following questions, variables called String1 and String2 are string variables.

Question 1

Write a statement to assign String1 with the first 3 characters of String2.

Question 2

Write some code to assign String1 with the text "Date = " followed by the current date.

Question 3

String1 contains some text containing the text "Level" followed by three characters containing an integer. Extract the integer and assign it to a variable called x.

Question 4

Write a statement to play a wav file called "HomeTime.wav" at 5:30PM every day.

Question 5

A System IO variable called "Desired Level" contains a C-Bus level in %. A boolean System IO variable called "Set Now" is used to set the level when it is set to true. Write some code to set the "Lounge Light" to the level of the "Desired Level" when "Set Now" is true.

Question 6

Write a statement to set the "All On" Scene at 8:30AM on weekdays (Monday to Friday) which are not a public holiday.

[Tutorial Answers](#)

4.20.12 PercentToLevel Function

The PercentToLevel function converts a C-Bus level to a percent.

Syntax

```
PercentToLevel(level)
```

Description

The PercentToLevel function converts a percent (0 to 100) to a C-Bus level (0 to 255). See percent for more details.

Example

To store the percent equivalent of the variable PC in the variable Level :

```
Level := PercentToLevel(PC);
```

See also [LevelToPercent Function](#)

4.20.13 ExecuteSpecialFunction Procedure

The ExecuteSpecialFunction procedure executes a Special Function.

Applicability

Colour C-Touch and Black & White C-Touch only.

Syntax

```
ExecuteSpecialFunction(SpecialFunction, Parameter);
```

SpecialFunction is a Special Function [Tag](#)

Parameter is an Integer or String. It is only needed for some special functions as described below.

Description

The ExecuteSpecialFunction procedure executes a Special Function. Not all Special Functions are available in logic. Some Special Functions require a Parameter as show below :

Special Function	Parameter
Irrigation, Run Zone	Irrigation Zone number
Labels, Set Language	Label language number
Page Transition	Page Transition tag or number
Security Keypad	ASCII value of key
Telephony Divert	Phone Number (string)
Media Transport Control	Media Link Group
<i>All others</i>	<i>Ignored (leave as 0)</i>

Example

To switch on the alarm when a [User System IO Variable](#) called "Alarm Trigger" becomes true :

```
once GetBoolSystemIO("Alarm Trigger") = ON then
  ExecuteSpecialFunction("Alarm On", 0);
```

To have Media Link Group 1 change to the next track:

```
ExecuteSpecialFunction("Media Transport Control Next Track", 1);
```

To transmit an emulation of a security keypad "1" key press (ASCII 49 = "1"):

```
ExecuteSpecialFunction("Security Keypad", 49);
```

4.21 C-Bus Unit Functions

The following functions are only applicable to C-Bus Units (PAC, C-Touch Mark II and Wiser Home Control) :

- [SetLEDState Procedure](#)
- [ToggleLEDState Procedure](#)
- [IsPAC Function](#)
- [IsCTouch Function](#)
- [IsCBusUnit Function](#)
- [IsWiser Function](#)

The C-Bus Units do not support :

- [Sets](#)
- [System IO](#)
- [Special Days](#)
- [Graphics](#)
- [Socket \(TCP/IP\) IO](#)
- [Files](#)
- [Beep Procedure](#)
- [Execute Procedure](#)
- [GetAccessLevel Function](#)
- [LogMessage Procedure](#)
- [ShowPage Procedure](#)
- [ShowingPage Function](#)

The C-Bus Unit firmware will be updated from time to time, so additional functions may be available in the future.

4.21.1 SetLEDState Procedure

The SetLEDState procedure sets the state of the PAC User LED.

Applicability

Black & White C-Touch and PAC only.

Syntax

```
SetLEDState(state);
```

Description

The SetLEDState procedure sets the state of the PAC User LED to that of the state parameter. If state = true the User LED is on. The User LED can be used for debugging purposes.

When the PICED software is running, the PAC LED state is shown on the status bar of the main form (it is shown as a green circle if the LED is on).

Example

To set the User LED to be the same as the state of the Kitchen Light :

```
SetLEDState(GetLightingState("Kitchen"));
```

See also [ToggleLEDState Procedure](#)

4.21.2 ToggleLEDState Procedure

The ToggleLEDState procedure toggles the state of the PAC User LED.

Applicability

Black & White C-Touch and PAC only.

Syntax

```
ToggleLEDState(state);
```

Description

The ToggleLEDState procedure toggles (swaps) the state of the PAC User LED. If the User LED is was on it goes off and vice versa. The User LED can be used for debugging purposes.

When the PICED software is running, the PAC LED state is shown on the status bar of the main form (it is shown as a green circle if the LED is on).

Example

To toggle the User LED each time the Counter gets to 10 :

```
if Counter = 10 then
begin
  ToggleLEDState;
  Counter := 0;
end;
```

See also [SetLEDState Procedure](#)

4.21.3 IsPAC Function

The IsPAC function returns whether the logic is running in a PAC.

Syntax

```
IsPAC
```

Description

The IsPAC function is used to determine whether the logic is running on a PAC. It returns a [Boolean](#) result. This is usually used to select various parts of code depending on whether the code is running on the computer or in the PAC.

Example

To execute some code only if the logic is running in the PAC :

```
if IsPAC then
begin
  ...
end;
```

4.21.4 IsCTouch Function

The IsCTouch function returns whether the logic is running in a C-Touch unit.

Syntax

```
IsCTouch
```

Description

The IsCTouch function is used to determine whether the logic is running on a C-Touch. It returns a [Boolean](#) result. This is usually used to select various parts of code depending on whether the code is running on the computer or in the C-Touch.

Example

To execute some code only if the logic is running in the C-Touch :

```
if IsCTouch then
begin
...
end;
```

4.21.5 IsCBusUnit Function

The IsCBusUnit function returns whether the logic is running in a C-Bus Unit.

Syntax

```
IsCBusUnit
```

Description

The IsCBusUnit function is used to determine whether the logic is running in a C-Bus Unit (PAC, Black and White C-Touch, Colour C-Touch, C-Touch Spectrum or Wisier). It returns a [Boolean](#) result which has a value of true if running in a C-Bus unit. This is usually used to select various parts of code depending on whether the code is running on the computer or in the C-Bus Unit.

Example

To open a serial port only if the code is running in a C-Bus unit:

```
if IsCBusUnit then
begin
OpenSerial(1, 1, 9600, 8, scOneStopBit, scNoFlowControl, scNoParity);
end;
```

To execute some debugging code only if the logic is running on the computer :

```
if not IsCBusUnit then
begin
WriteLn('Group Address 30 = ', GetLightingLevel(30));
end;
```

See also [IsCTouch Function](#), [IsPAC Function](#), [IsWiser Function](#)

4.21.6 IsWiser Function

The IsWiser function returns whether the logic is running in a Wisier Home Control unit.

Syntax

```
IsWiser
```

Description

The IsWiser function is used to determine whether the logic is running on a Wiser Home Control. It returns a [Boolean](#) result. This is usually used to select various parts of code depending on whether the code is running on the computer or in the Wiser Home Control.

Example

To execute some code only if the logic is running in the Wiser Home Control :

```
if IsWiser then
begin
...
end;
```

4.21.7 IsMasterUnit Function

The IsMasterUnit function returns whether the unit is the Master Unit.

Syntax

```
IsMasterUnit
```

Description

The IsMasterUnit function is used to determine whether the unit is the Master Unit. It returns a [Boolean](#) result.

The Master Unit [In-Built System IO Variable](#) can also be used to read and set the Master Unit setting.

Example

To execute some code only if the unit is the Master Unit :

```
if IsMasterUnit then
begin
...
end;
```

4.22 Flow Control

Unless directed otherwise, the statements in a program will be executed in the order in which they appear. Sometimes there is a need to not execute some statements, or to execute statements several times.

Looping means repeating a statement or compound statement over and over until some condition is met.

There are three types of loops:

- [fixed repetition](#) - only repeats a fixed number of times
- [pretest](#) - tests a Boolean expression, then goes into the loop if TRUE
- [posttest](#) - executes the loop, then tests the Boolean expression

The [IF THEN](#) statement is used to execute statements under specific circumstances. There is also a [ONCE](#) statement and [HasChanged](#) and [ConditionStaysTrue](#) functions, which are not standard Pascal, but were incorporated in the Logic Engine because it simplifies many common tasks for automation.

[Modules](#) can also be used to control which parts of a program will be executed.

4.22.1 If Statement

The IF statement allows you to perform an action based on the result of a Boolean condition. There are two forms of if statement: if...then and the if...then...else. The syntax of an if...then statement is

```
if condition then statement
```

where condition returns a [Boolean](#) value. If condition is True, then statement is executed; otherwise it is not. For example,

```
if J <> 0 then
  Result := I/J;
```

In this example, the result will only be calculated if J is not equal to zero.

The syntax of an if...then...else statement is

```
if condition then statement1 else statement2
```

where condition returns a Boolean value. If condition is True, then statement1 is executed; otherwise statement2 is executed. For example,

```
if J = 0 then
  WriteLn('error')
else
  Result := I/J;
```

The then and else clauses contain one statement each, but it can be a [compound statement](#). For example,

```
if J <> 0 then
begin
  Result := I/J;
  Count := Count + 1;
end
else if Count = Last then
  Done := True;
```

Notice that there is never a semicolon between the then clause and the word else. You can place a semicolon after an entire if statement to separate it from the next statement in its block, but the then and else clauses require nothing more than a space or carriage return between them. Placing a semicolon immediately before else (in an if statement) is a common programming error.

A special difficulty arises in connection with nested if statements. The problem arises because some if statements have else clauses while others do not, but the syntax for the two kinds of statement is otherwise the same. In a series of nested conditionals where there are fewer else clauses than if statements, it may not seem clear which else clauses are bound to which ifs. Consider a statement of the form

```
if condition1 then if condition2 then statement1 else statement2;
```

There would appear to be two ways to interpret this:

```
if condition1 then [ if condition2 then statement1 else statement2 ];
```

```
if condition1 then [ if condition2 then statement1 ] else statement2;
```

The compiler always interprets it the first way. Hence the statement

```
if ... { condition1 } then
  if ... { condition2 } then
    ... { statement1 }
  else
    ... { statement2 } ;
```

is equivalent to

```
if ... { condition1 } then
begin
  if ... { condition2 } then
    ... { statement1 }
  else
    ... { statement2 }
end;
```

The rule is that nested conditionals are parsed starting from the innermost conditional, with each else bound to the nearest available if on its left. To force the compiler to read our example in the second way, you would have to write it explicitly as

```
if ... { condition1 } then
begin
  if ... { condition2 } then
    ... { statement1 }
  end
else
  ... { statement2 } ;
```

There is no harm in explicitly placing the begin and end statements, even if they are not required. At least there is less chance of making an error.

Optimisation of IF Statements

With a complex If / Then statement, such as :

```
if condition1 and condition2 and condition3 and condition4 and condition5
then ...
```

all of the conditions are evaluated each [scan](#). This could also be written as :

```
if condition1 then
  if condition2 then
    if condition3 then
      if condition4 and condition5 then ...
```

In this case, only condition1 is evaluated each scan. If it is true, then condition2 will be evaluated and so on. This results in code which executes much quicker, as there is less work being done each scan. The trade-off is that the code may be a little less readable. It also can not be done where OR logic is used.

The code can be optimised further by having condition1 as something that is simple to evaluate (such as time = "9:00PM") rather than something more time consuming (such as MyComplexFunction = 23). Ideally, condition1 should be something that is only true occasionally,

so that condition2 etc rarely get evaluated. For example, if condition1 was (time = "9:00PM"), then it will only be true for one second per day, and hence the other conditions will only be evaluated once per day.

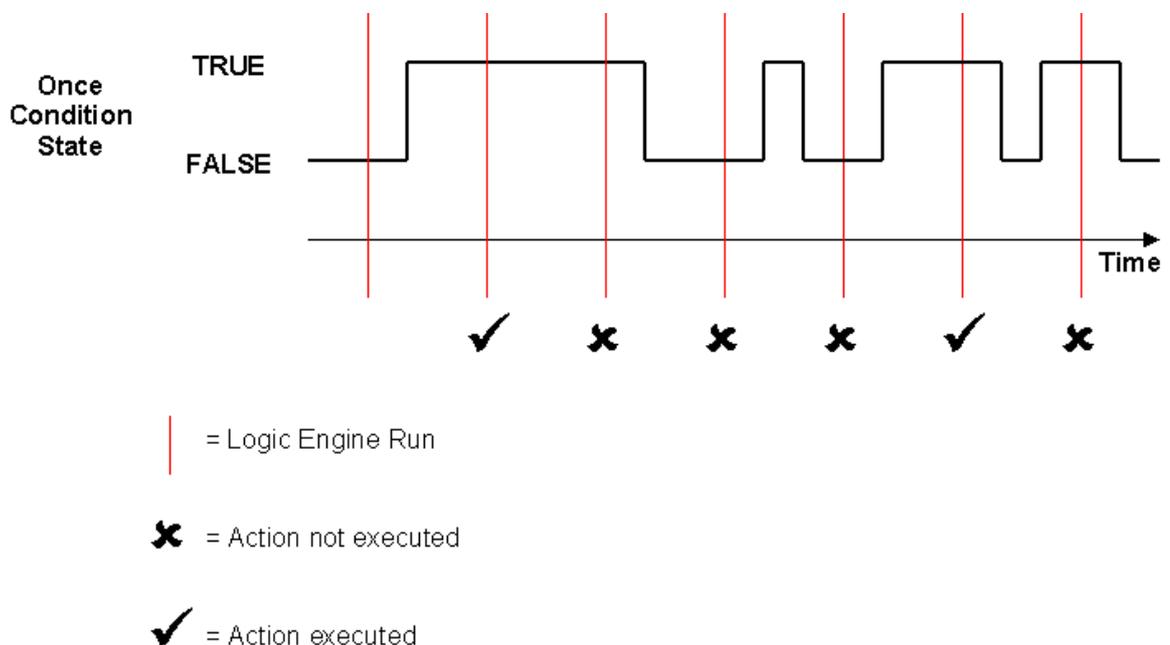
See also [When to use if and once](#)

4.22.2 Once Statement

The format of the Once statement is :

```
once condition then
  statement;
```

The Once statement is similar to the [IF THEN](#) statement, except that the statement is only executed on the scan when the condition first becomes true. The condition needs to go false, then true again for the statement to be executed again as shown in the diagram below :



Note that there can not be an "else" clause as there can with an if statement.

If the condition is true when the Logic Engine first runs, the statement will not be executed. The condition needs to change from false to true in order for the statement to be executed.

Examples

If you want to switch on a lamp when the lounge light goes on, you could do :

```
if GetLightingState("Lounge") then
  SetLightingState("Lamp", ON);
```

but the statement switching on the lamp would be executed every time the [Module](#) was run, resulting in the lamp being switched on repeatedly while the Lounge light was on. This is obviously not a good thing to do.

A better approach is to do :

```
once GetLightingState("Lounge") then
    SetLightingState("Lamp", ON);
```

In this case, the statement switching on the lamp would be executed only when the Lounge light was first switched on. Until the Lounge light is switched off, then on again, the lamp will not be switched on again.

See also [When to use if and once](#) and [HasChanged Function](#)

4.22.3 When to use if and once

The [if](#) and [once](#) statements do have different purposes, and are usually not interchangeable.

The **once** statement is used when you want an action when a condition first becomes true. The **if** statement is used when you want an action every [scan](#) when the condition is true.

Each time the **once** statement is evaluated, the Logic Engine looks at the state of the condition. If the condition is TRUE and it was FALSE the last time it was executed then it will execute the statement. The consequences of this are that there are several situations where a once statement may not do what you may expect :

- A once statement will never execute on the first scan, even if the condition is true (because the previous state is unknown).
- A once statement should never be used inside a loop ([repeat](#), [while](#), [for](#)) - see below
- A once statement should never be used inside an if or once statement - see below
- A once statement should never be used inside a [Function](#) or [Procedure](#)

Once Inside a Loop

Do not use a once statement inside a for loop, such as :

```
for Group := 10 to 15 do
    once GetLightingState(Group) = ON then { don't do this }
        SetLightingLevel("Corridor", ON);
```

In the above example, the intention is to switch on the Corridor light when any of Group Address 10 to 15 first go on. The problem is that each time the condition is evaluated, the Group variable has changed, and so it is not comparing the current value of the Group Address with what it was previously. It is actually comparing each Group Address with the previous one.

In the above example, suppose Group Addresses 10, and 12 are off and the others are on, and that they have been at these levels for some time. On the first loop, Group = 10 and the once condition is false. On the second loop, Group = 11 and the condition is true. The once condition has gone from false to true, and so the statement will be executed even though none of the Group levels have changed. On the third loop, Group = 12 and the condition is false again. On the fourth loop, Group = 13 and the condition is true and once again the statement will be executed. On the fifth loop, Group = 14 and the condition is true, but since it was true previously, then the statement will not be executed.

This could be made to work if the code was re-written as :

```
once GetLightingState(10) or GetLightingState(11) or GetLightingState(12) or
GetLightingState(13) or
    GetLightingState(14) or GetLightingState(15) then
    SetLightingLevel("Corridor", ON);
```

or if there was a scene called "office lights" containing Group Addresses 10 to 15, you could use :

```
once GetSceneMaxLevel("office lights") > 0 then
  SetLightingLevel("Corridor", ON);
```

Once Inside if

The problem with having a **once** statement inside an **if** statement is similar. Do not write code like :

```
if GetLightingState("Partition") = ON then
begin
  once GetLightingState("Conference Room 1") = ON then
    SetLightingLevel("Conference Room 2", ON);
  once GetLightingState("Conference Room 1") = OFF then
    SetLightingLevel("Conference Room 2", OFF);
  once GetLightingState("Conference Room 2") = ON then
    SetLightingLevel("Conference Room 1", ON);
  once GetLightingState("Conference Room 2") = OFF then
    SetLightingLevel("Conference Room 1", OFF);
end;
```

In this example there is a conference room with a moveable partition. There is a light in both sides of the room. There is a sensor attached to the partition which switches on a Group Address when the partition is open. When the partition is closed, the lights operate independently. If the partition is open, the lights need to operate together.

When the partition is open or closed, the code operates as expected. When the partition first opens ("Partition" group goes on), there is a problem. In this case, if any of the once conditions are true, the Logic Engine will look at the value the last time the condition was evaluated. The problem is that the last time the conditions were evaluated was when the partition was first closed, which may have been days ago.

Consider the case where Conference Room 1 light is off and then the partition is closed. The once conditions will not be evaluated while the partition is closed. If the light is now switched on and some time later the partition is opened, the once condition will be evaluated again and it will be found to have changed from false to true and hence switch on Conference Room 2 light even though light 1 has actually been on for some time.

There are several ways of overcoming this problem. The easiest is to put the if statement inside the once statement :

```
once GetLightingState("Conference Room 1") = ON then
  if GetLightingState("Partition") = ON then
    SetLightingLevel("Conference Room 2", ON);
once GetLightingState("Conference Room 1") = OFF then
  if GetLightingState("Partition") = ON then
    SetLightingLevel("Conference Room 2", OFF);
once GetLightingState("Conference Room 2") = ON then
  if GetLightingState("Partition") = ON then
    SetLightingLevel("Conference Room 1", ON);
once GetLightingState("Conference Room 2") = OFF then
  if GetLightingState("Partition") = ON then
    SetLightingLevel("Conference Room 1", OFF);
```

There is still one bug here, which is quite minor. When light 1 goes on, it will switch light 2 on (if the partition is open). The third once statement will see light 2 going on and it will switch light 1 on again. Since only one extra message is generated, it is not really worthwhile worrying about, but it can be fixed quite easily. This is left to the reader as an exercise :-)

See also [Handling Triggers](#)

4.22.4 ConditionStaysTrue Function

The ConditionStaysTrue function returns whether a condition has stayed true for a certain time.

Syntax

```
ConditionStaysTrue(condition, duration)
```

condition is a [Boolean](#) expression

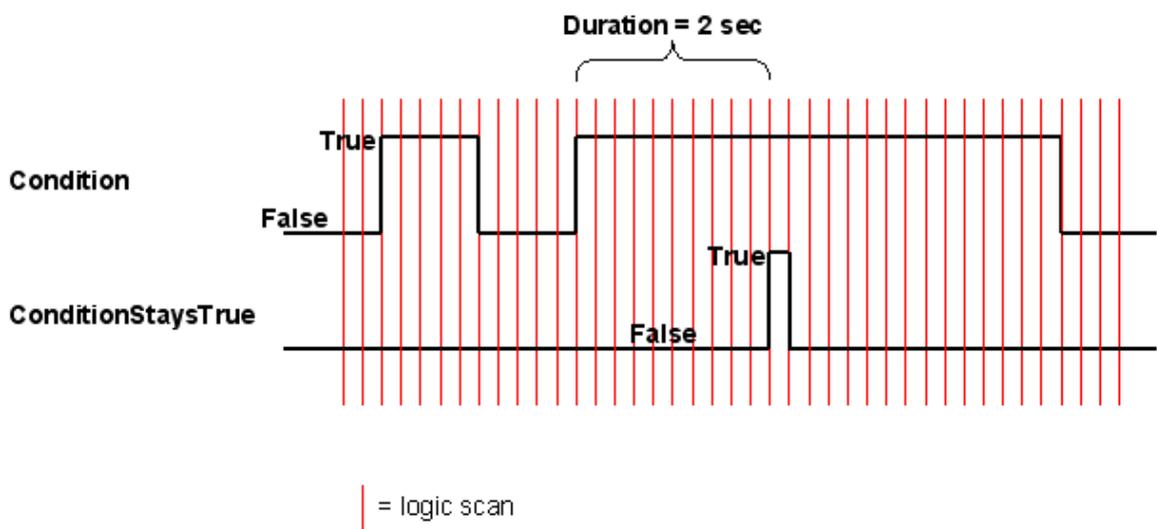
duration is an [Integer](#) which is the delay time in seconds.

Description

The ConditionStaysTrue function is used to perform an action after a certain condition has been true and stayed true for a certain amount of time. It is often used in place of a [Delay](#).

The ConditionStaysTrue function result is true when the condition has gone true and stayed true for the duration period. It will only be true on the [scan](#) when the duration is complete. After that it will return false again. The condition will need to go false again then stay true again for the duration before the ConditionStaysTrue function result will go true again.

In the example below, a condition is being tested to see if it stays true for more than 2 seconds. The ConditionStaysTrue function will only be true on the scan when the duration has been 2 seconds. Even though the condition stays true for another 2 seconds, the ConditionStaysTrue function does not go true again because the condition has not returned to false.



Notes

It does not matter whether the ConditionStaysTrue gets used with an [If Statement](#) or a [Once Statement](#), but "if" is preferred.

Like a once statement, the ConditionStaysTrue function should not be used [inside a loop](#).

There is a limit of 50 (PAC or C-Touch Mark 2) or 250 (Colour C-Touch) ConditionStaysTrue functions in a program.

Example

If you wanted to switch on a fan if a light has been on for one minute, you could use the code :

```
if ConditionStaysTrue(GetLightingState("light") = ON, 60) then
    SetLightingState("fan", ON);
```

This is different from the following code :

```
if GetLightingState("light") = ON then
begin
    delay(60:
    SetLightingState("fan", ON);
end;
```

There are two differences :

1. In the second case, the fan goes on after 60 seconds, even if the light goes off again before 60 seconds has finished
2. In the second case, the module stops while the delay happens. In the first case, the rest of the module is still running.

If you wanted the execution of the module to pause while the condition stayed true, you could use code like :

```
StartTime := RunTime;
WaitUntil((GetLightingState("light") = OFF) or (RunTime - StartTime >= 60));
SetLightingState("fan", ON);
```

4.22.5 HasChanged Function

The HasChanged function returns whether a value has changed.

Syntax

```
HasChanged(value)
```

value is an [Integer](#), [Real](#) or [Boolean](#) expression

Description

The HasChanged function returns whether an integer, real or boolean value has changed since the last time the value was evaluated (generally, the last scan). It is used to perform an action when something changes.

Notes

It generally does not matter whether the HasChanged gets used with an [If Statement](#) or a [Once Statement](#), but using "if" is preferred.

Like a once statement, the HasChanged function should not be used [inside a loop](#).

There is a limit of 50 (PAC or C-Touch Mark 2) or 250 (Colour C-Touch) HasChanged functions in a program.

Examples

To perform some action when the "Lounge" light changes level:

```
if HasChanged(GetLightingLevel("Lounge")) then...
```

To set the "Outside" light to be the same as the "Control" [User System IO Variable](#) each time it changes:

```

if HasChanged(GetIntSystemIO("Control")) then
begin
  SetLightingLevel("Outside", GetIntSystemIO("Control"), 0);
end;

```

See also [Once Statement](#)

4.22.6 Case Statement

The case statement provides a simpler alternative to complex [if](#) statements. A case statement has the form

```

case SelectorExpression of
  CaseList1: statement1;
  ...
  CaseListn: statementn;
end;

```

where SelectorExpression is any expression of an [ordinal type](#) (string types are invalid). Each CaseList is a numeral, declared constant, or other expression that the compiler can evaluate without executing your program. It must be of an ordinal type compatible with SelectorExpression. Thus the constants 7, True and 'A' can all be used in Case Lists, but variables and function calls cannot.

When a case statement is executed, at most one of its constituent statements is executed. Whichever CaseList has a value equal to that of SelectorExpression determines the statement to be used.

The case statement allows you to rewrite code which uses a lot of if else statements, making the program logic much easier to read. Consider the following code portion written using if else statements (operator is a [Char Type](#)):

```

if      operator = '*' then result := number1 * number2
else if operator = '/' then result := number1 / number2
else if operator = '+' then result := number1 + number2
else if operator = '-' then result := number1 - number2;

```

Rewriting this using case statements results in much clearer code :

```

case operator of
  '*' : result:= number1 * number2;
  '/' : result:= number1 / number2;
  '+' : result:= number1 + number2;
  '-' : result:= number1 - number2;
end;

```

The value of operator is compared against each of the values specified. If a match occurs, then the program statement(s) associated with that match are executed.

It is possible to group cases as shown below :

```

case user_request of
  'A', 'a' : call_addition_subprogram;
  'S', 's' : call_subtraction_subprogram;
end;

```

A second form of the case statement contains an "else" statement:

```
case SelectorExpression of
  CaseList1: statement1;
  ...
  CaseListn: statementn;
else
  ElseStatement;
end;
```

If none of the CaseLists has the same value as SelectorExpression, then the ElseStatement will be executed.

For example, with the code:

```
case a of
  1 : b := 2;
  2 : b := 5;
  5 : b := 10;
else
  b := 1;
end;
```

If a is less than 1, equals 3 or 4 or is greater than 5, then b will be assigned a value of 1.

4.22.7 When to use if and case

Similar things can be achieved using the [If Statement](#) and the [Case Statement](#). If you wanted to use a lighting level to select between doing three different things, you could write code like :

```
if GetLightingLevel("select") = 0 then
begin
  { do first thing }
end;
if GetLightingLevel("select") = 1 then
begin
  { do second thing }
end;
if GetLightingLevel("select") = 2 then
begin
  { do third thing }
end;
```

The problem with this code is that each of the statements gets executed on every [scan](#), even though only one can be true at a time (ie they are mutually exclusive). A better way to write the code would be :

```
if GetLightingLevel("select") = 0 then
begin
  { do first thing }
end
else if GetLightingLevel("select") = 1 then
begin
  { do second thing }
end
else if GetLightingLevel("select") = 2 then
begin
  { do third thing }
```

```
end;
```

In this case, the second if statement is only evaluated if the first one is false. Similarly, the third is only evaluated if the first two are false. It is still a little inefficient in that the `GetLightingLevel` needs to be evaluated up to 3 times. This could be improved as follows :

```
Select := GetLightingLevel("select");
if Select = 0 then
begin
  { do first thing }
end
else if Select = 1 then
begin
  { do second thing }
end
else if Select = 2 then
begin
  { do third thing }
end;
```

An alternative way is to use a [Case Statement](#) as follows :

```
case GetLightingLevel("select") of
  0 : begin
    { do first thing }
    end;
  1 : begin
    { do second thing }
    end;
  2 : begin
    { do third thing }
    end;
end;
```

In general, if you are selecting between 4 or more values, you should use a case statement. If you are selecting between 2 values, always use an if statement. For 3 values, you could choose either.

4.22.8 Repeat Statement

The syntax of a repeat statement is

```
repeat
  statement1;
  ...
  statementn;
until expression;
```

where expression returns a Boolean value. The last semicolon before "until" is optional. The repeat statement executes its sequence of constituent statements continually, testing expression after each iteration. When expression returns True, the repeat statement terminates. The sequence is always executed at least once because expression is not evaluated until after the first iteration.

There is no need to use the begin/end keywords to group more than one program statement, as all statements between repeat and until are treated as a block.

This loop is also called a posttest loop because the condition is tested AFTER the body of the loop executes. The REPEAT loop is useful when you want the loop to execute at least once, no matter

what the starting value of the Boolean expression is, whereas the [while](#) loop statements may not get executed at all.

4.22.9 While Statement

A while statement is similar to a [repeat](#) statement, except that the control condition is evaluated before the first execution of the statement sequence. Hence, if the condition is false, the statement sequence is never executed.

The syntax of a while statement is

```
while expression do
  statement;
```

where expression returns a Boolean value and statement can be a compound statement (ie starting with "begin" and ending with "end;"). The while statement executes its constituent statement repeatedly, testing expression before each iteration. As long as expression returns True, execution continues.

The WHILE ... DO loop is also called a pretest loop because the condition is tested before the body of the loop executes. So if the condition starts out as FALSE, the body of the while loop never executes.

4.22.10 For Statement

A for statement, unlike a repeat or while statement, requires you to specify explicitly the number of iterations you want the loop to go through. The syntax of a for statement is

```
for counter := InitialValue to FinalValue do
  statement;
```

or

```
for counter := InitialValue downto FinalValue do
  statement;
```

where

counter is a local variable (declared in the block containing the for statement) of [ordinal](#) type, without any qualifiers.

InitialValue and FinalValue are expressions that are assignment-compatible with counter.

statement is a simple or structured statement that does not change the value of counter.

The for statement initially assigns the value of InitialValue to counter, then executes statement repeatedly, incrementing or decrementing counter after each iteration. (The for...to syntax increments counter, while the for...downto syntax decrements it.) When counter returns the same value as FinalValue, statement is executed once more and the for statement terminates. In other words, statement is executed once for every value in the range from InitialValue to FinalValue. If InitialValue is equal to FinalValue, statement is executed exactly once. If InitialValue is greater than FinalValue in a for...to statement, or less than FinalValue in a for...downto statement, then statement is never executed. After the for statement terminates, the value of counter is undefined.

For purposes of controlling execution of the loop, the expressions InitialValue and FinalValue are evaluated only once, before the loop begins. Hence the for...to statement is almost, but not quite, equivalent to this while construction:

```

begin
  counter := initialValue;
  while counter <= finalValue do
  begin
    statement;
    counter := Succ(counter);
  end;
end

```

The difference between this construction and the for...to statement is that the while loop re-evaluates FinalValue before each iteration. This can result in noticeably slower performance if FinalValue is a complex expression, and it also means that changes to the value of FinalValue within statement can affect execution of the loop.

You can use the counter in calculations within the body of the loop, but you should not change the value of the counter. An example of using the counter is:

```

sum := 0;
for count := 1 to 100 do
  sum := sum + count;

```

A for loop can occur within another, so that the inner loop (which contains a block of statements) is repeated by the outer loop. These are called "nested" loops. With nested loops :

- 1 Each loop must use a separate variable
- 2 The inner loop must begin and end entirely within the outer loop.

Example

To find the maximum value in an [array](#) called Data :

```

Max := 0;
for I := 1 to 100 do
  if Data[I] > Max then
    Max := Data[I];

```

To count the number of Group Addresses in the range 10 to 30 which are on :

```

Count := 0;
for i := 10 to 30 do
  if GetLightingState(i) then
    Count := Count + 1;

```

4.22.11 Tutorial 7

Question 1

Write a program to display the first 10 triangle numbers. Triangle numbers are formed as follows :

- The first is 1
- The second is 1 + 2 = 3
- The third is 1 + 2 + 3 = 4
- etc

Question 2

Write a program to display the larger of two variables A and B.

Question 3

a, b, c and d are integers. What is displayed when the following is run ?

```

a := 5;
b := 3;
c := 99;
d := 5;
if a > 6 then writeln('A');
if a > b then writeln('B');
if b = c then
begin
  writeln('C');
  writeln('D')
end;
if b <> c then
  writeln('E')
else
  writeln('F');
if a >= c then
  writeln('G')
else
  writeln('H');
if a <= d then
begin
  writeln('I');
  writeln('J')
end;
end;

```

Question 4

a, b and c are integers. What is displayed when the following is run ?

```

a := 5;
b := 3;
c := 99;
if (a = 5) or (b > 2) then writeln('A');
if (a < 5) and (b > 2) then writeln('B');
if (a = 5) and (b = 2) then writeln('C');
if (c <> 6) or (b > 10) then
  writeln('D')
else
  writeln('E');
if (b = 3) and (c = 99) then writeln('F');
if (a = 1) or (b = 2) then writeln('G');
if not( (a < 5) and (b > 2) ) then writeln('H');

```

Question 5

Write a Pascal statement to compare the character variable "Letter" to the character constant 'A', and if less, prints the text string "Too low", otherwise print the text string "Too high"

Question 6

What will be displayed when the following is run ("i" is an integer) ?

```

i := 1;
repeat
  i := i * 2;
  WriteLn(i);
until i > 100;

```

Question 7

Convert the following statements to use case statements :

```
if i = 0 then WriteLn('A')
else if i = 1 then WriteLn('B')
else if i = 2 then WriteLn('C')
else if i = 3 then WriteLn('D');
```

Question 8

Write some code which counts how many times the "Bathroom Light" group address has been switched on.

Question 9

Write some code to get the "Lounge Lamp" to go on and off when the "Lounge Light" goes on and off.

Question 10

Write some code to control the "Bedroom Light" group from the "Bedroom Switch" group following these rules :

- The Bedroom Switch is an on/off (toggle) switch
- If the time is after 9PM, then switch the bedroom lights to 70% over 4 seconds when the switch goes on
- Otherwise switch the bedroom lights to 100% when the switch goes on
- Switch the bedroom lights off when the switch goes off

Question 11

Write some code to implement the following requirements :

A conference room has a moveable room divider with a sensor controlling the "Divider Closed" group. A switch on the wall of room 1 controls the "Room 1 Switch" group. When the divider is closed, toggling the room 1 switch should set the "Room 1 Off" and "Room 1 On" scenes. When the divider is open, toggling the room 1 switch should set the "All Off" and "All On" scenes.

Question 12

Write some code to implement the following requirements :

When the outside PIR sensor on group "Outside PIR" is triggered between 9PM and midnight, the lights in the three rooms "Room 1", "Room 2" and "Room 3" are to be switched on with a delay of two seconds between each, to make it look like someone is home. When the PIR goes off, the lights are to be returned to the levels they were initially at.

Question 13

Write some code to implement the following requirements :

A house requires a "lived in" look. The lights in four rooms "Room 1", "Room 2" "Room 3" and "Room 4" are to be switched at random times subject to the following rules :

- They are only to be on after sunset + 1 hour and before 11PM
- There must always be one light on
- Only one light may be on at a time
- The lights must change every 5 – 20 minutes at random
- The "lived in" look is to be enabled by a group called "away mode", set by the security system

Question 14

The code :

```

if time = "8:00PM" then
  SetScene("Night");

```

is not good because the Logic Engine could be run several times during the second where the time is exactly 8PM, resulting in the Scene being set several times. Why is the following code OK :

```

if time = "8:00PM" then
begin
  SetScene("Night");
  Delay("3:00:00");
  SetScene("Bed Time");
end;

```

Question 15

What is wrong with the following lines of code (5 errors) :

```

if day = 14 and month = 7 and time = 9PM then
  SetLightingLevel(Kitchen Light, 100%)

```

Tutorial Answers

4.23 Sub-Programs



Procedures and functions (referred to collectively as Routines or Sub-Programs) are self-contained statement blocks that can be called from different locations in a program. A function is a routine which returns a value when it executes. A procedure is a routine which does not return a value.

Sub-programs are used to make code clearer, and to minimise having common bits of code written several times. There are two types of sub-programs :

- [Procedures](#)
- [Functions](#)

4.23.1 Procedures

A procedure is a subprogram. Subprograms help reduce the amount of redundancy in a program. Statements which appear several times in a program are often put into subprograms. Subprograms also facilitate top-down design.

A procedure declaration has a form similar to a [program](#) :

```

procedure ProcedureName(Parameter List);
{ Local Declarations }
begin
  { statements }
end;

```

where ProcedureName is any valid identifier

statements is a sequence of statements that execute when the procedure is called, and Parameter List and Local Declarations are optional (see below)

Parameter List

Most procedure and function headers include a [parameter](#) list. For example, in the header

```
procedure NumString(N: Integer; var S: string);
```

the parameter list is (N: Integer; var S: string).

If a routine does not take any parameters, omit the identifier list and the parentheses in its declaration:

```
procedure UpdateRecords;
begin
  ...
end;
```

Within the procedure or function body, the parameter names (N and S in the first example above) can be used as local variables. Do not re-declare the parameter names in the local declarations section of the procedure or function body.

Local Declarations

Within a procedure's statement block, you can use [variables](#) and other [identifiers](#) declared in the Local Declarations part of the procedure. You can also use the parameter names from the parameter list (like N and S in the example above). The parameter list defines a set of local variables, so don't try to re-declare the parameter names in the Local Declarations section. Finally, you can use any identifiers within whose [scope](#) the procedure declaration falls.

Example

Here is an example of a procedure which sets all of the Group Addresses between Group1 and Group2 to Level :

```
procedure SetLevels(Group1, Group2, Level : integer);
var
  i: integer;
begin
  for i := Group1 to Group2 do
    SetLightingLevel(i, Level);
  end;
```

In the above example, the parameter list contains three integer variables : Group1, Group2 and Level. These parameters are variables which can be used in the procedure. There is also another variable, i, which is declared for use in the procedure. This variable can not be used anywhere else (for example, other procedures or [Modules](#)).

Given this procedure declaration, you can call the SetLevels procedure to set all Groups between 10 and 30 to the new level 50% like this:

```
SetLevels(10, 30, 50%);
```

See also [Forward Declarations](#)

4.23.2 Parameters

Procedures may accept data (parameters) to work with when they are called. A parameter list is a sequence of parameter declarations separated by semicolons and enclosed in parentheses. Each declaration is a comma-delimited series of parameter names, followed by a colon and a type identifier. Parameter names must be valid identifiers. Any declaration can be preceded by the reserved word `var`. Examples:

```
(X, Y: Real)
(var S: string; X: Integer)
```

The parameter list specifies the number, order, and type of parameters that must be passed to the routine when it is called.

Value Parameters

Generally, when variables are passed to procedures, the procedures work with a copy of the original variable. The value of the original variables which are passed to the procedure are not changed. The copy that the procedure makes can be altered by the procedure, but this does not alter the value of the original. When procedures work with copies of variables, they are known as value parameters.

Consider the following code example,

```
procedure NoChange(letter : char; number : integer);
begin
  WriteLn(letter);
  WriteLn(number);
  letter := 'A';           {this does not alter MainLetter}
  number := 32;           {this does not alter MainNumber}
  WriteLn(letter);
  WriteLn(number)
end;

{ var section }
MainLetter : char;      {these variables known only from here on}
MainNumber : integer;

{ Module section }
mainletter := 'B';
mainnumber := 12;
WriteLn(MainLetter);
WriteLn(MainNumber);
NoChange(MainLetter, MainNumber);
WriteLn(MainLetter);
WriteLn(MainNumber)
```

In this case, the output of the code would be :

```
B      { written from Module }
12     { written from Module }
B      { written from NoChange }
12     { written from NoChange }
A      { written from NoChange }
32     { written from NoChange }
B      { written from Module }
12     { written from Module }
```

Variable parameters

Procedures can also be implemented to change the value of variables which are accepted by the procedure. To illustrate this, we will develop a little procedure called swap. This procedure accepts two integer values, swapping them over.

Procedures which accept value parameters cannot do this, as they only work with a copy of the original values. To force the procedure to use variable parameters, precede the declaration of the variables (inside the parenthesis after the function name) with the keyword var. This has the effect of using the original variables, rather than a copy of them.

```

procedure SWAP (var value1, value2 : integer);
var
  temp : integer;
begin
  temp := value1;
  value1 := value2; { value1 is actually number1 }
  value2 := temp   { value2 is actually number2 }
end;

{ var section }
number1, number2 : integer;

{ module section }
number1 := 10;
number2 := 33;
WriteLn('Number1 = ', number1, ' Number2 = ', number2);
SWAP( number1, number2 );
WriteLn('Number1 = ', number1, ' Number2 = ', number2)

```

When this program is run, it prints out

```

Number1 = 10  Number2 = 33
Number1 = 33  Number2 = 10

```

Another example of the use of variable parameters is to return a [String Type](#) result. For example, the following code returns a string of characters:

```

procedure StringOfChar(n : integer; c: char; var s : String);
var
  i : integer;
begin
  s := '';
  for i := 1 to n do
    Append(s, c);
  end;
end;

```

The following code will result in string MyString being set to 'xxxxx':

```

StringOfChar(5, 'x', MyString);

```

4.23.3 Functions

Functions work the same way as [procedures](#), but they always return a value to the where they were called from, separate from the variables passed to the function. A function declaration is like a procedure declaration except that it specifies a return type. Function declarations have the form :

```

function FunctionName(ParameterList): ReturnType;
LocalDeclarations;
begin
    statements
end;

```

where FunctionName is any valid identifier
statements is a sequence of statements that execute when the function is called, and
ParameterList and LocalDeclarations are optional (see below)
Return**Type** is the [Type](#) of the result returned by the function

The function's statement block is governed by the same rules that apply to procedures. Within the statement block, you can use variables and other identifiers declared in the LocalDeclarations part of the function, parameter names from the parameter list, and any identifiers within whose scope the function declaration falls. In addition, the function name itself acts as a special variable that holds the function's return value.

For example,

```

function WF: Integer;
begin
    WF := 17;
end;

```

defines a function called WF that takes no parameters and always returns the integer value 17.

Here is a more complicated function declaration which returns the maximum of three parameters :

```

function Max(A, B, C: Integer): integer;
begin
    Max := A;
    if B > Max then
        Max := B;
    if C > Max then
        Max := C;
end;

```

You can assign a value to the function name repeatedly within a statement block, as long as you assign only values that match the declared return type. When execution of the function terminates, whatever value was last assigned to the function name becomes the function's return value.

If execution terminates without an assignment being made to the function name, then the function's return value is undefined.

If it is necessary for a function to return more than one value, then [variable parameters](#) need to be used to modify variables from the calling [block](#) of code.

Note that functions can not return a [String Type](#) result. A string value can be set using a [Variable Parameter](#) in a [Procedure](#).

4.23.4 Blocks

A block consists of a series of declarations followed by a [compound statement](#). All declarations must occur together at the beginning of the block. So the form of a block is

```

declarations
begin

```

```

    statements
end

```

The declarations section can include, in any order, declarations for [variables](#), [constants](#), [types](#), [procedures](#), [functions](#). For example, in a function declaration like :

```

function Convert(const S: string): string;
var
    Ch: Char;
    L: Integer;
    Source, Dest: PChar;
begin
    ...
end;

```

the first line of the declaration is the function heading and all of the succeeding lines make up the block. Ch, L, Source, and Dest are local variables; their declarations apply only to the Convert function block and override (in this block only) any declarations of the same identifiers that may occur in the program block.

4.23.5 Scope

A [Block](#) ([program](#), [procedure](#) or [function](#)) can declare its own variables to work with. These variables belong to the procedure in which they are declared. Where these variables can be used is called their "scope".

An identifier, such as a variable or function name, can be used only within the scope of its declaration. The location of a declaration determines its scope. An identifier declared within the declaration of a program, function, or procedure has a scope limited to the [block](#) in which it is declared. Identifiers with narrower scope (especially identifiers declared in functions and procedures) are sometimes called local, while identifiers with wider scope are called global.

If the identifier is declared in the declaration of a program, function, or procedure, its scope extends from the point where it is declared to the end of the current block, including all blocks enclosed within that scope.

When one block encloses another, the former is called the outer block and the latter the inner block. If an identifier declared in an outer block is re-declared in an inner block, the inner declaration overrides the outer one and determines the meaning of the identifier for the duration of the inner block. For example, if you declare a global variable called MaxValue in the program, and then declare another variable with the same name in a function declaration within that program, any occurrences of MaxValue in the function block are governed by the second, local declaration. Similarly, a function declared within another function creates a new, inner scope in which identifiers used by the outer function can be re-declared locally.

Example

Consider the program :

```

{ var section }
i, j : integer;

procedure Test;
var
    i : integer;
begin
    i := 23;

```

```

    WriteLn(i + j);
end;

{ Module section }
i := 3;
j := 4;
Test;
WriteLn(i + j);

```

When this program is run, it will write the value 27 then 7. There are two occurrences of the variable `i`. There is a global variable `j` which is available to the main program and to the `Test` procedure. The global variable `i` is available to the main program, but not to the `Test` procedure, since it has a local variable called `i`. Within the `Test` procedure, any reference to the variable `i` will use the local one.

4.23.6 Recursion

Recursion is a difficult topic to grasp, and is generally not necessary for the purposes of automation control.

Recursion means allowing a function or procedure to call itself. The summation function (which adds all of the numbers between 1 and `n`) is a popular example of recursion:

```

function Summation (num : integer) : integer;
begin
    if num = 1 then
        Summation := 1
    else
        Summation := Summation(num-1) + num
    end;
end;

```

Suppose you call `Summation` with a value of 3 :

```
a := Summation(3);
```

- `Summation(3)` becomes `Summation(2) + 3`.
- `Summation(2)` becomes `Summation(1) + 2`.
- At `Summation(1)`, the recursion stops and the result is 1.
- `Summation(2)` becomes `1 + 2 = 3`.
- `Summation(3)` becomes `3 + 3 = 6`.
- `a` becomes 6.

Recursion works backward until a given point is reached at which an answer is defined, and then works forward with that definition, solving the other definitions which rely upon that one.

All recursive procedures/functions must have some sort of test so stop the recursion. Under one condition, called the base condition, the recursion should stop. Under all other conditions, the recursion should go deeper. In the example above, the base condition was `if num = 1`. If you don't build in a base condition, the recursion will either not take place at all, or continue indefinitely and cause a [Run Time Error](#).

4.23.7 Forward Declarations

In the [Scope](#) section, it was stated that procedures/functions can only see variables and other subprograms that have already been defined. There is an exception.

If you have two subprograms, each of which calls the other, you have a dilemma that no matter which you put first, the other still can't be called from the first.

To resolve this chicken-and-the-egg problem, use forward declarations :

```

procedure Later(parameter list); forward;

procedure Sooner(parameter list);
begin
    ...
    Later(parameter list);
end;

procedure Later(parameter list);
begin
    ...
    Sooner(parameter list);
end;

```

The same applies to functions. Just put the reserved word **forward** at the end of the declaration.

4.23.8 Tutorial 8

Question 1

Write a Pascal procedure called Multiply, which accepts two integers, number1 and number2, and prints the result of multiplying the two integers together

Question 2

What is the output of the following Pascal program

```

program Sample( output );
var x, y : integer;

procedure godoit( x, y : integer );
begin
    x := y; y := 0;
    writeln( x, y );
end;

begin
    x := 1;
    x := 2;
    godoit( x, y );
    writeln( x, y )
end.

```

Question 3

Write a Pascal function called Multiply2 which returns an integer result. The function accepts two integer parameters, number1 and number2 and returns the value of multiplying the two parameters

Question 4

What is displayed by the following code :

```

{ procedures }
procedure test1(x : integer);
begin
    x := x + 1;

```

```

    WriteLn(x);
end;

procedure test2(var x : integer);
begin
    x := x + 1;
    WriteLn(x);
end;
{ ... }
{ main program }
i := 2;
test1(i);
WriteLn(i);
test2(i);
WriteLn(i);

```

Question 5

What is wrong with this code (3 errors) :

```

begin
    x = x + 1;
end;

begin
    WriteLn('x = ' x);
end;

```

[Tutorial Answers](#)

4.24 Modules

Modules contain the main part of the user program. Modules separate the code in separate sections to :

- Place related parts of code together to make the code easier to understand
- Enable a section of code to be enabled or disabled
- Control which parts of the code gets suspended with a [Delay](#) or [WaitUntil](#) procedure

Modules are created with the [Logic Editor](#). When the program is [Compiled](#), a Pascal program is generated containing the code from each module "wrapped" in some code to implement the necessary features for Module operation. The Module code is also "wrapped" in a [Repeat](#) loop so that the Modules get executed on a [regular basis](#). The result is a Pascal program which looks a bit like :

```

program LogicEngine;
begin

    { Initialisation Code goes here }

repeat

    if ModuleEnabled("Module 1") then
    begin
        { Module 1 code goes here }
    end;

    if ModuleEnabled("Module 2") then
    begin
        { Module 2 code goes here }
    end;
end;

```

```
end;  
  
...  
  
WaitFor200ms;  
  
until LogicEngineRunTimeError;  
end.
```

The user does not have to get involved with the complexities of how Modules get enabled or disabled. This is all handled by the compiler.

To an extent, Modules create behaviour a bit like a multi-tasking / multi-process / multi-threaded system. The main difference being that each Module does not have its own memory - they all share [global Variables](#).

Although there are many benefits in modularizing your code, each additional Module does require extra processing due to the "wrapping" process described above. For example, creating hundreds of separate Modules, each with just a few lines of code would be very inefficient.

For each [scan](#), the Modules all get executed one after the other, in the order that they are listed in the [Logic Tree](#). Any Modules which are [Disabled](#) or are [Delayed](#) are skipped until they are [Enabled](#) again.

There are several functions used for controlling the behaviour of Modules :

- [Delay Procedure](#)
- [EnableModule Procedure](#)
- [DisableModule Procedure](#)
- [ExitModule Procedure](#)
- [ModuleDisabled Function](#)
- [ModuleEnabled Function](#)
- [ModuleWaiting Function](#)
- [WaitUntil Procedure](#)

Note that the [InitialisationCode](#) only gets executed once - when the program first runs.

See also Software Limits and [Program Execution](#)

4.24.1 Module Tags

Modules can be referred to by [Tags](#) which correspond to the name of the Module in the [Logic Editor](#). The tag will be the Module name enclosed within double quotes.

Example

```
"Pool Control"
```

4.24.2 Module Groups

Modules can be arranged into groups for convenience using the [Logic Editor](#). The Module Groups do not affect the generation of the code in any way, they are just there to allow related Modules to be grouped together so that they can be found easily.

4.24.3 Initialisation

The initialisation code is code which only gets executed when the logic engine is first run. It is used for initialising [variables](#) and any other actions which do not need to be executed for every [scan](#) of the

Logic Engine.

Note that when the Logic Engine is first run, all variables are initialised to the following values :

- [Integer Type](#) : 0
- [Real Type](#) : 0.0
- [Boolean Type](#) : false
- [Char Type](#) : NULL ([ASCII](#) 0)
- [String Type](#) : " (empty string)

For example, to set the initial value of the variable CounterValue to 1, the code in the initialisation section would be :

```
CounterValue := 1;
```

See also [Using Counters](#).

4.24.4 Delay Procedure

The Delay procedure suspends the current Module for a specified amount of time.

Syntax

```
Delay(t);
```

t is an [Real number](#) or [Time Tag](#)

Description

This suspends (pauses) the current Module for t seconds. At the end of this time, processing of the rest of the Module will continue. It has no effect on any other Modules.

Notes

Delays can only be used within [Modules](#), not within [Functions](#), [Procedures](#) or the [Initialisation](#) sections.

The minimum delay is 0.2 seconds.

See [Program Execution](#) for details of what happens during a delay.

Example

To delay for 0.2 seconds :

```
Delay(0.2);
```

To delay for 1 hour :

```
Delay("1:00:00");
```

To switch on the Porch Light, wait for 10 minutes and then switch it off again :

```
SetLightingGroup("Porch Light", on);  
Delay("0:10:00");  
SetLightingGroup("Porch Light", off);
```

In the above example, the Module in which this code exists will wait at the line with the delay until the delay is complete. When the delay is complete, the Module execution will continue from the line after the delay.

Note

A delay of zero will cause a [Run Time Error](#). Hence do not write code like this :

```
Delay(random("1:00:00"));
```

since the random function could return a value of zero. A simple way to solve this would be to have :

```
Delay(random("1:00:00") + 1);
```

See also [Random Event Times](#) and [Program Execution](#)

4.24.5 EnableModule Procedure

The EnableModule procedure enables the selected Module.

Syntax

```
EnableModule(ModuleNumber);
```

ModuleNumber is an integer or [Module Tag](#).

Description

This enables the selected Module. It has no effect on any other Modules. Note that the index of the first Module is [0, not 1](#).

If the selected Module occurs later in the sequence of Modules (see [Program Execution](#)), then the re-enabled Module will be executed on the current scan. If the Module occurs earlier in the sequence of Modules, then the re-enabled Module will be executed on the next scan. The re-enabled Module will run from the first line of code the next time it is run.

Enabling a Module will cancel any [WaitUntil](#) statements which are in progress (if the module is already enabled), but has no effect on [Delays](#).

Example

To enable Module number 1 (the second one in the list) :

```
EnableModule(1);
```

To enable the Module called "Pool Control" :

```
EnableModule("Pool Control");
```

See also [Program Execution](#)

4.24.6 ExitModule Procedure

The ExitModule procedure causes the Module code to be terminated.

Syntax

```
ExitModule;
```

Description

This causes the code execution to leave the current Module and go to the next Module in the list. Processing of the current Module will continue from the first line of the Module on the next [scan](#). It has no effect on any other Modules.

Example

To exit the current Module if the variable n is zero :

```
if n = 0 then ExitModule;
```

4.24.7 DisableModule Procedure

The DisableModule procedure disables the selected Module.

Syntax

```
DisableModule(ModuleNumber);
```

ModuleNumber is an integer or [Module Tag](#).

Description

This disables the selected Module. The Module will not be run again until the Module is re-enabled. It has no effect on any other Modules. If a Module disables itself using this command, the rest of the Module will be executed, but it will not be run again until re-enabled.

Note that the index of the first Module is [0, not 1](#).

Example

To disable Module number 1 (the second one in the list) :

```
DisableModule(1);
```

To disable the Module called "Pool Control"

```
DisableModule("Pool Control");
```

See also [Program Execution](#)

4.24.8 ModuleDisabled Function

The ModuleDisabled function returns whether the selected Module is disabled.

Syntax

```
ModuleDisabled(ModuleNumber)
```

ModuleNumber is an integer or [Module Tag](#).

Description

A module can be [enabled](#), disabled or [waiting](#). The ModuleDisabled function returns a boolean value with the enable state (true/false) of the selected Module. The result is true if the module is disabled. The result is false if it is [enabled](#) or waiting for a [Delay](#) or a [WaitUntil](#).

Note that the index of the first Module is [0, not 1](#).

Example

To assign the enable state of Module number 1 (the second one in the list) to a variable called State :

```
State := not ModuleDisabled(1);
```

To perform an action if the Module called "Pool Control" is disabled :

```
if ModuleDisabled("Pool Control") then ...
```

4.24.9 ModuleEnabled Function

The ModuleEnabled function returns whether the selected Module is enabled.

Syntax

```
ModuleEnabled(ModuleNumber)
```

ModuleNumber is an integer or [Module Tag](#).

Description

A module can be enabled, [disabled](#) or [waiting](#). The ModuleEnabled function returns a boolean value with the enable state (true/false) of the selected Module. The result is true if the module is enabled and running. The result is false if it is [disabled](#) or waiting for a [Delay](#) or a [WaitUntil](#).

Note that the index of the first Module is [0, not 1](#).

Example

To assign the enable state of Module number 1 (the second one in the list) to a variable called State :

```
State := ModuleEnabled(1);
```

To perform an action if the Module called "Pool Control" is enabled :

```
if ModuleEnabled("Pool Control") then ...
```

4.24.10 ModuleWaiting Function

The ModuleWaiting function returns whether the selected Module is waiting in a [Delay](#) or a [WaitUntil](#).

Syntax

```
ModuleWaiting(ModuleNumber)
```

ModuleNumber is an integer or [Module Tag](#).

Description

A module can be [enabled](#), [disabled](#) or waiting. The ModuleWaiting function returns a boolean value with the waiting state (true/false) of the selected Module. The result is true if the module is waiting for a [Delay](#) or a [WaitUntil](#). The result is false if it is [disabled](#) or [enabled](#).

Note that the index of the first Module is [0, not 1](#).

Example

To assign the waiting state of Module number 1 (the second one in the list) to a variable called State

```

:
    State := ModuleWaiting(1);

```

To perform an action if the Module called "Pool Control" is waiting :

```

    if ModuleWaiting("Pool Control") then ...

```

4.24.11 WaitUntil Procedure

The WaitUntil procedure suspends the current Module until a condition is true.

Syntax

```

WaitUntil(BooleanCondition);

```

BooleanCondition is a [Boolean](#) expression.

Description

This suspends the current Module (in the same way as a [Delay](#)) until the Boolean Condition is true. The Boolean Condition is evaluated each [scan](#). Once the condition is true, processing of the Module will continue. It has no effect on any other Modules.

Note that the WaitUntil procedure can only be used within [Modules](#), not within [Functions](#), [Procedures](#) or the [Initialisation](#) sections.

Example

To wait until the EnableFlag variable is true :

```

WaitUntil(EnableFlag);

```

To wait until the time is 9:00 AM :

```

WaitUntil(Time = "9:00:00");

```

To wait until the Kitchen light goes on :

```

WaitUntil(GetLightingState("Kitchen"));

```

See also [Program Execution](#)

4.24.12 Tutorial 9

Question 1

Write some code to switch on the "Porch Light" at 7:00 PM and off at 11:00PM. Write the code in three versions :

- Using a Delay procedure
- Using a WaitUntil procedure
- Using neither

Question 2

Write some code to disable "Module 2" when a variable called Counter reaches 100m and re-enable the Module when the counter drops below 50.

[Tutorial Answers](#)

4.25 Graphics



Most of the time, the standard PICED components can be used to provide the necessary visual information for the user. If needed, the Logic Engine can "draw" graphics over the top of the PICED image.

It is usually necessary to check which page is [Showing](#) before drawing any graphics.

When a graphic function is executed (with the exception of ClearScreen), it is added to the Graphics Commands List. This is a list of "things to draw" when PICED has completed drawing its images. The Graphics Commands List can be viewed in the [Logic Editor](#) to ensure that they are correct.

There are a series of functions for drawing graphics :

- [ClearScreen Procedure](#)
- [DrawImage Procedure](#)
- [DrawText Procedure](#)
- [DrawTextBlock Procedure](#)
- [Ellipse Procedure](#)
- [LineTo Procedure](#)
- [MoveTo Procedure](#)
- [Rectangle Procedure](#)
- [RoundRect Procedure](#)
- [SetBrushColor Procedure](#)
- [SetBrushStyle Procedure](#)
- [SetFontColor Procedure](#)
- [SetFontName Procedure](#)
- [SetFontSize Procedure](#)
- [SetFontStyle Procedure](#)
- [SetPenColor Procedure](#)
- [SetPenStyle Procedure](#)
- [SetPenWidth Procedure](#)
- [TextPos Procedure](#)

There are also functions for determining where the screen has been clicked :

- [GetClick Function](#)
- [GetClickX Function](#)
- [GetClickY Function](#)

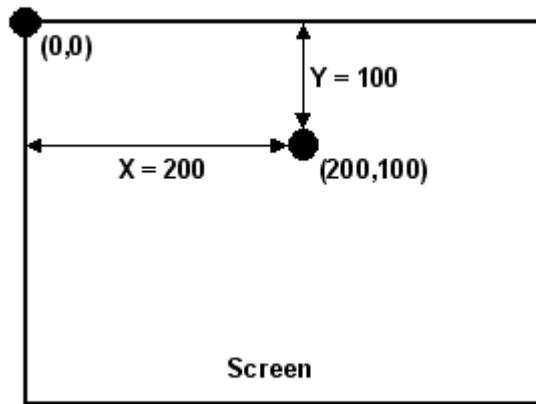
The following [Constants](#) can also be used for graphics :

- ScreenWidth - this is the width of the screen in pixels
- ScreenHeight - this is the height of the screen in pixels

4.25.1 Coordinates

The graphics routines use the top left corner of the window as the origin (0, 0). Hence increasing values for the vertical value are going down the screen.

Examples of screen coordinates are shown below :



4.25.2 Colours

The Colour of the [Brush](#) and the [Pen](#) can be set for use with the graphic procedures.

A Colour is specified as an [Integer](#), but for convenience, [Hexadecimal](#) notation is simplest to use. The colour is a 3-byte hexadecimal number, with the three bytes represent RGB colour intensities for blue, green, and red, respectively. Constants are defined for the common colours.

Examples

Value	Colour	Constant
\$FF0000	pure blue	clBlue
\$00FF00	pure green	clGreen
\$0000FF	pure red	clRed
\$000000	black	clBlack
\$FFFFFF	white	clWhite
\$FFFF00	cyan	clCyan
\$FF00FF	magenta	clMagenta
\$00FFFF	yellow	clYellow
\$808080	grey	clGray

4.25.3 ClearScreen Procedure

The ClearScreen procedure clears the [Graphics Commands List](#).

Applicability

Colour C-Touch only.

Syntax

```
ClearScreen;
```

Description

Any commands which are in the Graphics Commands List are deleted. The graphics properties are reset to their defaults :

- Pen [Colour](#) and [Style](#)
- Brush [Colour](#) and [Style](#)
- Font [Colour](#), [Name](#), [Size](#) and [Style](#)

4.25.4 DrawImage Procedure

The DrawImage procedure draws an image on the screen.

Applicability

Colour C-Touch only.

Syntax

```
DrawImage(X, Y, ImageNumber, Transparent);
```

X and Y are [Integers](#)

ImageNumber is an [Integer](#) or Image Name [Tag](#)

Transparent is a [Boolean](#) expression

Description

This draws image ImageNumber on the screen at a position (X, Y). If Transparent is true, then the image background will be transparent. In this case, the colour of the bottom left pixel of the bitmap (doesn't work for JPEGs) is used as the transparent colour. Only images used in the project can be used. The name of the image is used as a [Tag](#) to identify the image.

Note that this will not work with animated images.

Example

To draw the transparent image "light bulb.bmp" at coordinate (100, 200) on the screen :

```
DrawImage(100, 200, "light bulb.bmp", true);
```

In Colour C-Touch, there is only a limited amount of RAM (see the "Show Usage" feature of the log to find out how much). Attempting to use too many images may cause the Colour C-Touch to run out of memory. Note that a full-screen image will use nearly 1MB of memory, regardless of whether it is a JPEG, Bitmap or other file type.

4.25.5 DrawText Procedure

The DrawText procedure draws text on the screen.

Applicability

Colour C-Touch only.

Syntax

```
DrawText(Argument_List);
```

Argument_List is a list of expressions

Description

This writes text to the screen at a position set by the [TextPos](#) procedure. The format of the argument list is the same as for the [WriteLn](#) procedure. The font used can be set using the [SetFontColor](#), [SetFontName](#), [SetFontSize](#) and [SetFontStyle](#) procedures.

Example

To write the string 'hello' to the screen at coordinates 300,200 :

```
TextPos(300, 200);
DrawText('hello');
```

To write the value of variable LightLevel to the screen :

```
DrawText(LightLevel);
```

To write the value of variable Temperature with 1 decimal place to the screen :

```
DrawText(Temperature:4:1, 'C');
```

4.25.6 DrawTextBlock Procedure

The DrawTextBlock procedure draws text on the screen within a rectangular area.

Applicability

Colour C-Touch only.

Syntax

```
DrawTextBlock(Text, Left, Top, Right, Bottom, Alignment);
```

Text is a [String](#)

Left, Top, Right, Bottom and Alignment are [integers](#)

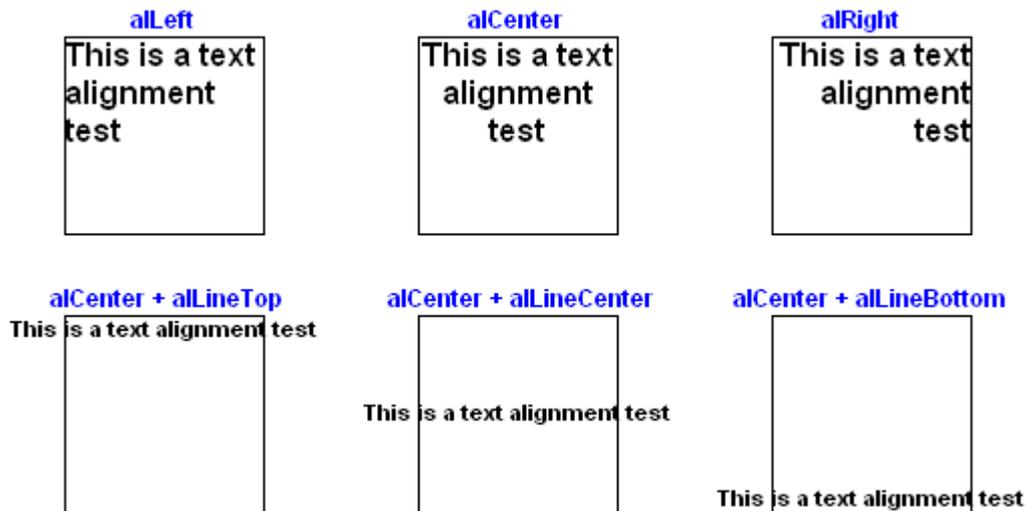
Description

This writes text to the screen within the area bounded by [points](#) (Left, Top) and (Right, Bottom). The font used can be set using the [SetFontColor](#), [SetFontName](#), [SetFontSize](#) and [SetFontStyle](#) procedures.

The alignment of the text within the rectangle is defined by the Alignment parameter, which can have the values shown in the table below.

Constant	Meaning
alLeft	Text is written in a block, aligned to the left
alCenter	Text is written in a block, aligned to the middle
alRight	Text is written in a block, aligned to the right
alLineTop	Text is written in a single line, aligned to the top
alLineCenter	Text is written in a single line, aligned to the middle
alLineBottom	Text is written in a single line, aligned to the bottom

A horizontal and a vertical constant can be combined. Any other values may give unpredictable results and should be avoided. Examples of results of the different alignments are shown below.



Examples

To write the text in variable `WarningMessage` in a block (lines "wrapped") with the text centre-aligned within the area :

- Left = 100
- Top = 100
- Right = 300
- Bottom = 200

```
TextBlock(WarningMessage, 100, 100, 300, 200, alCenter);
```

To write the text 'Middle' in the middle of the screen :

```
TextBlock('Middle', 0, 0, ScreenWidth, ScreenHeight, alCenter + alLineCenter);
```

See also [TextHeight Function](#), [TextWidth Function](#)

4.25.7 Ellipse Procedure

The Ellipse procedure draws an ellipse or circle on the screen.

Applicability

Colour C-Touch only.

Syntax

```
Ellipse(Left, Top, Bottom, Right);
```

Left, Top, Bottom and Right are [Integers](#)

Description

This draws the ellipse defined by a bounding rectangle on the screen. If the bounding rectangle is a square, a circle is drawn.

The ellipse is outlined using the value of the Pen, and filled using the value of the Brush. The Pen can be set using the [SetPenColor](#), [SetPenStyle](#), and [SetPenWidth](#) procedures. The Brush can be

set using the [SetBrushColor](#) and [SetBrushStyle](#) procedures.

Example

To draw an ellipse from [coordinate](#) (100, 100) to coordinate (300, 200) on the screen :

```
Ellipse(100, 100, 300, 200);
```

To draw a circle of diameter 50 centred on coordinate (300, 200) on the screen :

```
Ellipse(250, 150, 350, 250);
```

4.25.8 LineTo Procedure

The LineTo procedure draws a line on the screen.

Applicability

Colour C-Touch only.

Syntax

```
LineTo(X, Y);
```

X and Y are [Integers](#)

Description

Use LineTo to draw a line from the current pen position (set using [MoveTo](#) or a previous LineTo) up to, but not including the [point](#) (X,Y). LineTo changes the value of the current Pen position to (X,Y).

The line is drawn using the Pen. The Pen can be set using the [SetPenColor](#), [SetPenStyle](#), and [SetPenWidth](#) procedures.

Example

To draw an line from the current pen position to coordinate (300, 200) on the screen :

```
LineTo(300, 200);
```

4.25.9 MoveTo Procedure

The MoveTo procedure sets the Pen Position.

Applicability

Colour C-Touch only.

Syntax

```
MoveTo(X, Y);
```

X and Y are [Integers](#)

Description

Use MoveTo to set the current Pen position to [point](#) (X,Y). This is needed before using [LineTo](#).

Example

To move the current pen position to coordinate (300, 200) on the screen :

```
MoveTo(300, 200);
```

4.25.10 Rectangle Procedure

The Rectangle procedure draws a rectangle or square on the screen.

Applicability

Colour C-Touch only.

Syntax

```
Rectangle(Left, Top, Right, Bottom);
```

Left, Top, Right and Bottom are [Integers](#)

Description

The rectangle is outlined using the value of the Pen, and filled using the value of the Brush. The Pen can be set using the [SetPenColor](#), [SetPenStyle](#), and [SetPenWidth](#) procedures. The Brush can be set using the [SetBrushColor](#) and [SetBrushStyle](#) procedures.

Example

To draw a rectangle from coordinate (100, 100) to coordinate (300, 200) on the screen :

```
Rectangle(100, 100, 300, 200);
```

To draw a square of width 100 centred on coordinate (300, 200) on the screen :

```
Rectangle(250, 150, 350, 250);
```

4.25.11 RoundRect Procedure

The RoundRect procedure draws a rounded rectangle or square on the screen.

Applicability

Colour C-Touch only.

Syntax

```
RoundRect(Left, Top, Right, Bottom, Radius);
```

Left, Top, Right, Bottom and Radius are [Integers](#)

Description

The RoundRect procedure draws a rectangle or square with rounded corners having a specified Radius.

The rounded rectangle is outlined using the value of the Pen, and filled using the value of the Brush. The Pen can be set using the [SetPenColor](#), [SetPenStyle](#), and [SetPenWidth](#) procedures. The Brush can be set using the [SetBrushColor](#) and [SetBrushStyle](#) procedures.

Example

To draw a rounded rectangle from coordinate (100, 100) to coordinate (300, 200), with a radius of 10 :

```
RoundRect(100, 100, 300, 200, 10);
```

To draw a rounded square of width 100 centred on coordinate (300, 200), with a radius of 20 :

```
RoundRect(250, 150, 350, 250, 20);
```

4.25.12 SetBrushColor Procedure

The SetBrushColor procedure sets the colour of the brush used for drawing solid shapes on the screen.

Applicability

Colour C-Touch only.

Syntax

```
SetBrushColor(c);
```

c is an [Integer](#)

Description

The SetBrushColor procedure sets the colour to be used for filling [Rectangles](#), [Round Rectangles](#) and [Ellipses](#).

See the [Colours](#) topic for details of specifying colours.

Example

To set the brush colour to blue :

```
SetBrushColor(clBlue);
```

4.25.13 SetBrushStyle Procedure

The SetBrushStyle procedure sets the style of the brush used for drawing solid shapes on the screen.

Applicability

Colour C-Touch only.

Syntax

```
SetBrushStyle(s);
```

s is an [Integer](#)

Description

The SetBrushStyle procedure sets the style to be used for filling [Rectangles](#), [Round Rectangles](#) and [Ellipses](#). The styles are as follows :

Value	Brush Style	Constant
0	Solid (default)	bsSolid
1	Clear	bsClear
2	Horizontal	bsHorizontal

3	Vertical	bsVertical
4	Backwards Diagonal	bsBDiagonal
5	Forwards Diagonal	bsFDiagonal
6	Cross	bsCross
7	Diagonal Cross	bsDiagCross

For the backgrounds with lines, the lines are drawn in the [Brush Colour](#).

Example

To set the brush style to horizontal lines :

```
SetBrushStyle(bsHorizontal);
```

4.25.14 SetFontColor Procedure

The SetFontColor procedure sets the colour of the font.

Applicability

Colour C-Touch only.

Syntax

```
SetFontColor(c);
```

c is an [Integer](#)

Description

The SetFontColor procedure sets the colour to be used for writing [Text](#) on the screen. The background of the text is drawn in the [Brush Colour](#).

See the [Colours](#) topic for details of specifying colours.

Example

To set the font colour to blue :

```
SetFontColor(c1Blue);
```

4.25.15 SetFontName Procedure

The SetFontName procedure sets the name of the font.

Applicability

Colour C-Touch only.

Syntax

```
SetFontName(FontName);
```

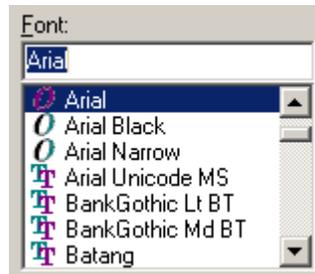
FontName is a [String](#)

Description

The SetFontName procedure sets the name of the font to be used for writing [Text](#) on the screen. Font names can be found by running any program which uses fonts and look at the list of fonts

available. If you have not selected a font name, 'Arial' will be used.

For example, in PICED, select the **Default Font** button on the Toolbar. Click the **Select Other Fonts** button. In the Font list, you will see the names of all fonts which can be used.



For colour C-Touch projects, you will need to either use one of the colour C-Touch pre-installed fonts (Arial, Courier New, System, Tahoma, Times New Roman, Webdings and Wingdings) or include your selected font in the transfer archive.

Example

To set the font name to Courier New :

```
SetFontName('Courier New');
```

4.25.16 SetFontSize Procedure

The SetFontSize procedure sets the size of the font.

Applicability

Colour C-Touch only.

Syntax

```
SetFontSize(n);
```

n is an [Integer](#)

Description

The SetFontSize procedure sets the size of the font to be used for writing [Text](#) on the screen.

Example

To set the font size to 12 (the default) :

```
SetFontSize(12);
```

4.25.17 SetFontStyle Procedure

The SetFontStyle procedure sets the style of the font.

Applicability

Colour C-Touch only.

Syntax

```
SetFontStyle(n);
```

n is an [Integer](#)

Description

The SetFontStyle procedure sets the style of the font to be used for writing [Text](#) on the screen. The value of n is the sum of the various style options required :

Value	Style	Constant
1	Bold	fsBold
2	Italics	fsItalic
4	Underline	fsUnderline
8	Strikeout	fsStrikeout

To get a combination of styles, add the values together.

Example

To set the font to standard (bold etc is off) :

```
SetFontStyle(0);
```

To set the font to bold :

```
SetFontStyle(fsBold);
```

To set the font to bold italics :

```
SetFontStyle(fsBold + fsItalic);
```

4.25.18 SetPenColor Procedure

The SetPenColor procedure sets the colour of the pen used for drawing lines and the outline of solid shapes on the screen.

Applicability

Colour C-Touch only.

Syntax

```
SetPenColor(c);
```

c is an [Integer](#)

Description

The SetPenColor procedure sets the colour to be used for drawing [Lines](#) and the outline of [Rectangles](#), [Round Rectangles](#) and [Ellipses](#).

See the [Colours](#) topic for details of specifying colours.

Example

To set the pen colour to blue :

```
SetPenColor(c1Blue);
```

4.25.19 SetPenStyle Procedure

The SetPenStyle procedure sets the style of the pen used for drawing lines and the outline of solid shapes on the screen.

Applicability

Colour C-Touch only.

Syntax

```
SetPenStyle(s);
```

s is an [Integer](#)

Description

The SetPenStyle procedure sets the style to be used for drawing [Lines](#) and the outline of [Rectangles](#), [Round Rectangles](#) and [Ellipses](#). The styles are as follows :

Value	Pen Style	Constant
0	Solid (default)	psSolid
1	Dash	psDash
2	Dot	psDot
3	Dash Dot	psDashDot
4	Dash Dot Dot	psDashDotDot
5	None (no line drawn)	psClear

Note that for dotted and dashed lines, the pen [width](#) must be 1. For these lines, the "spaces" in the line are drawn in the [Brush Colour](#).

Example

To set the pen style to dashes :

```
SetPenStyle(psDash);
```

4.25.20 SetPenWidth Procedure

The SetPenWidth procedure sets the width of the pen used for drawing lines and the outline of solid shapes on the screen.

Applicability

Colour C-Touch only.

Syntax

```
SetPenWidth(w);
```

w is an [Integer](#)

Description

The SetPenWidth procedure sets the width of the Pen to be used for drawing [Lines](#) and the outline of [Rectangles](#), [Round Rectangles](#) and [Ellipses](#).

Example

To set the pen width to 2 :

```
SetPenWidth(2);
```

4.25.21 TextHeight Function

The TextHeight function returns the height of some text.

Applicability

Colour C-Touch only.

Syntax

```
TextHeight(text)
```

text is a [String](#)

Description

Use the TextHeight function to obtain the height of some text (in pixels) when drawn in the current font (as set by the [SetFontName](#), [SetFontSize](#) and [SetFontStyle](#) procedures).

Example

To draw the text in string WarningMessage at the left centre (vertically) of the screen :

```
TextPos(0, (ScreenHeight - TextHeight(WarningMessage)) div 2);  
DrawText(WarningMessage);
```

See also [TextWidth Function](#), [DrawTextBlock Procedure](#)

4.25.22 TextPos Procedure

The TextPos procedure sets the position for writing Text on the screen.

Applicability

Colour C-Touch only.

Syntax

```
TextPos(X, Y);
```

X and Y are [Integers](#)

Description

Use TextPos to set the text position to [coordinate](#) (X,Y). This is needed before using the [DrawText Procedure](#).

Example

To set the text position to coordinate (300, 200) on the screen :

```
TextPos(300, 200);
```

See also [TextHeight Function](#), [TextWidth Function](#)

4.25.23 TextWidth Function

The TextWidth function returns the width of some text.

Applicability

Colour C-Touch only.

Syntax

```
TextWidth(text)
```

text is a [String](#)

Description

Use the TextWidth function to obtain the width of some text (in pixels) when drawn in the current font (as set by the [SetFontName](#), [SetFontSize](#) and [SetFontStyle](#) procedures).

Example

To draw the text in string WarningMessage at the top centre of the screen :

```
TextPos((ScreenWidth - TextWidth(WarningMessage)) div 2, 0);  
DrawText(WarningMessage);
```

See also [TextHeight Function](#), [DrawTextBlock Procedure](#)

4.25.24 GetClick Function

The GetClick function returns whether the user has clicked the screen.

Applicability

Colour C-Touch only.

Syntax

```
GetClick
```

Description

The GetClick function returns a boolean value indicating whether the screen has been clicked since the last [scan](#). If the PICED page has changed since the click, then GetClick will return false, otherwise you may think that the user has clicked on the new page, which is not the case. This means that if the user clicks on a button which changes the page, the GetClick function will always return false.

Example

4.25.25 GetClickX Function

The GetClickX function returns the X coordinate of the last screen click.

Applicability

Colour C-Touch only.

Syntax

```
GetClickX
```

Description

The GetClickX function returns an integer value with the X [coordinate](#) of the most recent click on the screen. The [GetClick Function](#) will return whether the screen has been clicked, and hence whether the GetClickX data is valid.

[Example](#)

4.25.26 GetClickY Function

The GetClickY function returns the Y coordinate of the last screen click.

Applicability

Colour C-Touch only.

Syntax

```
GetClickY
```

Description

The GetClickY function returns an integer value with the Y [coordinate](#) of the most recent click on the screen. The [GetClick Function](#) will return whether the screen has been clicked, and hence whether the GetClickY data is valid.

[Example](#)

4.25.27 Click Example

Some logic needs to know whether the user has clicked on the rectangular area of the screen bound by the points (left, top) and (right, bottom). To determine whether the region of the screen has been clicked by the user, and if so, set a scene :

```
if GetClick then
begin
  if (GetClickX >= Left) and (GetClickX <= Right) and (GetClickY >= Top) and
  (GetClickY <= Bottom) then
    SetScene("All On");
end;
```

4.26 Serial IO



It is possible to read from and write to serial ports from the Logic Engine. This enables interfaces to many automation and Audio/Visual products to be created.

Up to four serial ports can be used simultaneously. The serial ports are referred to by their index (1 -

4) which is not the same as the COM Port number. For example, serial port number (index) 1 could be COM Port 5.

The functions included for the support of serial ports are :

- [CloseSerial Procedure](#)
- [OpenSerial Procedure](#)
- [ReadSerial Procedure](#)
- [WriteSerial Procedure](#)
- [SetSerialDTR Procedure](#)
- [SetSerialRTS Procedure](#)

Notes

Serial ports must be [opened](#) before they can be used. The serial port should be opened in the [Initialisation](#) section of the code.

The [WriteSerial](#) procedure does not wait until the command has been sent before continuing with the next line of code. Hence the following code will most probably result in the ReplyString being empty, as there will not be enough time for the TransmitString to be sent, let alone for a reply to have been received :

```
WriteSerial(2, TransmitString);
ReadSerial(2, ReplyString, #13#10);
if ReplyString <> '' then
begin
...

```

The preferred way would be something like :

```
WriteSerial(2, TransmitString);
delay(1);
ReadSerial(2, ReplyString, #13#10);
if ReplyString <> '' then
begin
...

```

Because [delays](#) can not be used in the [initialisation](#) section, this code can not be used there. It can only be used in a module.

See also [Serial IO Examples](#) and [UTF-8 Example](#)

Logic serial messages can be shown with the **Show Logic Serial Messages** option on the Log. The log will show the data in the receive buffer at the start of each [scan](#). If the data from the serial port is not read, then the buffer will not be cleared and the data will keep being logged.

4.26.1 CloseSerial Procedure

The CloseSerial procedure closes a serial port.

Applicability

Colour C-Touch and Black & White C-Touch only.

Syntax

```
CloseSerial(SerialPortIndex);
```

SerialPortIndex is an [Integer](#)

Description

This closes the [serial port number](#) SerialPortIndex (1 - 4) and makes it available to be re-used. This procedure should rarely need to be used.

Example

To close serial port number 2 :

```
CloseSerial(2);
```

4.26.2 OpenSerial Procedure

The OpenSerial procedure opens a serial port.

Applicability

Colour C-Touch, Black & White C-Touch and PAC only.

Syntax

```
OpenSerial(SerialPortIndex, COMPortNo, BaudRate, DataBits, StopBits,
FlowControl, Parity);
```

SerialPortIndex, COMPortNo, BaudRate, DataBits, StopBits, FlowControl and Parity are [Integers](#)

Description

This opens the [serial port number](#) SerialPortIndex (1 - 4) with the following properties :

- COMPortNo is the PC COM Port number
- BaudRate is the baud rate (in bits per second)
- DataBits is the number of data bits (5 to 8)
- StopBits is the number of stop bits (0 = 1.5 stop bits, 1 = one stop bit, 2 = two stop bits)
- FlowControl (0 = none, 1 = hardware flow control, 2 = software flow control)
- Parity (0 = none, 1 = odd, 2 = even, 3 = mark, 4 = space)

For use with the Pascal Automation Controller, there are the following limitations :

- SerialPortIndex is the User Port number (1 or 2)
- COMPortNo is the User Port number (1 or 2)
- BaudRate is limited to 600, 1200, 2400, 4800, 9600, 19200, 38400
- DataBits is limited to 7 or 8
- StopBits is limited to 1 (= one stop bit) or 2 (= two stop bits)
- FlowControl is not used
- Parity options are limited to 0 (= none), 1 (= odd) or 2 (= even)

For use with the C-Touch Mark 2 or C-Touch Spectrum, there are the following limitations :

- SerialPortIndex can only be 1
- COMPortNo can only be 1
- BaudRate is limited to 600, 1200, 2400, 4800, 9600, 19200, 38400
- DataBits is limited to 7 or 8
- StopBits is limited to 1 (= one stop bit) or 2 (= two stop bits)
- FlowControl is not used
- Parity options are limited to 0 (= none), 1 (= odd) or 2 (= even)

With the Colour C-Touch, you can only use COM Port number 1.

The following constants have been defined for use with opening a serial port :

Value	Meaning	Constant Name
0	1.5 Stop Bits	scOneAndHalfStopBits
1	1 Stop Bit	scOneStopBit
2	2 Stop Bits	scTwoStopBits
0	No Flow Control	scNoFlowControl
1	Hardware Flow Control	scHardwareFlowControl
2	Software Flow Control	scSoftwareFlowControl
0	No Parity	scNoParity
1	Odd Parity	scOddParity
2	Even Parity	scEvenParity
3	Mark Parity	scMarkParity
4	Space Parity	scSpaceParity

Example

To open serial port number 1, with COM Port 4, 9600 baud, 8 data bits, 1 stop bit, no flow control, no parity :

```
OpenSerial(1, 4, 9600, 8, 1, 0, 0);
```

OR

```
OpenSerial(1, 4, 9600, 8, scOneStopBit, scNoFlowControl, scNoParity);
```

See also [IsCBusUnit Function](#)

4.26.3 ReadSerial Procedure

The ReadSerial procedure reads data from a serial port.

Applicability

Colour C-Touch, Black & White C-Touch and PAC only.

Syntax

```
ReadSerial(SerialPortIndex, DataString, Terminator);
```

SerialPortIndex is an [Integer](#)
 DataString is a [String](#) variable
 Terminator is a [String](#) expression

Description

This reads received data from the [serial port number](#) SerialPortIndex (1 - 4). The Terminator is used to determine where one string ends and the next commences. Typically, the Terminator will be a Carriage Return / Line Feed pair. The result (not including the Terminator) is stored in the DataString variable.

If the Terminator is a null string, the whole of the received string will be placed in the DataString.

Received strings are placed in a buffer until they are read by the user using the ReadSerial procedure. If the buffer exceeds 10,000 bytes (1000 bytes for PAC), the newest data will be ignored (i.e. lost). The ReadSerial procedure just reads whatever is currently in the buffer. It does **not** wait until there is data to be read.

Example

To read a string from serial port number 2 which will be terminated by a Carriage Return (ASCII #13) / Line Feed (ASCII #10) pair :

```
ReadSerial(2, s, #13#10);
```

To read all received data from serial port number 1 :

```
ReadSerial(1, s, '');
```

To keep reading any received strings in the buffer until there are none left :

```
repeat
  ReadSerial(2, s, #13#10);
  if s <> '' then
  begin
    ...
  end;
until s = '';
```

4.26.4 WriteSerial Procedure

The WriteSerial procedure sends data to a serial port.

Applicability

Colour C-Touch, Black & White C-Touch and PAC only.

Syntax

```
WriteSerial(SerialPortIndex, DataString);
```

SerialPortIndex is an [Integer](#)

DataString is a [String](#)

Description

This writes data to the [serial port number](#) SerialPortIndex (1 - 4). Note that the data gets stored in a buffer and is sent out as a "background" process. If the buffer exceeds 10,000 bytes (1000 bytes for PAC), the newest data will be ignored.

Example

To write a string s to serial port number 2 :

```
WriteSerial(2, s);
```

To write the string 'stop' terminated with a [carriage return character](#) to serial port 1 :

```
WriteSerial(1, 'stop'#13);
```

4.26.5 SetSerialDTR Procedure

The SetSerialDTR procedure sets the serial port Data Terminal Ready (DTR) line state.

Applicability

Colour C-Touch only.

Syntax

```
SetSerialDTR(SerialPortIndex, State);
```

SerialPortIndex is an [Integer](#)

State is [Boolean](#)

Description

This sets the serial port Data Terminal Ready (DTR) line for the [serial port number](#) SerialPortIndex (1 - 4). For an RS232 serial port, when State is true, the DTR line will be set to a negative voltage.

4.26.6 SetSerialRTS Procedure

The SetSerialRTS procedure sets the serial port Request To Send (RTS) line state.

Applicability

Colour C-Touch only.

Syntax

```
SetSerialRTS(SerialPortIndex, State);
```

SerialPortIndex is an [Integer](#)

State is [Boolean](#)

Description

This sets the serial port Request To Send (RTS) line for the [serial port number](#) SerialPortIndex (1 - 4). For an RS232 serial port, when State is true, the RTS line will be set to a negative voltage.

4.26.7 Serial IO Examples

The first step in using serial IO to control a device is to obtain the serial protocol for the device. Generally this information is available in the instruction manual for the device or it can be obtained from the manufacturer's web site.

Understanding and Testing the Device Protocol

Sometimes it may not be clear whether the format of the protocol uses [ASCII](#) characters or not (see examples below). It is critical to ensure that you have a sufficient understanding of the protocol before you start writing logic code. See [Debugging Serial](#) for information on how to do this.

Example 1 - ASCII Based Protocol

The simplest serial protocols use readable [ASCII](#) text. For example, an audio amplifier may have several commands :

Command	Format	Response
Set Source	Source=<Source><CR><LF> <Source> = 0 to 4	OK<CR><LF> (if command succeeds) FAIL<CR><LF> (if the command fails)
Set Volume Level	Volume=<Level><CR><LF> <Level> = 0 to 100 0 = -50dB 100 = 0dB	OK<CR><LF> (if command succeeds) FAIL<CR><LF> (if the command fails)

Set Bass Level	Bass=<Level><CR><LF> <Level> = 0 to 100 0 = -10dB 50 = 0dB 100 = +10dB	OK<CR><LF> (if command succeeds) FAIL<CR><LF> (if the command fails)
Set Treble Level	Treble=<Level><CR><LF> <Level> = 0 to 100 0 = -10dB 50 = 0dB 100 = +10dB	OK<CR><LF> (if command succeeds) FAIL<CR><LF> (if the command fails)

Using this protocol, you would send the serial command "Source=2" to select the second input source. Similarly, you would send the command "Volume=50" for a mid-level volume.

All commands in this particular protocol are terminated with <CR><LF>, which is an ASCII "Carriage Return" (number 13 decimal) followed by an ASCII "Line Feed" (number 10 decimal). Different protocols use different terminators, and some don't use one at all.

For our example, we want to use the level on a group address ([Tag](#) "Amp Source") to select the source input. A level of 0 will select the first source and so on. A touch screen will be used with a "selector" component to select the source.

```
{ global variables section of code }
CurrentSource : integer; { this is the source currently selected }
RequiredSource : integer; { this is the source required }
CommandString : string; { this is the string to be sent to the amplifier }

{ Module 1 code }
RequiredSource := GetLightingLevel("Amp Source");
{ If the required source is different from the current source then send a command to the Amp }
if RequiredSource <> CurrentSource then
begin
  Format(CommandString, 'Source=', RequiredSource:0, #13#10);
  WriteSerial(1, CommandString);
  CurrentSource := RequiredSource;
end;
```

You would need to write similar code to control the volume, bass and treble. The code to control the volume is shown below :

```
{ global variables section of code }
CurrentVolume : integer; { this is the current amplifier volume }
RequiredVolume : integer; { this is the required amplifier volume }

{ Module 1 code }
{ The volume group address can be 0 to 255, but the amp requires a value of 0 to 100 }
RequiredVolume := LevelToPercent(GetLightingLevel("Amp Volume"));
{ If the required volume is different from the current volume then send a command to the Amp }
if RequiredVolume <> CurrentVolume then
begin
  Format(CommandString, 'Volume=', RequiredVolume:0, #13#10);
  WriteSerial(1, CommandString);
  CurrentVolume := RequiredVolume;
end;
```

So far, we are just controlling the amplifier. It may also be necessary to read data from the amplifier or to respond to changes the user has made to the amplifier settings. Let's assume that the amplifier has the following additional commands which can be used :

Command	Format	Response
Get Source	Get Source<CR><LF>	Source=<Source><CR><LF> <Source> = 0 to 4
Get Volume Level	Get Volume<CR><LF>	Volume=<Level><CR><LF> <Level> is as above
Get Bass Level	Get Bass<CR><LF>	Bass=<Level><CR><LF> <Level> is as above
Get Treble Level	Get Treble<CR><LF>	Treble=<Level><CR><LF> <Level> is as above

In addition, this hypothetical amplifier will send the same message shown in the response section above if the user makes a change to the settings on the amplifier.

So for example, if you want to find which source is selected, you would send the command "Get Source" and the amplifier would reply with "Source=2" if the third source was selected.

To complete the example, we will need two more modules of code. The first is used to read the data from the amplifier and the second deals with the responses from the amplifier, and to any messages related to the user making changes.

```

{ global variables section of code }
ReceivedString : string;      { this is the string received from the serial port }
EqualsPos : integer;         { this is the position of an = sign in the received
string }
ReceivedDataString : string; { this is the data portion of the string }
ReceivedData : integer;      { this is the actual data }

{ Module 2 code }
WriteSerial(1, 'Get Source'#13#10);
delay(1);
WriteSerial(1, 'Get Volume'#13#10);
delay(1);
WriteSerial(1, 'Get Bass'#13#10);
delay(1);
WriteSerial(1, 'Get Treble'#13#10);
DisableModule("Module 2");

{ Module 3 code }
ReadSerial(1, ReceivedString, #13#10);
{ see if there is an "=" sign in the string }
EqualsPos := Pos('=', ReceivedString);
if EqualsPos > 0 then
begin
  { extract the data from the received string }
  Copy(ReceivedDataString, ReceivedString, EqualsPos + 1, 3);
  ReceivedData := StringToInt(ReceivedDataString);
  { see if the source has changed, and if so, send to C-Bus }
  if Pos('Source', ReceivedString) = 1 then
  begin
    if ReceivedData <> CurrentSource then
    begin
      CurrentSource := ReceivedData;
      SetLightingLevel("Amp Source", CurrentSource, 0);
    end;
  end;
end;

```

```

end;
{ repeat the above for volume, bass and treble }
end;

```

Example 2 - Binary Protocol

Although ASCII protocols are very easy to use because they are "human readable", they aren't very efficient. For example, using the above protocol to set the amplifier volume to full level, you need to send the command "Volume=100" which contains 10 characters (bytes) of data. This is very wasteful and slow, because you really only need two bytes; one to determine the type of command and one with the new level. A "binary" protocol to control the same amplifier might be something like that below :

Command	Format	Response
Set Source	1<Source><CR><LF> <Source> = 0 to 4	0<CR><LF> (if the command succeeds) 1<CR><LF> (if the command fails)
Set Volume Level	2<Level><CR><LF> <Level> = 0 to 100 (usage as above)	0<CR><LF> (if the command succeeds) 1<CR><LF> (if the command fails)
Set Bass Level	3<Level><CR><LF> <Level> = 0 to 100 (usage as above)	0<CR><LF> (if the command succeeds) 1<CR><LF> (if the command fails)
Set Treble Level	4<Level><CR><LF> <Level> = 0 to 100 (usage as above)	0<CR><LF> (if the command succeeds) 1<CR><LF> (if the command fails)
Get Source	5<CR><LF>	1<Source><CR><LF> <Source> is as above
Get Volume Level	6<CR><LF>	2<Level><CR><LF> <Level> is as above
Get Bass Level	7<CR><LF>	3<Level><CR><LF> <Level> is as above
Get Treble Level	8<CR><LF>	4<Level><CR><LF> <Level> is as above

In this case, to select the second source, you would send a command with a character (byte) with the value 1 (not [ASCII](#) 1 which has a value of 49), followed by a byte with the value 2 and the carriage return and line feed.

The big advantage of using binary protocols (other than efficiency) is their simplicity for writing code. You can see that extracting data from the response is going to be particularly easy. The first byte is the data type and the second is the data itself.

The equivalent code to do the same as Example 1 is :

```

{ global variables section of code }
CurrentSource : integer;   { this is the source currently selected }
RequiredSource : integer; { this is the source required }
CommandString : string;   { this is the string to be sent to the amplifier }
CurrentVolume : integer;  { this is the current amplifier volume }
RequiredVolume : integer; { this is the required amplifier volume }
ReceivedString : string;  { this is the string received from the serial port }
ReceivedData : integer;   { this is the received data }

{ Module 1 code }
RequiredSource := GetLightingLevel("Amp Source");

```

```

{ If the required source is different from the current source then send a command
to the Amp }
if RequiredSource <> CurrentSource then
begin
  Format(CommandString, #1:0, chr(RequiredSource):0, #13#10);
  WriteSerial(1, CommandString);
  CurrentSource := RequiredSource;
end;
{ The volume group address can be 0 to 255, but the amp requires a value of 0 to
100 }
RequiredVolume := LevelToPercent(GetLightingLevel("Amp Volume"));
{ If the required volume is different from the current volume then send a command
to the Amp }
if RequiredVolume <> CurrentVolume then
begin
  Format(CommandString, #2:0, chr(RequiredVolume):0, #13#10);
  WriteSerial(1, CommandString);
  CurrentVolume := RequiredVolume;
end;

{ Module 2 code }
WriteSerial(1, #5#13#10);
delay(1);
WriteSerial(1, #6#13#10);
delay(1);
WriteSerial(1, #7#13#10);
delay(1);
WriteSerial(1, #8#13#10);
DisableModule("Module 2");

{ Module 3 code }
ReadSerial(1, ReceivedString, #13#10);
{ extract the data from the received string }
ReceivedData := ord(ReceivedString[2]);
{ see if the source has changed, and if so, send to C-Bus }
if ord(ReceivedString[1]) = 1 then
begin
  if ReceivedData <> CurrentSource then
  begin
    CurrentSource := ReceivedData;
    SetLightingLevel("Amp Source", CurrentSource, 0);
  end;
end;
{ repeat the above for volume, bass and treble }
end;

```

Other Protocols

It is common for a mixture of ASCII and binary to be used in a protocol. For example, setting the volume may use a command with the ASCII string "Volume=" followed by a character (byte) containing the level, rather than an ASCII string.

Serial protocols often use [Hexadecimal Numbers](#). This can cause additional confusion, as it can be used with both ASCII and binary protocols, and is sometimes not well explained in the protocol documents. For example, setting the volume may use a command with the ASCII string "Volume=" followed by a hexadecimal string containing the level. In this case, to set the level to 100, the command would be "Volume=64" (because "64" is the hexadecimal equivalent of the number 100).

If the protocol says you need to use a character with a hexadecimal value of 0A for example (which corresponds to an ASCII line feed), you can express the character as either a decimal constant (#10) or a hexadecimal constant (#\$0A). Hence the following two lines will result in the same string being sent :

```
WriteSerial(1, 'Get Source'#13#10);  
WriteSerial(1, 'Get Source'#$0D#$0A);
```

Notes

The above examples are not intended to be a solution to any specific real-world problem. They are introduced merely to provide an introduction to some of the techniques required to implement serial control of a device. The protocols used with real devices can be considerably more complex and can have some significant difficulties which need to be overcome.

Also, the implementation used above where group addresses are mapped to "levels" in the device will not necessarily suit all applications. It is common to need to receive "triggers" from C-Bus to change a channel up or down, or to increase/decrease volume, rather than directly mapping a group address level to a function. This type of implementation will require significantly different code.

4.26.8 Debugging Serial

If you are having problems communicating with a device using serial IO, there are several possible causes which are listed below.

Connections

Check that the cables are wired correctly. Make sure the cable is the correct type, connecting Transmit (Tx) from the computer to Receive (Rx) on the device and vice versa. If the device requires the use of hardware flow control, ensure that all of the signals are connected.

Protocol

Check that you understand the operation of the device protocol. Often the documentation for devices is incomplete or even incorrect.

The best way to ensure you correctly understand the protocol is to use a serial terminal program to send commands to the device and to observe the responses. If you can control the device this way, then it indicates that you understand how the protocol works. If you can't control the device this way, it is probably going to be a waste of time trying to write logic code to control the device.

To do this :

1. Connect your computer to the device you want to control (see above)
2. Run a serial terminal program (Hyper Terminal is fine for ASCII based protocols. For other protocols, you will need to use other software)
3. Set the serial terminal program properties to match those for the device being controlled :
 - COM Port
 - Baud rate
 - Data bits
 - Stop bits
 - Parity
 - Flow control
4. Send some test commands and observe what happens

Messages

On the Log Form, there is an option to **Show Logic Serial Messages**. When this is enabled :

- all serial messages sent by the logic are shown, along with the line number of the logic code
- at the start of each scan, the content of the serial receive buffers are shown if they are not empty. This allows you to see the received data and whether it is being read by the logic code.

All control characters (non-printable characters) in strings are replaced by the numerical value of the character contained within "<" and ">" delimiters. For example, if a string 'ABC#13#10' was sent to Com Port 1 by logic line 123, the message logged would be :

```
Logic : COM1 Tx ABC<13><10> (Logic Line 123)
```

If the contents of the received buffers exceed 100 characters in length, they will be truncated before being logged and "<x more...>" will be added to the end of the data (where x is the number of additional characters).

Using Different COM Ports in Device and PC

If you want to use a serial port in an embedded device (C-Touch or PAC) and there is no corresponding port in your computer, there are two ways of preventing errors when simulating the project.

Example 1

You want to use serial port 2 on the PAC. You want to test the project in PICED, but the computer does not have COM Port 2, but it does have COM Port 1.

The solution is to open a different port depending on where the code is running (in the PAC or on your computer), as shown below:

```
{ Initialisation section }

if IsPAC then
  OpenSerial(2, 2, 9600, 8, 1, scNoFlowControl, scNoParity)
else
  OpenSerial(2, 1, 9600, 8, 1, scNoFlowControl, scNoParity);
```

Elsewhere in the code when you use serial port index 2 it will use serial port 2 in a PAC, but COM 1 on your computer.

Example 2

You want to use serial port 2 on the PAC. You want to test the project in PICED, but the computer does not any COM Ports. You do not want to test the serial functions of the PAC.

The solution is to have all serial functions only run in the PAC, as shown below:

```
{ Initialisation section }

if IsPAC then
  OpenSerial(2, 2, 9600, 8, 1, scNoFlowControl, scNoParity);

{ Modules section }

if IsPAC then
  WriteSerial(2, CommandString);
```

4.26.9 Tutorial 10

Question 1

Write some code to send a command string 'AT' to a modem on serial port COM1 and wait for up to 10 seconds for a reply of 'OK'. All commands and replies are terminated by a carriage return / Line Feed pair. If the connection was successful, log the message "Modem Connected", otherwise log "Modem Connection Failed".

Tutorial Answers

4.27 Internet



Various Internet technologies can be used with the Logic Engine, including:

- [TCP/IP](#)
- [UDP](#)
- [Ping](#)
- [DNS](#)
- [HTTP](#)
- [E-Mail](#)

It is also possible to obtain information about the computer's [Network Adaptors](#).

4.27.1 TCP/IP

It is possible to read from and write to TCP/IP Sockets from the Logic Engine. This enables interfaces to many automation and Audio/Visual products to be created, as well as interfacing PICED to other software products.

Client Sockets

Client sockets are used to allow the software to connect to other software and devices and communicate with them. More than one client socket can be open at once. See Software Limits for details.

The functions included for the support of client sockets are :

- [ClientSocketConnected Function](#)
- [ClientSocketError Function](#)
- [CloseClientSocket Procedure](#)
- [OpenClientSocket Procedure](#)
- [ReadClientSocket Procedure](#)
- [WriteClientSocket Procedure](#)

Server Sockets

Server sockets are used to allow other software and devices to connect to the software. There is only one server socket available for use with logic. It is not possible to open multiple server sockets at the same time.

The functions included for the support of server sockets are :

- [CloseServerSocket Procedure](#)
- [OpenServerSocket Procedure](#)
- [ReadServerSocket Procedure](#)

[ServerSocketActive Function](#)
[ServerSocketError Function](#)
[ServerSocketHasClient Function](#)
[WriteServerSocket Procedure](#)

Notes

Sockets may take a small amount of time to be connected. The following code may not work, because the socket may not be connected by the time the message is sent:

```
OpenClientSocket(1, '192.168.100.208', 10002);  
WriteClientSocket(1, 'Hello');
```

It is better to allow time for the socket to connect, like this, for example:

```
OpenClientSocket(1, '192.168.100.208', 10002);  
WaitUntil(ClientSocketConnected(1));  
WriteClientSocket(1, 'Hello');
```

It is also advisable to allow some time after sending a message (if you are expecting a reply) before reading from the socket or closing it.

Logic TCP/IP messages can be shown with the **Show Logic TCP/IP Messages** option on the Log. The log will show the data in the receive buffer at the start of each [scan](#). If the data from the socket is not read, then the buffer will not be cleared and the data will keep being logged.

With Colour C-Touch, only ports in the range 10,000 to 19,999 can be used. These ports need to be enabled in the Colour C-Touch firewall before use.

See also [UTF-8 Example](#)

For more information on the use of TCP/IP sockets, refer to a suitable text book.

4.27.1.1 ClientSocketConnected Function

The ClientSocketConnected function returns whether the Client [Socket](#) has connected to a remote server.

Applicability

Colour C-Touch and Wiser Home Control only.

Syntax

```
ClientSocketConnected(SocketNumber)
```

SocketNumber is an [Integer](#)

Description

This returns a boolean value with the connected state (true/false) of the Client Socket.

Example

To perform an action only if the first Client socket is connected :

```
if ClientSocketConnected(1) then ...
```

4.27.1.2 ClientSocketError Function

The ClientSocketError function returns the error status of the Client [Socket](#).

Applicability

Colour C-Touch and Wiser Home Control only.

Syntax

```
ClientSocketError(SocketNumber)
```

SocketNumber is an [Integer](#)

Description

This returns an [Integer](#) value with the most recent Client Socket error message.

Value	Meaning
0	No Error
1	The socket received an error message that does not fit into any of the following categories.
2	An error occurred when trying to write to the socket connection.
3	An error occurred when trying to read from the socket connection.
4	A connection request that was already accepted could not be completed.
5	An error occurred when trying to close a connection.
6	A problem occurred when trying to accept a client connection request.
7	A problem occurred when trying to open a connection.

Reading this resets the value to 0. So if you need to use the value, you will need to assign it to another variable (see example below).

Example

```
ErrorNumber := ClientSocketError(1);
if ErrorNumber > 0 then
begin
  format(ErrorString, 'Error number ', ErrorNumber:0);
  ...
end;
```

4.27.1.3 CloseClientSocket Procedure

The CloseClientSocket procedure closes the Client [Socket](#).

Applicability

Colour C-Touch and Wiser Home Control only.

Syntax

```
CloseClientSocket(SocketNumber);
```

SocketNumber is an [Integer](#)

Description

This closes the Client socket and makes it available to be re-used.

4.27.1.4 CloseServerSocket Procedure

The CloseClientSocket procedure closes the Server [Socket](#).

Applicability

Colour C-Touch and Wiser Home Control only.

Syntax

```
CloseServerSocket;
```

Description

This closes the Server socket and makes it available to be re-used.

4.27.1.5 OpenClientSocket Procedure

The OpenClientSocket procedure opens the Client [Socket](#).

Applicability

Colour C-Touch and Wiser Home Control only.

Syntax

```
OpenClientSocket(SocketNumber, IPAddress, PortNumber);
```

SocketNumber is an [Integer](#)

IPAddress is a [String](#)

PortNumber is an [Integer](#)

Description

This opens the Client Socket to connect to IPAddress, and PortNumber.

Example

To open the first Client Socket to connect to the local machine, on port number 23 :

```
OpenClientSocket(1, '127.0.0.1', 23);
```

To open the second Client Socket to connect to IP Address 1.2.3.4, on port number 1000 :

```
OpenClientSocket(2, '1.2.3.4', 1000);
```

4.27.1.6 OpenServerSocket Procedure

The OpenServerSocket procedure opens the Server [Socket](#).

Applicability

Colour C-Touch and Wiser Home Control only.

Syntax

```
OpenServerSocket (PortNumber) ;
```

PortNumber is an [Integer](#)

Description

This opens the Server Socket to allow connections on PortNumber.

Example

To open the Server Socket for connects on port number 23 :

```
OpenServerSocket (23) ;
```

4.27.1.7 ReadClientSocket Procedure

The ReadClientSocket procedure reads data from a Client [Socket](#).

Applicability

Colour C-Touch and Wiser Home Control only.

Syntax

```
ReadClientSocket (SocketNumber, DataString, Terminator) ;
```

SocketNumber is an [Integer](#)

DataString is a [String](#) variable

Terminator is a [String](#) expression

Description

This reads received data from the Client Socket. The Terminator is used to determine where one string ends and the next commences. Typically, the Terminator will be a Carriage Return / Line Feed pair. The result is stored in the DataString variable.

If the Terminator is a null string, the whole of the received string will be placed in the DataString.

Received strings are placed in a buffer until they are read by the user. If the buffer exceeds 10,000 bytes, the newest data will be ignored.

Example

To read a string which will be terminated by a Carriage Return / Line Feed pair :

```
ReadClientSocket (1, s, #13#10) ;
```

To read all received data from the Client Socket :

```
ReadClientSocket (1, s, '');
```

To keep reading any received strings in the buffer until there are none left :

```
repeat
  ReadClientSocket (1, s, #13#10) ;
  if s <> '' then
  begin
    ...
  end ;
until s = '' ;
```

4.27.1.8 ReadServerSocket Procedure

The ReadServerSocket procedure reads data from a Server [Socket](#).

Applicability

Colour C-Touch and Wiser Home Control only.

Syntax

```
ReadServerSocket(DataString, Terminator);
```

DataString is a [String](#) variable

Terminator is a [String](#) expression

Description

This reads received data from the Server Socket. The Terminator is used to determine where one string ends and the next commences. Typically, the Terminator will be a Carriage Return / Line Feed pair. The result is stored in the DataString variable.

If the Terminator is a null string, the whole of the received string will be placed in the DataString.

Received strings are placed in a buffer until they are read by the user. If the buffer exceeds 10,000 bytes, the newest data will be ignored.

Example

To read a string which will be terminated by a Carriage Return / Line Feed pair :

```
ReadServerSocket(s, #13#10);
```

To read all received data from the Server Socket :

```
ReadServerSocket(s, '');
```

To keep reading any received strings in the buffer until there are none left :

```
repeat
  ReadServerSocket(s, #13#10);
  if s <> '' then
  begin
    ...
  end;
until s = '';
```

4.27.1.9 ServerSocketActive Function

The ServerSocketActive function returns whether the Server [Socket](#) is Active.

Applicability

Colour C-Touch and Wiser Home Control only.

Syntax

```
ServerSocketActive
```

Description

This returns a [boolean](#) value which shows whether the Server Socket is active or not.

Example

To perform an action only if the Server socket is active :

```
if ServerSocketActive then ...
```

4.27.1.10 ServerSocketError Function

The ServerSocketError function returns the error status of the Server [Socket](#).

Applicability

Colour C-Touch and Wiser Home Control only.

Syntax

```
ServerSocketError
```

Description

This returns an [Integer](#) value with the most recent Server Socket error message. See [ClientSocketError](#) for error values.

Reading this resets the value to 0. So if you need to use the value, you will need to assign it to another variable (see example below).

Example

```
ErrorNumber := ServerSocketError;  
if ErrorNumber > 0 then  
begin  
  format(ErrorString, 'Error number ', ErrorNumber:0);  
  ...  
end;
```

4.27.1.11 ServerSocketHasClient Function

The ServerSocketHasClient function returns whether the Server [Socket](#) has a Client.

Applicability

Colour C-Touch and Wiser Home Control only.

Syntax

```
ServerSocketHasClient
```

Description

This returns a [boolean](#) value which shows whether the Server Socket has a Client connected to it or not.

Example

To perform an action only if the Server socket has a Client :

```
if ServerSocketHasClient then ...
```

4.27.1.12 WriteClientSocket Procedure

The WriteClientSocket procedure sends data to the Client [Socket](#).

Applicability

Colour C-Touch and Wiser Home Control only.

Syntax

```
WriteClientSocket(SocketNumber, DataString);
```

SocketNumber is an [Integer](#)

DataString is a [String](#)

Description

This writes data to the Client socket.

Example

To write a string s to the first Client socket :

```
WriteClientSocket(1, s);
```

4.27.1.13 WriteServerSocket Procedure

The WriteServerSocket procedure sends data to the Server [Socket](#).

Applicability

Colour C-Touch and Wiser Home Control only.

Syntax

```
WriteServerSocket(DataString);
```

DataString is a [String](#)

Description

This writes data to the Server socket.

Example

To write a string s to the Server socket :

```
WriteServerSocket(s);
```

4.27.1.14 TCP/IP Socket Examples

The processes in using [TCP/IP sockets](#) to communicate with third-party devices are mostly the same as for serial devices. Please read [Serial IO Examples](#) before continuing.

Protocol

For these examples, the protocol consists of two messages to switch group addresses on and off:

```
on <group address><CR><LF>
off <group address><CR><LF>
```

Where <group address> is the group address number to be controlled.

All commands in this particular protocol are terminated with <CR><LF>, which is an ASCII "Carriage Return" (number 13 decimal) followed by an ASCII "Line Feed" (number 10 decimal).

For example, to switch group address 1 on, the command would be:

```
on 1<CR><LF>
```

Client Socket Example

For this example, the PICED is acting as a TCP/IP client. Messages will be sent to a server switching group addresses on and off.

In the initialisation section of the code, the client socket (number 12345 in this case) needs to be opened (server it at IP Address 127.0.0.1 in this example):

```
OpenClientSocket(1, '127.0.0.1', 12345);
```

Each time group 1 changes, a message will be sent to the server:

```
once GetLightingState("Group 1") = ON then
begin
  WriteClientSocket(1, 'on 1');
end;

once GetLightingState("Group 1") = OFF then
begin
  WriteClientSocket(1, 'off 1');
end;
```

Server Socket Example

For this example, the PICED is acting as a TCP/IP server. Messages will be received from a client switching group addresses on and off.

In the initialisation section of the code, the server socket (number 12345 in this case) needs to be opened:

```
OpenServerSocket(12345);
```

In a module, the messages are read from the socket, processed to determine the action required, and then the action is performed (switching a group address on or off):

```
{ Read a message from the socket }
ReadServerSocket(ReceivedCommand, #13#10);

{ If a message has been received, then process it }
if length(ReceivedCommand) > 0 then
begin
  if pos('on', ReceivedCommand) = 1 then          // is it an "on" command?
```

```
begin
  Copy(GroupString, ReceivedCommand, 4, 3); // get the group address
  Group := StringToInt(GroupString);
  SetLightingState(Group, ON);           // switch the group on
end
else
if pos('off', ReceivedCommand) = 1 then // is it an "off" command?
begin
  Copy(GroupString, ReceivedCommand, 5, 3); // get the group address
  Group := StringToInt(GroupString);
  SetLightingState(Group, OFF);           // switch the group off
end
else
end;
```

4.27.2 UDP

It is possible to read from and write to UDP Sockets from the Logic Engine. This enables interfaces to many automation and Audio/Visual products to be created, as well as interfacing PICED to other software products.

A single UDP server socket is available, but it can also be used as a UDP client. The UDP socket is opened with a receiving port number. When a UDP message is sent, the IP Address and Port number for the target is used.

Procedures which can be used with UDP include:

- [OpenUDPSocket Procedure](#)
- [CloseUDPSocket Procedure](#)
- [WriteUDPSocket Procedure](#)
- [ReadUDPSocket Procedure](#)
- [UDPSocketError Function](#)
- [UDPSocketActive Function](#)
- [SendWOL Procedure](#)

4.27.2.1 OpenUDPSocket Procedure

The OpenUDPSocket procedure opens the [UDP Socket](#).

Applicability

Colour C-Touch only.

Syntax

```
OpenUDPSocket (PortNumber);
```

PortNumber is an [Integer](#)

Description

This opens the UDP Socket, listening on PortNumber.

See also [UDP Example](#)

4.27.2.2 CloseUDPSocket Procedure

The CloseUDPSocket procedure closes the [UDP Socket](#).

Applicability

Colour C-Touch only.

Syntax

```
CloseUDPSocket;
```

Description

This closes the UDP socket and makes it available to be re-used.

See also [UDP Example](#)

4.27.2.3 WriteUDPSocket Procedure

The WriteUDPSocket procedure sends data to the [UDP Socket](#).

Applicability

Colour C-Touch only.

Syntax

```
WriteUDPSocket(IPAddress, PortNumber, DataString);
```

IPAddress is a [String](#)

PortNumber is an [Integer](#)

DataString is a String

Description

This sends a UDP message to the specified IP Address and Port Number.

To send a UDP broadcast, use the IP Address '255.255.255.255'.

Note that UDP messages are not guaranteed to receive their destination.

See also [UDP Example](#)

4.27.2.4 ReadUDPSocket Procedure

The ReadUDPSocket procedure reads data from the [UDP Socket](#).

Applicability

Colour C-Touch only.

Syntax

```
ReadUDPSocket(DataString, Terminator, IPAddress);
```

DataString is a [String](#) variable

Terminator is a [String](#) expression

IPAddress is a string variable

Description

This reads received data from the UDP socket. The Terminator is used to determine where one string ends and the next commences. Typically, the Terminator will be a Carriage Return / Line Feed pair. The result is stored in the DataString variable.

If the Terminator is a null string, the whole of the received string will be placed in the DataString.

Received strings are placed in a queue until they are read by the user. If the buffer exceeds 100 items, the newest data will be ignored.

The IP Address of the source of the message is stored in the IPAddress variable.

See also [UDP Example](#)

4.27.2.5 UDPSocketError Function

The UDPSocketError function returns the error status of the UDP Socket.

Applicability

Colour C-Touch only.

Syntax

UDPSocketError

Description

This returns an [Integer](#) value with the most recent Client Socket error message.

Error Number	Meaning
0	No error
1	Read failed
2	Write failed
3	Open failed
4	Close failed
5	Unknown error

Reading this resets the value to 0. So if you need to use the value, you will need to assign it to another variable (see example below).

Example

```

ErrorNumber := UDPSocketError;
if ErrorNumber > 0 then
begin
  format(ErrorString, 'Error number ', ErrorNumber:0);
  ...
end;

```

4.27.2.6 UDPSocketActive Function

The UDPSocketActive function returns whether the [UDP Socket](#) is Active.

Applicability

Colour C-Touch only.

Syntax

```
UDPSocketActive
```

Description

This returns a [boolean](#) value which shows whether the UDP Socket is active or not. The UDP Socket is active if it has been successfully [opened](#). Once the UDP socket has been [closed](#), it is no longer active.

Example

To perform an action only if the UDP socket is active :

```
if UDPSocketActive then ...
```

4.27.2.7 SendWOL Procedure

The SendWOL procedure sends a UDP Wake On LAN (WOL) message.

Applicability

Colour C-Touch only.

Syntax

```
SendWOL(IPAddress, MACAddress, Port);
```

IPAddress is a [String](#)

MACAddress is a String

Port is an integer

Description

This sends a UDP WOL "magic packet" to the specified IP Address, MAC Address and Port (Port number 7 and 9 are most often used). This is used to wake devices which are "sleeping" or "hibernating".

To send as a UDP broadcast, use the IP Address '255.255.255.255'.

Note that UDP messages are not guaranteed to receive their destination.

If the UDP Socket is not already open, the SendWOL will open the UDP Socket first.

Example

To send a WOL as a broadcast to MAC Address 01:23:45:67:89:ab, port 7:

```
SendWOL('255.255.255.255', '01:23:45:67:89:ab', 7);
```

To send the WOL to a specific IP Address (192.168.1.123) so that the messages will go through a router:

```
SendWOL('192.168.1.123', '01:23:45:67:89:ab', 7);
```

To send the WOL to a subnet (192.168.1.x), the subnet broadcast address (192.168.1.255) can be used:

```
SendWOL('192.168.1.255', '01:23:45:67:89:ab', 7);
```

4.27.2.8 UDP Example

For this example, we want to broadcast a UDP message "hello" to port 12345 of all devices on the network, wait for 2 seconds, then display any replies.

Variables `s` and `IPAddress` are strings.

```
{ open the UDP socket }
OpenUDPSocket(12345);

{ send a UDP broadcast on port 12345 }
WriteUDPSocket('255.255.255.255', 12345, 'hello');
delay(2);

repeat
  ReadUDPSocket(s, '', IPAddress);
  if s <> '' then
    WriteLn('Received reply from IP Address ', IPAddress, ' : "', s, '"');
until s = '';

{ close the UDP socket }
CloseUDPSocket;
```

4.27.3 Ping

The "Ping" utility is used to measure the time taken for a message to get from one machine to another and back again. The most common use is to determine whether a device exists at a particular IP Address.

The procedures available for use with ping include:

- [SendPing Procedure](#)
- [GetPingResult Function](#)

4.27.3.1 SendPing Procedure

The `SendPing` procedure sends a "ping" command to an IP Address.

Applicability

Colour C-Touch only.

Syntax

```
SendPing(IPAddress);
```

`IPAddress` is a [String](#)

Description

The `SendPing` command sends a ping message to IP Address. The reply time can be read using

the [GetPingResult Function](#). Make sure a [Delay](#) is used between sending the ping and reading the result to allow time for the transaction to occur.

See also [Ping Example](#)

4.27.3.2 GetPingResult Function

The GetPingResult function returns the result of the [SendPing Procedure](#).

Applicability

Colour C-Touch only.

Syntax

```
GetPingResult
```

Description

The GetPingResult function returns the result in milliseconds. A value of 0 means that no ping reply was received. A value of -1 means that the ping is in progress (waiting for a reply).

See also [Ping Example](#)

4.27.3.3 Ping Example

To find whether there is a device at IP Address 192.168.1.123, you can use a [ping](#) as follows:

```
SendPing('192.168.1.123');

// wait for a reply
repeat
  delay(0.2);
until GetPingResult >= 0;

// show result
if GetPingResult = 0 then
  WriteLn('No device found')
else
  WriteLn('Device found');
```

4.27.4 DNS

The Domain Name System (DNS) is a hierarchical naming system, primarily used for naming web sites, but also used for computers and other services. These names are more convenient for people than IP Addresses. For example, the domain name "google.com" is easier to remember than the corresponding IP Address "66.102.11.104".

The following procedures are available for use with DNS:

- [DNSLookup Procedure](#)
- [GetDNSLookupResult Function](#)
- [GetDNSLookupIPAddress Procedure](#)

4.27.4.1 DNSLookup Procedure

The DNSLookup procedure executes a DNS lookup for the specified domain name.

Applicability

Colour C-Touch only.

Syntax

```
DNSLookup ( DomainName ) ;
```

DomainName is a [String](#)

Description

This looks up the IP Address of DomainName. The result and IP Address can be read using the [GetDNSLookupResult Function](#) and [GetDNSLookupIPAddress Procedure](#) respectively.

Note that the logic engine may pause slightly while the DNS lookup occurs, so this should not be done on a regular basis. It is recommended that the IP Address of a given domain name only be looked up once, and that the IP Address be stored for later use.

See also [DNS Example](#)

4.27.4.2 GetDNSLookupResult Function

The GetDNSLookupResult function returns the error status of the UDP Socket.

Applicability

Colour C-Touch only.

Syntax

```
GetDNSLookupResult
```

Description

This returns an [Integer](#) value with the result of the most recent [DNS Lookup](#).

Value	Meaning
0	No reply yet
1	OK
2	Unknown error
3	Domain Name not found

See also [DNS Example](#)

4.27.4.3 GetDNSLookupIPAddress Procedure

The GetDNSLookupIPAddress procedure returns the IP Address from the most recent [DNSLookup Procedure](#).

Applicability

Colour C-Touch only.

Syntax

```
GetDNSLookupIPAddress ( IPAddress ) ;
```

IPAddress is a [String](#) variable

Description

This writes the IP Address from the DNS Lookup into variable IPAddress.

See also [DNS Example](#)

4.27.4.4 DNS Example

In this example, we want to look up the IP Address of "google.com":

```
DNSLookup('google.com');
delay(1);
if GetDNSLookupResult = 1 then
begin
  GetDNSLookupIPAddress(IPAddress);
  WriteLn('Google.com found at ', IPAddress);
end
else
  WriteLn('Google.com not found');
```

4.27.5 HTTP Data

In addition to being able to communicate via [TCP/IP](#), there are procedures included specifically for dealing with HTTP data:

- [GetHTTPData Procedure](#): this initiates the retrieval of HTTP data
- [ReadHTTPData Procedure](#): this reads the retrieved data into logic
- [PostHTTPData Procedure](#): this posts HTTP data
- [ReadHTTPPostData Procedure](#): this reads the result from an HTTP Post

4.27.5.1 GetHTTPData Procedure

The GetHTTPData procedure reads data from a web site and stores it in a buffer for use by the [ReadHTTPData Procedure](#).

Applicability

Colour C-Touch only.

Syntax

```
GetHTTPData(URL);
```

URL is a [String](#)

Description

The GetHTTPData procedure uses an HTTP "get" command to read data from the given URL and stores this data in a buffer. Since the HTTP command may take some time to execute, this happens in the background so that the logic engine can perform other tasks.

4.27.5.2 ReadHTTPData Procedure

The ReadHTTPData procedure reads data obtained from a web site using [GetHTTPData Procedure](#).

Applicability

Colour C-Touch only.

Syntax

```
ReadHTTPData(DataString);
```

DataString is a [String](#) variable

Description

The ReadHTTPData procedure reads data stored in a buffer by the [GetHTTPData](#) procedure and stores it in the DataString variable.

Example

In this example, a web site <http://MyWeatherForecast.com/index.html> returns the following HTML data:

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en-AU" lang="en">
  <head>
    <title>Forecast Summary</title>
  </head>
  <body>
    Today's forecast is sunny with showers in the evening
  </body>
</html>
```

We want to retrieve this data using the GetHTTPData procedure, then retrieve the forecast (the text immediately after "forecast is"). The following code will do this:

```
// get the data
GetHTTPData('http://MyWeatherForecast.com/index.html');
// wait for data to come back
delay(5);
// read the data into a string
ReadHTTPData(DataString);
if DataString <> '' then
begin
  // Find where the data starts and ends
  DataStart := pos('forecast is', DataString) + 12;
  DataEnd := pos('</body>', DataString);
  // extract the data into the string ForecastString
  Copy(ForecastString, DataString, DataStart, DataEnd - DataStart);
end;
```

4.27.5.3 PostHTTPData Procedure

The PostHTTPData procedure sends data to a web server as part of the request and stores the response in a buffer for use by the [ReadHTTPPostData Procedure](#).

Applicability

Colour C-Touch only.

Syntax

```
PostHTTPData(URL, Data);
```

URL is a [String](#)
 Data is a string

Description

The PostHTTPData procedure uses an HTTP "post" command to send data to the given URL and stores the response data in a buffer. Since the HTTP command may take some time to execute, this happens in the background so that the logic engine can perform other tasks.

4.27.5.4 ReadHTTPPostData Procedure

The ReadHTTPPostData procedure reads data obtained from a web site using [PostHTTPData Procedure](#).

Applicability

Colour C-Touch only.

Syntax

```
ReadHTTPPostData(DataString);
```

DataString is a [String](#) variable

Description

The ReadHTTPPostData procedure reads data stored in a buffer by the [PostHTTPData Procedure](#) and stores it in the DataString variable.

Example

In this example, we post data to the web site <http://www.snee.com/xml/crud/posttest.html>

We want to retrieve the data using the GetHTTPPostData procedure. The following code will do this:

```
// make the post request
PostHTTPData('http://www.snee.com/xml/crud/posttest.cgi',
'fname=John&lname=Doe');
// wait for data to come back
delay(5);
// read the data into a string
ReadHTTPPostData(DataString);
if DataString <> '' then
begin
    // Find where the data starts and ends
    DataStart := pos('First name: ', DataString) + 12;
    DataEnd := pos('</p>', DataString);
    // extract the data into the string TheFirstNameString
    Copy(TheFirstNameString, DataString, DataStart, DataEnd - DataStart);
    WriteLn(TheFirstNameString);
end;
```

4.27.6 E-Mail

E-Mail accounts can be configured using the E-Mail Manager. The data retrieved by the E-Mail Manager can be accessed using the following functions :

- [DeleteEMail Procedure](#)

- [GetEMailBodyLineCount Function](#)
- [GetEMailBodyLine Procedure](#)
- [GetEMailCount Function](#)
- [GetEMailSender Procedure](#)
- [GetEMailSubject Procedure](#)
- [SendEMail Procedure](#)

E-Mail can be read using POP3 and sent using SMTP. Only unencrypted accounts can be used within Logic Engine.

Wiser 2 can send and receive emails using secure communications on SMTP TLS port 587 and POP3 port 995.

To implement secure email in a Wiser 2 project simply set the account SMTP port to 587 and the POP3 port to 995.

Note: Wiser 2 is not able to send email using secure Yahoo or secure Hotmail accounts.

4.27.6.1 GetEMailCount Function

The GetEMailCount function returns the number of E-Mails in the E-Mail account.

Applicability

Colour C-Touch and Wiser Home Control only.

Syntax

```
GetEMailCount (AccountNumber )
```

Description

This returns the number of E-Mail messages in the E-Mail account which have not yet been downloaded from the server..

Example

To perform an action if there are E-Mail messages present in the first E-Mail account (number 0) :

```
if GetEMailCount(0) > 0 then ...
```

4.27.6.2 GetEMailBodyLineCount Function

The GetEMailBodyLineCount function returns the number of lines of text in an E-Mail message.

Applicability

Colour C-Touch only.

Syntax

```
GetEMailBodyLineCount (AccountNumber , MessageNumber )
```

Description

This returns the number of E-Mail messages in the specified E-Mail message. Note that this is only valid if the E-Mail account in the E-Mail Manager has been configured to download the body of the E-Mail messages.

Example

```
To process each line of text in the first message (number 0) of the first E-Mail account (number 0) :
for LineNo := 0 to GetEMailBodyLineCount(0, 0) - 1 do
begin
  GetEMailBodyLine(0, 0, LineNo, s);
  ...
end;
```

4.27.6.3 GetEMailAddress Procedure

The GetEMailAddress function returns the E-Mail address of the sender of an E-Mail message.

Applicability

Colour C-Touch only.

Syntax

```
GetEMailAddress(AccountNumber, MessageNumber, Sender);
```

AccountNumber is an [Integer](#)
 MessageNumber is an integer
 Sender is a [String](#) Variable

Description

This stores the E-Mail address of the sender of a specified E-Mail message in a string variable. Note that this is only valid if the E-Mail account in the E-Mail Manager has been configured to download the sender of the E-Mail messages.

Example

To store the sender's e-mail address of the first message (number 0) of the first E-Mail account (number 0) in string variable SenderAddress :

```
GetEMailAddress(0, 0, SenderAddress);
```

4.27.6.4 GetEMailSender Procedure

The GetEMailSender function returns the name of the sender of an E-Mail message.

Applicability

Colour C-Touch and Wiser Home Control only.

Syntax

```
GetEMailSender(AccountNumber, MessageNumber, Sender);
```

AccountNumber is an [Integer](#)
 MessageNumber is an integer
 Sender is a [String](#) Variable

Description

This stores the sender of a specified E-Mail message in a string variable. Note that this is only valid if the E-Mail account in the E-Mail Manager has been configured to download the sender of the E-

Mail messages.

Note that in Logic Engine version 4.9.2 or earlier, this procedure returned the name and e-mail address of the sender in a single string. The E-Mail address of the Sender is now obtained using the [GetEmailAddress Procedure](#).

Example

To store the sender of the first message (number 0) of the first E-Mail account (number 0) in string variable SenderName :

```
GetEmailSender(0, 0, SenderName);
```

4.27.6.5 GetEmailSubject Procedure

The GetEmailSubject function returns the Subject of an E-Mail message.

Applicability

Colour C-Touch and Wiser Home Control only.

Syntax

```
GetEmailSubject(AccountNumber, MessageNumber, Subject);
```

AccountNumber is an [Integer](#)

MessageNumber is an integer

Subject is a [String](#) Variable

Description

This stores the subject of a specified E-Mail message in a string variable. Note that this is only valid if the E-Mail account in the E-Mail Manager has been configured to download the subject of the E-Mail messages.

Example

To store the subject of the first message (number 0) of the first E-Mail account (number 0) in string variable SubjectName :

```
GetEmailSubject(0, 0, SubjectName);
```

4.27.6.6 GetEmailBodyLine Procedure

The GetEmailBodyLine function returns a line of text from the body of an E-Mail message.

Applicability

Colour C-Touch only.

Syntax

```
GetEmailBodyLine(AccountNumber, MessageNumber, LineNumber, Data);
```

AccountNumber is an [Integer](#)

MessageNumber is an integer

LineNumber is an integer

Data is a [String](#) Variable

Description

This stores the content of a specified line of text of a specified E-Mail message in a string variable. Note that this is only valid if the E-Mail account in the E-Mail Manager has been configured to download the body of the E-Mail messages.

Example

To store the first line of text (number 0) of the first message (number 0) of the first E-Mail account (number 0) in string variable Line Text :

```
GetEMailBodyLine(0, 0, 0, LineText);
```

4.27.6.7 DeleteEMail Procedure

The DeleteEMail procedure deletes an E-Mail message.

Applicability

Colour C-Touch only.

Syntax

```
DeleteEMail(AccountNumber, MessageNumber);
```

AccountNumber is an [Integer](#)

MessageNumber is an integer

Description

This deletes a specified E-Mail message from the E-Mail server. Note that this is only valid if the E-Mail account in the E-Mail Manager has been configured to download Sender/Subject and/or body of the E-Mail messages.

Example

To delete the first message (number 0) of the first E-Mail account (number 0) :

```
DeleteEMail(0, 0);
```

4.27.6.8 SendEMail Procedure

The SendEMail procedure sends an E-Mail message.

Applicability

Colour C-Touch and Wiser Home Control only.

Syntax

```
SendEMail(AccountNumber, EMailAddress, Subject, Body);
```

AccountNumber is an [Integer](#)

EMailAddress is a [String](#)

Subject is a string

Body is a string

Description

This sends an E-Mail using the specified E-Mail account. Only a single line of text is possible,

Example

To send a test E-Mail using the first E-Mail account (number 0) :

```
SendEMail(0, 'username@domainname.com', 'test', 'this is a test message');
```

4.27.7 Network Adaptors

Network Adaptors are used to connect a computer to a network. A computer may have more than one Network Adaptor, for example, one for Ethernet (LAN) and one for Wireless (WiFi).

The procedures for use with Network Adaptors are:

- [GetNetworkAdaptorCount Function](#)
- [GetIPAddress Procedure](#)

4.27.7.1 GetNetworkAdaptorCount Function

The GetNetworkAdaptorCount function returns the number of [Network Adaptors](#).

Applicability

Colour C-Touch only.

Syntax

```
GetNetworkAdaptorCount
```

Description

This returns an integer value with the number of Network Adaptors on the computer.

Example

To perform an action only if there are one or more Network Adaptors :

```
if GetNetworkAdaptorCount > 0 then ...
```

4.27.7.2 GetIPAddress Procedure

The GetIPAddress procedure returns the IP Address of a given [Network Adaptor](#).

Applicability

Colour C-Touch only.

Syntax

```
GetIPAddress(NetworkAdaptorNumber, IPAddress);
```

NetworkAdaptorNumber is an [Integer](#)

IPAddress is a [String](#) variable

Description

This writes the IP Address for network adaptor NetworkAdaptorNumber into variable IPAddress.

Example

To store the IP Address of the first network adaptor in variable MyIPAddress :

```
GetIPAddress(1, MyIPAddress);
```

4.28 Page Properties



It is possible to change some properties of Pages. The following properties can be set:

Property Name	Type	Property Data	Comments
Access Level	Integer	Level as per Access Control Manager	Zero based numbering
Background Colour	Integer	See Colours	
Background Gradient Colour	Integer	See Colours	
Theme Page	Integer	Theme page used by the page	Zero based numbering
Time-out Duration	Integer	Page time-out	Time is in seconds 0 = default -1 = no time-out

The following functions can be used with Page Properties:

- [GetPageIntegerProp Function](#)
- [SetPageIntegerProp Procedure](#)

Notes

 Setting Page properties should be done with caution. Setting properties of Pages from logic bypasses some of the checks normally imposed by Logic Engine, so it is possible to cause unexpected behaviour.

4.28.1 GetPageIntegerProp Function

The GetPageIntegerProp Function gets an [integer](#) property of a page.

Applicability

Colour C-Touch only.

Syntax

```
GetPageIntegerProp(PageNumber, PropertyNumber)
```

PageNumber is a Page [Tag](#) or number ([Integer, zero based](#))

Property is a [Page Property Name](#) Tag

Description

This function returns an integer [property](#) of a Page. It can be used with logic to make the appearance or operation of a Project change.

Example

To get the current access level for page called "Tools":

```
Level := GetPageIntegerProp("Tools", "Access Level");
```

4.28.2 SetPageIntegerProp Procedure

The SetPageIntegerProp Procedure sets an [integer](#) property of a page.

Applicability

Colour C-Touch only.

Syntax

```
SetPageIntegerProp(PageNumber, PropertyNumber, NewValue)
```

PageNumber is a Page [Tag](#) or number ([Integer](#), [zero based](#))

Property is a [Page Property Name](#) Tag

NewValue is an [integer](#)

Description

This function sets an integer [property](#) of a Page to a new value. It can be used with logic to make the appearance or operation of a Project change. This procedure should be used with care.

Example

To set the background colour of a page called "Main" to blue:

```
SetPageIntegerProp("Main", "Background Colour", clBlue);
```

4.29 Component Properties



It is possible to change some properties of Components. The following properties can be set:

Property Name	Type	Property Data	Comments
Alpha Blend Active	Integer	Alpha Blend level	0 to 100%
Alpha Blend Inactive	Integer	Alpha Blend level	0 to 100%
Background Colour Active	Integer	Active colour of background	See Colours
Background Colour Inactive	Integer	Inactive colour of background (if used)	See Colours
Background Style Active	Integer	Active fill style of background	See Brush Style
Background Style Inactive	Integer	Inactive fill style of background (if used)	See Brush Style
Border Colour Active	Integer	Active colour of border	See Colours
Border Colour Inactive	Integer	Inactive colour of border (if used)	See Colours
Border Shape	Integer	Shape of border	See below
Border Style Active	Integer	Active style of border	See Pen Styles
Border Style Inactive	Integer	Inactive style of border (if used)	See Pen Styles
Border Width	Integer	Width of the border	Must be 1 or more
C-Bus Application	Integer	C-Bus Application to be controlled	
C-Bus Duration	Integer	Pulse duration	Seconds (0 = no pulse)
C-Bus Group	Integer	C-Bus Group Address to be	The SetCompCBusProp

		controlled	Procedure can be used with C-Bus tags
C-Bus Level	Integer	Level to be set	The SetCompCBusProp Procedure can be used with C-Bus tags
C-Bus Network	Integer	C-Bus Network to be controlled	
C-Bus Ramp Rate	Integer	Ramp rate to be used	Seconds (0 = instantaneous) Only valid ramp rates are possible.
Current Sub-Page	Integer	Sub-Page currently displayed in a frame	
Default Sub-Page	Integer	Default Sub-Page for a frame	
Font Colour Active	Integer	Active colour of font	See Colours
Font Colour Inactive	Integer	Inactive colour of font (if used)	See Colours
Font Name Active	Integer	Active name of font	
Font Name Inactive	Integer	Inactive name of font (if used)	
Font Size Active	Integer	Active size of font	
Font Size Inactive	Integer	Inactive size of font (if used)	
Font Style Active	Integer	Active style of font	See Font Styles
Font Style Inactive	Integer	Inactive style of font (if used)	See Font Styles
Graph Title	String	Title of the graph	
Graph X Axis Label	String	X axis label of graph	
Graph X Axis Point Interval	Integer	Interval between points on X axis	Time is in seconds
Graph X Axis Tick Interval	Integer	Interval between ticks on X axis	Time is in seconds
Graph Y Axis Max	Real	Minimum Y axis value	
Graph Y Axis Min	Real	Maximum Y axis value	
Graph Y Axis Tick Interval	Real	Interval between ticks on Y axis	
Height	Integer	Height of Component	
HTML Refresh Rate	Integer	Refresh rate for the Web Page	Time is in seconds
HTML URL	String	Web page URL	
IB System IO Number	Integer	In-built system IO variable used by the component	Use tags. See In-Built System IO Variables
Image Active	Integer	Component Active custom image	Use tags. See DrawImage Procedure
Image Inactive	Integer	Component Inactive custom image (if used)	Use tags. See DrawImage Procedure
Left	Integer	Left position of Component	
Page Link	Integer	Page linked by this Component	Use Tags
Postfix Text	String	Postfix text used by custom value	
Prefix Text	String	Prefix text used by custom value	
Scene	Integer	Scene Number for Component	Use Tags
Shadow	Integer	Shadow Level	0 to 100%
Slider Bar Colour Active	Integer	Active colour of slider bar	See Colours
Slider Bar Colour Inactive	Integer	Inactive colour of slider bar (if used)	See Colours
Slider Slot Background Colour	Integer	Background colour of slider slot	See Colours
Slider Slot Border Colour	Integer	Border colour of slider slot	See Colours
Slider Thumb Background Colour	Integer	Background colour of slider thumb	See Colours
Slider Thumb Border Colour	Integer	Border colour of slider thumb	See Colours
Special Function	Integer	Component Special Function	Use Tags
Sub-Page Link	Integer	Component sub-page link	-1 = no sub-page link
System IO Number	Integer	User system IO variable used	Use tags. See User System IO

Text Active	String	by the component Active text on Component	Variables Multi-line text uses #13 characters to separate the lines
Text Inactive	String	Inactive text on Component (if used)	Multi-line text uses #13 characters to separate the lines
Top	Integer	Top position of Component	
Visible	Boolean	Controls if component can be seen	
Webcam Refresh Rate	Real	Refresh rate for web cam	Time is in seconds
Webcam URL	Integer	URL for web cam	
Width	Integer	Width of Component	

The following functions can be used with Component Properties:

- [GetCompBooleanProp Function](#)
- [GetCompIntegerProp Function](#)
- [GetCompRealProp Function](#)
- [GetCompStringProp Procedure](#)
- [GetCompType Function](#)
- [GetPageCompCount Function](#)
- [SetCompBooleanProp Procedure](#)
- [SetCompCBusProp Procedure](#)
- [SetCompIntegerProp Procedure](#)
- [SetCompRealProp Procedure](#)
- [SetCompStringProp Procedure](#)
- [ShowingSubPage Function](#)
- [ShowSubPage Procedure](#)

Notes

The following border shapes can be used with the "Border Shape" property:

Component Shape	Number	Comment
Rectangle	0	
Round Rectangle	1	
Capsule	2	
Polygon	3	
Circle/Ellipse	4	
Triangle (Right)	5	
Triangle (Left)	6	
Triangle (Up)	7	
Triangle (Down)	8	
Pointer (Left)	9	
Pointer (Right)	10	
Pointer (Left & Right)	11	
Pointer (Up)	12	
Pointer (Down)	13	
Pointer (Up & Down)	14	
Star	15	It is not a good idea to change to or from this shape in logic
Line	16	It is not a good idea to change to or from this shape in logic
Arbitrary	17	It is not a good idea to change to or from this shape in logic

It is best to give Components a Name for use with these functions.

Many properties only apply to some [Component Types](#).

 Setting Component properties should be done with caution. Setting properties of Components from logic bypasses some of the checks normally imposed by Logic Engine, so it is possible to cause unexpected behaviour.

4.29.1 GetCompBooleanProp Function

The GetCompBooleanProp Function gets a [Boolean](#) property of a Component.

Applicability

Colour C-Touch only.

Syntax

```
GetCompBooleanProp(PageNumber, CompNumber, PropertyNumber)
```

PageNumber is a Page name [Tag](#) or number ([Integer](#), [zero based](#))

CompNumber is a Component name [Tag](#) ([Integer](#), [zero based](#))

Property is a [Component Property Name](#) Tag

Description

This function returns a boolean [property](#) of a Component on a Page. It can be used with logic to make the appearance or operation of a Project change.

Example

To perform an action if the component called "Next" on a page called "Tools" is visible:

```
if GetCompBooleanProp("Tools", "Next", "Visible") then...
```

4.29.2 GetCompIntegerProp Function

The GetCompIntegerProp Function gets an [integer](#) property of a Component.

Applicability

Colour C-Touch only.

Syntax

```
GetCompIntegerProp(PageNumber, CompNumber, PropertyNumber)
```

PageNumber is a Page name [Tag](#) or number ([Integer](#), [zero based](#))

CompNumber is a Component name [Tag](#) or number ([Integer](#), [zero based](#))

Property is a [Component Property Name](#) Tag

Description

This function returns an integer [property](#) of a Component on a Page. It can be used with logic to make the appearance or operation of a Project change.

Example

To get the page link of a component called "Next" on a page called "Tools":

```
NextPage := GetCompIntegerProp("Tools", "Next", "Page Link");
```

4.29.3 GetCompRealProp Function

The GetCompRealProp Function gets a [Real](#) property of a Component.

Applicability

Colour C-Touch only.

Syntax

```
GetCompRealProp(PageNumber, CompNumber, PropertyNumber)
```

PageNumber is a Page name [Tag](#) or number ([Integer, zero based](#))

CompNumber is a Component name [Tag](#) or number ([Integer, zero based](#))

Property is a [Component Property Name](#) Tag

Description

This function returns a real [property](#) of a Component on a Page. It can be used with logic to make the appearance or operation of a Project change.

Example

To get the maximum of the Y axis (not the maximum data point) of a component called "Power Graph" on a page called "Main":

```
Max := GetCompRealProp("Main", "Power Graph", "Graph Y Axis Max");
```

4.29.4 GetCompStringProp Procedure

The GetCompStringProp Procedure gets a string property of a Component.

Applicability

Colour C-Touch only.

Syntax

```
GetCompStringProp(PageNumber, CompNumber, PropertyNumber, StringVariable);
```

PageNumber is a Page name [Tag](#) or number ([Integer, zero based](#))

CompNumber is a Component name [Tag](#) or number ([Integer, zero based](#))

Property is a [Component Property Name](#) Tag

StringVariable is a [String](#) variable where the result is to be stored

Description

This function returns a string [property](#) of a Component on a Page. It can be used with logic to make the appearance or operation of a Project change.

Example

To get the active text of a component called "Main Light" on a page called "Timeout":

```
GetCompStringProp("Timeout", "Main Light", "Text Active", s);
```

If the component had two lines of text, "Light" and "On", then the variable s will now contain the text "Light" and "On" separated by a carriage return character (#13).

4.29.5 GetCompType Function

The GetCompType Function gets the type of a Component.

Applicability

Colour C-Touch only.

Syntax

```
GetCompType( PageNumber , CompNumber )
```

PageNumber is a Page name [Tag](#) or number ([Integer](#), [zero based](#))

CompNumber is a Component name [Tag](#) or number ([Integer](#), [zero based](#))

Description

This function returns an integer value, the type, of a Component on a Page. It can be used with logic to make the appearance or operation of a Project change.

The Component types are:

Component Type	Value	Comment
Button	1	
Shape	2	
Text	3	
Image	4	Includes Web Cam components
Clock	5	
Slider	7	Includes bar graph Components
Level	8	
Monitor	9	
Selector	10	
HTML	11	
Graph	12	
Calendar	13	

Example

To set the active colour of all button Components on a page called "Tools" to yellow:

```
for i := 1 to GetPageCompCount("Tools") do
begin
  if GetCompType("Tools", i - 1) = 1 then // Only do this for button
Components
begin
  SetCompIntegerProp("Tools", i - 1, "Background Colour Active",
clYellow);
end;
end;
```

4.29.6 GetPageCompCount Function

The GetPageCompCount Function gets the number of Components on a Page.

Applicability

Colour C-Touch only.

Syntax

```
GetPageCompCount ( PageNumber )
```

PageNumber is a Page name [Tag](#) or number ([Integer](#), [zero based](#))

Description

This function returns an integer value, the number Components on a Page. It can be used with logic to make the appearance or operation of a Project change.

Note that this does not include components on the page's theme page. So if a page displays 7 components, and 4 of them are on the theme page, the GetPageCompCount function will return a value of 3.

See [GetCompType Example](#)

4.29.7 SetCompBooleanProp Procedure

The SetCompBooleanProp Procedure sets a [Boolean](#) property of a Component.

Applicability

Colour C-Touch only.

Syntax

```
SetCompBooleanProp(PageNumber, CompNumber, PropertyNumber, NewValue);
```

PageNumber is a Page name [Tag](#) or number ([Integer](#), [zero based](#))

CompNumber is a Component name [Tag](#) or number ([Integer](#), [zero based](#))

Property is a [Component Property Name](#) Tag

NewValue is a [Boolean](#) value

Description

This function sets the value of a boolean [property](#) of a Component on a Page. It can be used with logic to make the appearance or operation of a Project change. This procedure should be used with care.

Example

To make the component called "Next" on a page called "Tools" invisible:

```
SetCompBooleanProp("Tools", "Next", "Visible", false);
```

4.29.8 SetCompCBusProp Procedure

The SetCompCBusProp Procedure sets a [C-Bus](#) properties of a Component.

Applicability

Colour C-Touch only.

Syntax

```
SetCompCBusProp(PageNumber, CompNumber, Network, Application, Group, Level);
```

PageNumber is a Page name [Tag](#) or number ([Integer](#), [zero based](#))

CompNumber is a Component name [Tag](#) or number ([Integer](#), [zero based](#))

Network is a C-Bus Network or [Tag](#)

Application is a C-Bus Application or [Tag](#)

Group is a C-Bus Group Address or [Tag](#)

Level is a C-Bus Level or [Tag](#)

Description

This function sets the C-Bus properties of a Component on a Page. It can be used with logic to make the appearance or operation of a Project change. This procedure should be used with care.

Although the C-Bus Properties can be set with the [SetCompIntegerProp Procedure](#), it is simpler to use the SetCompCBusProp Procedure as all properties are set at once and tags can be used for the group address and level.

Example

To make the component called "Scene" on a page called "Main" trigger a different Scene called "Relaxing" by setting a new level:

```
SetCompCBusProp("Main", "Scene", "Local", "Trigger", "My Scenes", "Relax Scene");
```

4.29.9 SetCompIntegerProp Procedure

The SetCompIntegerProp Procedure sets an [Integer](#) property of a Component.

Applicability

Colour C-Touch only.

Syntax

```
SetCompIntegerProp(PageNumber, CompNumber, PropertyNumber, NewValue);
```

PageNumber is a Page name [Tag](#) or number ([Integer](#), [zero based](#))

CompNumber is a Component name [Tag](#) or number ([Integer](#), [zero based](#))

Property is a [Component Property Name](#) Tag

NewValue is an [Integer](#) value

Description

This function sets the value of an Integer [property](#) of a Component on a Page. It can be used with logic to make the appearance or operation of a Project change. This procedure should be used with care.

Example

To make the component called "Scene" on a page called "Main" control the Scene called "Relaxing":

```
SetCompIntegerProp("Tools", "Next", "Scene", "Relaxing");
```

4.29.10 SetCompRealProp Procedure

The SetCompRealProp Procedure sets a [Real](#) property of a Component.

Applicability

Colour C-Touch only.

Syntax

```
SetCompRealProp(PageNumber, CompNumber, PropertyNumber, NewValue);
```

PageNumber is a Page name [Tag](#) or number ([Integer](#), [zero based](#))

CompNumber is a Component name [Tag](#) or number ([Integer](#), [zero based](#))

Property is a [Component Property Name](#) Tag

NewValue is a [Real](#) value

Description

This function sets the value of a Real [property](#) of a Component on a Page. It can be used with logic to make the appearance or operation of a Project change. This procedure should be used with care.

Example

To change the scale on a graph Component called "Outside Temp" on a page called "Main" to be from 10C to 30C:

```
SetCompRealProp("Main", "Outside Temp", "Graph Y Axis Min", 10);
```

```
SetCompRealProp("Main", "Outside Temp", "Graph Y Axis Max", 30);
```

4.29.11 SetCompStringProp Procedure

The SetCompStringProp Procedure sets a [String](#) property of a Component.

Applicability

Colour C-Touch only.

Syntax

```
SetCompStringProp(PageNumber, CompNumber, PropertyNumber, NewValue);
```

PageNumber is a Page name [Tag](#) or number ([Integer](#), [zero based](#))

CompNumber is a Component name [Tag](#) or number ([Integer](#), [zero based](#))

Property is a [Component Property Name](#) Tag

NewValue is a [String](#) value

Description

This function sets the value of a String [property](#) of a Component on a Page. It can be used with logic to make the appearance or operation of a Project change. This procedure should be used with care.

Example

To change the active text on a component called "Lights" on a page called "Main" to "Outside":

```
SetCompStringProp("Main", "Lights", "Text Active", 'Outside');
```

To change the active text on a component called "Lights" on a page called "Main" to have two lines, "Outside" and "Lights":

```
SetCompStringProp("Main", "Lights", "Text Active", 'Outside'#13'Lights');
```

4.29.12 ShowSubPage Procedure

The ShowSubPage procedure shows a Sub-Page in a Sub-Page Frame.

Applicability

Colour C-Touch only.

Syntax

```
ShowSubPage(PageNo, CompNo, SubPageNo);
```

PageNo is an [Integer](#) or Page name [Tag](#)

CompNo is an [Integer](#) or Component name [Tag](#)

SubPageNo is an [Integer](#) or Sub-Page name [Tag](#)

Description

The ShowSubPage procedure shows the selected Sub-Page in a Sub-Page Frame. Note that the first Page and Component number is [0, not 1](#).

Note that the same result can be achieved using the [SetComplIntegerProp Procedure](#) with the Current Sub-Page property.

Example

To display the Sub-Page called "Video" in a Sub-Page Frame called "Control Frame" on a page called "AV Control" :

```
ShowSubPage("AV Control", "Control Frame", "Video");
```

See also [ShowPage Procedure](#) and [ShowingSubPage Function](#)

4.29.13 ShowingSubPage Function

The ShowingSubPage function returns whether a Sub-Page is showing in a Sub-Page Frame.

Applicability

Colour C-Touch only.

Syntax

```
ShowingSubPage(PageNo, CompNo, SubPageNo)
```

PageNo is an [Integer](#) or Page name [Tag](#)

CompNo is an [Integer](#) or Component name [Tag](#)

SubPageNo is an [Integer](#) or Sub-Page name [Tag](#)

Description

The ShowingSubPage function returns whether the selected Sub-Page is currently being displayed in the Sub-Page Frame. Note that the first page and component number is [0, not 1](#).

Note that the same result can be achieved using the [GetComplIntegerProp Function](#) with the Current Sub-Page property and comparing it with the number of the Sub-Page.

Example

To perform an action if the Sub-Page called "Video" is being displayed in a Sub-Page Frame called "Control Frame" on a page called "AV Control" :

```
if ShowingSubPage("AV Control", "Control Frame", "Video") then ...
```

See also [ShowingPage Function](#) and [ShowSubPage Procedure](#)

4.30 Profiles

Profiles can be used to change Logic behaviour depending on which device the Logic is running in. The functions provided for use with profiles include:

- [GetProfile Function](#)
- [ProfileIsSet Function](#)
- [SetProfile Procedure](#)

4.30.1 GetProfile Function

The GetProfile Function returns the number of the currently selected [Profile](#).

Syntax

```
GetProfile
```

Description

This function returns an integer value which is the number of the current Profile. Note that the index of the first Profile is [0, not 1](#).

Example

To perform an action only if the first Profile is selected:

```
if GetProfile = 0 then
begin
  { do something here }
end;
```

See also [ProfileIsSet Function](#)

4.30.2 ProfileIsSet Function

The ProfileIsSet Function returns whether a [Profile](#) or Profile Group is currently set.

Syntax

```
ProfileIsSet(ProfileNo)
```

ProfileNo is an integer or Profile or Profile Group [Tag](#)

Result is [boolean](#)

Description

This function returns whether the Profile or Profile Group is currently set. It is recommended that tags be used for ProfileNo as it is actually a bit mask as shown in the table below:

Profile 1	Profile 2	Profile 3	Profile No
x	x	x	0 (Binary 00000000)

✓	x	x	1 (Binary 00000001)
x	✓	x	2 (Binary 00000010)
✓	✓	x	3 (Binary 00000011)
x	x	✓	4 (Binary 00000100)
✓	x	✓	5 (Binary 00000101)
x	✓	✓	6 (Binary 00000110)
✓	✓	✓	7 (Binary 00000111)

Example

To perform an action only if Profile "Room 2" is set for this project:

```

if ProfileIsSet("Room 2") then
begin
  { do something here }
end;
```

See also [GetProfile Function](#)

4.30.3 SetProfile Procedure

The SetProfile procedure sets the current [Profile](#).

Applicability

Colour C-Touch only.

Syntax

```
SetProfile(ProfileNumber);
```

ProfileNumber is an integer or profile [Tag](#)

Description

This sets the current Profile to be used.

Example

To set profile "Room 2" to be used:

```
SetProfile("Room 2");
```

4.31 Media Transport Control



Many aspects of the Media Transport Control Application can be accessed via the [ExecuteSpecialFunction Procedure](#) and [In-Built System IO Variables](#).

There are some additional functions and procedures for use with using the logic engine:

- [GetTransportControlData Procedure](#)
- [TransportControlData Procedure](#)
- [TransportControlDataCount Function](#)

- [TransportControlDataMLG Function](#)
- [TransportControlDataStart Function](#)
- [TransportControlDataType Function](#)
- [TransportControlDataValid Function](#)
- [TransportControlFlag Function](#)

The method of using these functions is as follows:

1. Use the [GetTransportControlData Procedure](#) to command the media server to send the required data.
2. Wait until all of the data has been returned. The [TransportControlDataValid Function](#) will tell you this.
3. Use the [TransportControlDataCount Function](#) to know how many data items there are (0 to 15).
4. Use the [TransportControlData Procedure](#) to get the actual data.

For details of the operation of the Transport Control Application, refer to the C-Bus Concepts document.

4.31.1 GetTransportControlData Procedure

The GetTransportControlData Procedure initiates the retrieval of all Media Transport Control Data from a media server.

Applicability

Colour C-Touch and C-Touch Mark 2 only.

Syntax

```
GetTransportControlData(MLG, DataType, StartIndex);
```

MLG is an integer for Media Link Group [Tag](#)
 DataType and StartIndex are integers

Description

This sends a command to the media server controlling Media Link Group MLG to send a particular type of text data. Only 15 text items can be sent each time, so you also need to set the starting index (first item is 0). The values for DataType are:

Value	Meaning	Comment
0	Category	Retrieving a list of Media Categories
1	Selection	Retrieving a list of Selections
2	Track	Retrieving a list of Tracks

 This procedure should be used with great care. Excessive use of this will cause a lot of C-Bus traffic and may interfere with the normal operation of the C-Bus system.

The data returned by the media server can be accessed using:

- [TransportControlData Procedure](#)
- [TransportControlDataCount Function](#)
- [TransportControlDataMLG Function](#)
- [TransportControlDataStart Function](#)
- [TransportControlDataType Function](#)
- [TransportControlDataValid Function](#)

Example

To initiate the retrieval of the first sources from Media Link Group 2:

```
GetTransportControlData(2, 0, 0);
```

4.31.2 TransportControlData Procedure

The TransportControlData Procedure gets some [Media Transport Control](#) Data, previously returned from a media server.

Applicability

Colour C-Touch and C-Touch Mark 2 only.

Syntax

```
TransportControlData(Index, s);
```

Index is an integer (0 to 14)

s is a string variable

Description

The TransportControlData procedure gets an item of data (previously requested using [GetTransportControlData Procedure](#)) and stores it in a string variable.

Example

To get the first piece of Media Transport Control data and store it in a string called TheSource:

```
TransportControlData(0, TheSource);
```

4.31.3 TransportControlDataType Function

The TransportControlDataType Function returns the type of [Media Transport Control](#) Data, previously returned from a media server.

Applicability

Colour C-Touch and C-Touch Mark 2 only.

Syntax

```
TransportControlDataType
```

Description

The TransportControlDataType function returns the type of the data (previously requested using [GetTransportControlData Procedure](#)). See [GetTransportControlData Procedure](#) for details of the data type values.

Example

To check if the Media Transport Control data is a list of sources:

```
if TransportControlDataType = 0 then...
```

4.31.4 TransportControlDataStart Function

The TransportControlDataStart Function returns the start index of [Media Transport Control](#) Data, previously returned from a media server.

Applicability

Colour C-Touch and C-Touch Mark 2 only.

Syntax

```
TransportControlDataStart
```

Description

The TransportControlDataStart function returns the start index of the data (previously requested using [GetTransportControlData Procedure](#)).

Example

To assign the start index of the Media Transport Control data to variable StartIndex:

```
StartIndex := TransportControlDataStart;
```

4.31.5 TransportControlDataMLG Function

The TransportControlDataMLG Function returns the Media Link Group of the [Media Transport Control](#) Data, previously returned from a media server.

Applicability

Colour C-Touch and C-Touch Mark 2 only.

Syntax

```
TransportControlDataMLG
```

Description

The TransportControlDataMLG function returns the Media Link Group of the data (previously requested using [GetTransportControlData Procedure](#)).

Example

To assign the Media Link Group of the Media Transport Control data to variable CurrentMLG:

```
CurrentMLG := TransportControlDataMLG;
```

4.31.6 TransportControlDataCount Function

The TransportControlDataCount Function returns the number of [Media Transport Control](#) Data items, previously returned from a media server.

Applicability

Colour C-Touch and C-Touch Mark 2 only.

Syntax

TransportControlDataCount

Description

The TransportControlDataCount function returns the number of data items (previously requested using [GetTransportControlData Procedure](#)). It will have a value of between 0 and 15.

Example

To iterate through all of the Media Transport Control data items:

```
for index := 0 to TransportControlDataCount - 1 do
begin
  TransportControlData(index, DataString);
  ...
end;
```

4.31.7 TransportControlDataValid Function

The TransportControlDataValid Function returns whether all of [Media Transport Control](#) Data, has been returned from a media server.

Applicability

Colour C-Touch and C-Touch Mark 2 only.

Syntax

TransportControlDataValid

Description

The TransportControlDataValid function returns a boolean value which indicates whether all data (previously requested using [GetTransportControlData Procedure](#)) has been received.

Example

To check the Media Transport Control data is complete before using it:

```
if TransportControlDataValid then
begin
  ...
end;
```

4.31.8 TransportControlFlag Function

The TransportControlFlag Function returns whether certain [Media Transport Control](#) messages have been received from a media client.

Applicability

Colour C-Touch and C-Touch Mark 2 only.

Syntax

TransportControlFlag(MLG, FlagNumber)

MLG is an integer for Media Link Group [Tag](#)
FlagNumber in an integer

Description

The TransportControlFlag function returns a boolean value which indicates certain C-Bus Messages have been received for a particular Media Link Group. This is of use if you want to control a media server (via [Serial](#) or [Sockets \(TCP/IP\)](#)) and to provide access to that via C-Bus.

Note that the flags are automatically cleared as soon as this function has been used to look at their state.

The flag number values are:

Flag Number	Usage
0	Has a Next Category command been received
1	Has a Previous Category command been received
2	Has a Next Selection command been received
3	Has a Previous Selection command been received
4	Has a Next Track command been received
5	Has a Previous Track command been received

Example

To perform an action if a Next Track message has been received on Media Link Group 2:

```
if TransportControlFlag(2, 4) then
begin
...
end;
```

4.32 Complex Data Types



In addition to the standard Pascal [Types](#), it is possible to create user defined types, including :

- [Enumerated Types](#)
- [Sub-Ranges](#)
- [Arrays](#)
- [Records](#)
- [Pointers](#)
- [Sets](#)

The need for these does not arise often for automation purposes.

4.32.1 Enumerated Types

An enumerated type (also called a scalar) defines an ordered set of values by simply listing [identifiers](#) that represent these values. The values have no inherent meaning, and their ordinality follows the sequence in which the identifiers are listed. The benefit of this is that you can use names rather than numbers to represent values.

To declare an enumerated type, use the syntax

```
{ type section }
TypeName = (val1, ..., valn);
```

where `TypeName` and each `val` are valid [identifiers](#). For example, the declaration

```
{ type section }
Suit = (Club, Diamond, Heart, Spade);
```

defines an enumerated type called `Suit` whose possible values are `Club`, `Diamond`, `Heart`, and `Spade`. When you declare an enumerated type, you are declaring each `val` to be a constant of type `TypeName`. If the `val` identifiers are used for another purpose within the same [scope](#), naming conflicts occur. For example, suppose you declare the type

```
{ type section }
TSound = (Click, Clack, Clock);
```

Unfortunately, if `Click` is also the name of a variable used within the same scope, you'll get a compilation error; the compiler interprets `Click` within the scope of the procedure as a reference to the `Click` variable. A good solution is to choose constant names that are not likely to conflict with other identifiers. Examples:

```
{ type section }
TSound = (tsClick, tsClack, tsClock);
TMyColor = (mcRed, mcBlue, mcGreen, mcYellow, mcOrange);
Answer = (ansYes, ansNo, ansMaybe);
```

The prefixes in the above example are used to indicate the type, and to prevent naming clashes. So, with the above declarations, you could still use a variable called `Click`.

You can use the `(val1, ..., valn)` construction directly in variable declarations, as if it were a type name:

```
{ var section }
MyCard: (Club, Diamond, Heart, Spade);
```

But if you declare `MyCard` this way, you can't declare another variable within the same scope using these constant identifiers. Thus

```
{ var section }
Card1: (Club, Diamond, Heart, Spade);
Card2: (Club, Diamond, Heart, Spade);
```

generates a compilation error. But

```
{ var section }
Card1, Card2: (Club, Diamond, Heart, Spade);
```

compiles cleanly, as does

```
{ type section }
Suit = (Club, Diamond, Heart, Spade);

{ var section }
Card1: Suit;
Card2: Suit;
```

Consider the following code,

```
{ type section }
```

```

Weekday = (Monday, Tuesday, Wednesday, Thursday, Friday);
{ var section }
Workday : Weekday;

```

The first symbol in the set has an ordinal value of zero, and each successive symbol has a value of one greater than its predecessor. Hence :

```

Tuesday < Friday

```

evaluates as true, because Tuesday occurs before Friday in the set.

The following [Ordinal Functions](#) can also be used on Enumerated Types.

Ord

`ord(symbol)` returns the value of the symbol, thus

```
ord(Tuesday)
```

will give a value of 1

Pred

`pred(symbol)` obtains the previous symbol, thus

```
pred(Wednesday)
```

will give Tuesday

Succ

`succ(symbol)` obtains the next symbol, thus

```
succ(Monday)
```

gives Tuesday

Enumerated values can be used in for statements :

```
for Workday := Monday to Friday
```

or as a constant in a case statement :

```

case Workday of
  Monday : writeln('Mondays always get me down. ');
  Tuesday, Wednesday, Thursday : writeln('Another Day, Another Dollar. ');
  Friday : writeln('Get ready for party time! ');
end;

```

Examples

If you wanted to record the state that the home security system was in, you could define an enumerated type :

```
TSecurityMode = (Disarmed, Armed, Away, Home, Night);
```

You could then define a variable of this type and use it :

```

{ var section }
SecurityMode : TSecurityMode;

{ module section }
if (Time = sunset) and (SecurityMode = Away) then ...

```

4.32.2 Sub-Ranges

A subrange type represents a subset of the values in another [ordinal](#) type (called the base type). Any construction of the form

```
Low..High
```

where Low and High are constant expressions of the same ordinal type and Low is less than High, identifies a subrange type that includes all values between Low and High. For example, if you declare the enumerated type

```
type
  TColors = (Red, Blue, Green, Yellow, Orange, Purple, White, Black);
```

you can then define a subrange type like

```
type
  TMyColors = Green..White;
```

Here TMyColors includes the values Green, Yellow, Orange, Purple, and White.

You can use numeric constants and characters to define subrange types:

```
type
  SomeNumbers = -128..127;
  Caps = 'A'..'Z';
```

When you use numeric or character constants to define a subrange, the base type is the smallest integer or character type that contains the specified range.

The Low..High construction itself functions as a type name, so you can use it directly in variable declarations. For example,

```
var
  SomeNum: 1..500;
```

declares an integer variable whose value can be anywhere in the range from 1 to 500.

The ordinality of each value in a subrange is preserved from the base type. (In the first example above, if Color is a variable that holds the value Green, Ord(Color) returns 2 regardless of whether Color is of type TColors or TMyColors.) Values do not wrap around the beginning or end of a subrange, even if the base is an integer or character type; incrementing or decrementing past the boundary of a subrange simply converts the value to the base type. Hence, while

```
type
  Percentile = 0..99;

var
  I: Percentile;
  ...
  I := 100;
```

produces an error,

```
I := 99;
Inc(I);
```

assigns the value 100 to I (unless compiler [range-checking](#) is enabled).

4.32.3 Arrays

An array is a structure which holds many variables, all of the same data type. The array consists of a certain number of elements, each element of the array capable of storing one piece of data (ie, a variable). Arrays can hence be used to store many variables in an ordered way. Array types are denoted by constructions of the form

```
array[IndexType1, ..., IndexTypeN] of baseType
```

where each IndexType is an [ordinal](#) type. Since the IndexType indexes the array, the number of elements an array can hold is limited by the product of the sizes of the IndexType. In practice, IndexType are usually integer subranges.

One Dimensional Arrays

In the simplest case of a one-dimensional array, there is only a single IndexType. For example,

```
var
  MyArray: array[1..100] of Char;
```

declares a variable called MyArray that holds an array of 100 character values. Given this declaration, MyArray[3] denotes the third character in MyArray. If you create an array but don't assign values to all its elements, the unused elements are still allocated and contain random data; they are like uninitialised variables.

Multi-Dimensional Arrays

A multidimensional array is an array of arrays. For example,

```
type
  TMatrix = array[1..10] of array[1..50] of Real;
```

is equivalent to

```
type
  TMatrix = array[1..10, 1..50] of Real;
```

Whichever way TMatrix is declared, it represents an array of 500 real values. A variable MyMatrix of type TMatrix can be indexed like this: MyMatrix[2,45]; or like this: MyMatrix[2][45].

An array type of the form

```
array[0..x] of Char
```

is called a zero-based character array. Zero-based character arrays are used to store [strings](#).

Example

If you want to keep track how long various loads have been on, you could create an array to store the on times :

```
var
  OnDuration : array[0..255] of integer;
```

You could then check if each load is on once per minute and accumulate the total :

```

once Second = 0 then
  for Group := 0 to 255 do
    if GetLightingState(Group) then
      OnDuration[Group] := OnDuration[Group] + 1;

```

4.32.4 Records

A record (similar to a structure in some languages) represents a set of elements. Each element is called a field. The declaration of a record type specifies a name and type for each field. The syntax of a record type declaration is

```

{ type section }
RecordTypeName = record
  FieldList1: type1;
  ...
  FieldListn: typen;
end;

```

Where RecordTypeName is a valid [identifier](#), each type denotes a [type](#), and each FieldList is a valid identifier or a comma-delimited list of identifiers. The final semicolon is optional. For example, the following declaration creates a record type called TDateRec.

```

{ type section }
TDateRec = record
  Year: Integer;
  Month: (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec);
  Day: 1..31;
end;

```

Each TDateRec contains three fields: an integer value called Year, a value of an enumerated type called Month, and another integer between 1 and 31 called Day. The identifiers Year, Month, and Day are the field designators for TDateRec, and they behave like variables. The TDateRec type declaration, however, does not allocate any memory for the Year, Month, and Day fields; memory is allocated when you instantiate the record, like this:

```

{ var section }
Record1, Record2: TDateRec;

```

This variable declaration creates two instances of TDateRec, called Record1 and Record2.

You can access the fields of a record by using the field designators with the record's name:

```

Record1.Year := 1904;
Record1.Month := Jun;
Record1.Day := 16;

```

Or use a with statement:

```

with Record1 do
begin
  Year := 1904;
  Month := Jun;
  Day := 16;
end;

```

You can copy the values of Record1's fields to Record2:

```
Record2 := Record1;
```

Because the scope of a field designator is limited to the record in which it occurs, you don't have to worry about naming conflicts between field designators and other variables.

Instead of defining record types, you can use the record construction directly in variable declarations:

```
var
  S: record
    Name: string;
    Age: Integer;
  end;
```

However, a declaration like this largely defeats the purpose of records, which is to avoid repetitive coding of similar groups of variables. Moreover, separately declared records of this kind will not be assignment-compatible, even if their structures are identical.

Variant records are also possible to use with the Logic Engine. As these are very uncommon, they are not discussed in this document. Refer to any [Pascal](#) book for more details.

Example

If you want to create [Scenes](#), but not use the in-built Scene functions, you could create your own. First you would create a record structure to hold each item in the Scene :

```
SceneItem = record
  GroupAddress : integer;
  Level : integer;
  RampRate : integer;
end;
```

Then you could create a record to contain a Scene, which is just an array of SceneItems :

```
Scene : array[1..SceneItemCount] of SceneItem;
```

To set the Scene, you could do as follows :

```
for i := 1 to SceneItemCount do
  SetLightingLevel(Scene[i].GroupAddress, Scene[i].Level, Scene[i].RampRate);
```

4.32.5 Pointers

A pointer is a data type which holds a memory address.

To declare a pointer data type, you must specify what it will point to. That data type is preceded with a carat (^). For example, if you are creating a pointer to an integer, you would use this code:

```
{ type section }
IntegerPointer = ^integer;
```

You can then, of course, declare variables to be of type IntegerPointer.

```
{ var section }
P1, P2, P3 : IntegerPointer;
```

Before accessing a pointer, you must create a memory space for it. This is done with the **new** function :

```
New(PointerVariable);
```

To access the data stored at that memory address, you "de-reference" the pointer by adding a caret after the variable name. For example, if P1 was declared as type IntegerPointer (from above), you can assign the memory location a value by using:

```
P1^ := 5;
```

After you are done with the pointer, you must deallocate the memory space. Otherwise, each time the program is run, it will allocate more and more memory until your computer has no more. To deallocate the memory, you use the [Mark and Release](#) commands.

A pointer can be assigned to another pointer. However, note that since only the address, not the value, is being copied, once you modify the data located at one pointer, the other pointer, when de-referenced, also yields modified data. Also, if you free (or deallocate) a pointer, the other pointer now points to meaningless data.

The primary use of pointers is in creating dynamically-sized data structures. If you need to store many items of one data type in order, you can use an array. However, your array has a predefined size. If you don't have a large enough size, you may not be able to accommodate all the data. If you have a huge array, you take up a lot of memory when sometimes that memory is not being used.

A dynamic data structure, on the other hand, takes up only as much memory as is being used. What you do is to create a data type that points to a record. Then, the record has that pointer type as one of its fields. For example, stacks and queues can all be implemented using this data structure:

```
{ type section }
  PointerType = ^RecordType;
  RecordType = record
    data : integer;
    next : PointerType;
  end;
```

Each element points to the next. To know when a chain of records has ended, the next field is assigned a value of nil.

Pointers which do not reference any memory location should be assigned the value nil.

For more information about Pointers, refer to any [Pascal](#) book.

4.32.6 Memory Management

Code Memory

The program code is allocated 20,000 compiled instructions. This typically corresponds to around 5,000 lines of code, but depends on the particular instructions used.

Stack Memory

Memory allocated at compile time (regular variables) is called Statically Allocated Memory and is stored on the "stack".

The stack contains enough storage for 20,000 (6,000 for PAC) data items ([integers](#), [real numbers](#) or [characters](#)). The constants area of the memory has allocation for :

- 500 (200 for PAC) integers; and
- 500 (200 for PAC) real numbers; and

- 1000 (none for PAC) set elements; and
- 10,000 (3,000 for PAC) characters (in strings)

Heap Memory

Memory allocated by the [New](#) function is called Dynamically Allocated Memory, and is stored in an area called the "heap".

The heap shares the memory allocated for the stack. Using more heap memory reduces the amount available for the stack.

Each time the Logic Engine is run, any memory used in the heap is cleared. Each time a new pointer is created, some heap memory is used up. Eventually, it is possible to use all of the heap memory, in which case a run time error will occur. Memory which is no longer required needs to be freed (released) so that the heap memory does not run out.

The method of releasing heap memory which is no longer required is different with the Logic Engine to how it is normally performed in Pascal. A heap memory manager is very complex, and needs to cope with the fragmentation of heap memory as it is allocated and then freed up again. In the Logic Engine, heap memory is released in a block, rather than trying to release individual bits of memory.

The process is as follows :

- Record (Mark) the start of the heap memory block which will be later released
- Allocate heap memory as required
- Release the heap memory when complete. This will release all heap memory allocated since Marking the start point.

To mark the start of a block, the Mark function is used. It is passed a parameter which is a pointer (of any type), which stores the address of the start point in the heap. The example below shows a pointer being allocated as the Mark parameter, the Mark statement, usage of memory and finally the release of the memory.

```
{ var }
MarkPtr : ^integer;
x, y, z : ^MyDataStructure;
{ ... }
{ main program }
mark(MarkPtr);
New(x);
New(y);
New(z);
{ use x, y and z }
release(MarkPtr); { free up heap memory used by x, y and z }
```

Note that you do not need to worry about releasing memory if :

- you are not using pointers; or
- if you are using pointers, but once allocated, they are used for the rest of the duration that the Logic Engine is running

4.32.7 Sets

Applicability

Colour C-Touch only.

Sets exist in every day life. They are a way of classifying common types into groups. In Pascal, sets contain a range of limited values, from an initial value through to an ending value.

Consider the following set of integer values :

1, 2, 3, 4, 5, 6, 7, 8, 9, 10

This is a set of numbers (integers) whose set value [ranges](#) from 1 to 10. To define this as a set type in Pascal, we would use the following syntax.

```
type
  numberset = set of 1..10;

var
  My Numbers : numberset;
```

The type statement declares a new type called numberset, which represents a set of integer values ranging from 1 as the lowest value, to 10 as the highest value. The value 1..10 means the numbers 1 to 10 inclusive. We call this the base set, that is, the set of values from which the set is taken.

The base set is a range of limited values. For example, we can have a set of char, but not a set of integers, because the set of integers has too many possible values, whereas the set of characters is very limited in possible values.

The var declaration makes a working variable in our program called MyNumbers, which is a set and can hold any value from the range defined in numberset.

See also :

[Set Operations](#)

[Set Example](#)

4.32.7.1 Set Operations

Applicability

Colour C-Touch only.

The typical operations associated with [sets](#) are,

- assign values to a set
- determine if a value is in one or more sets
- set addition (UNION)
- set subtraction (DIFFERENCE)
- set commonality (INTERSECTION)

Assigning Values to a set

The statement :

```
MyNumbers := [1, 2];
```

places the values 1 and 2 in the set.

An empty set can be created with

```
MyNumbers := [];
```

The statement

```
MyNumbers := [2..6];
```

assigns a subset of values (integer 2 to 6 inclusive) from the range given for the set type numberset. Please note that assigning values outside the range of the set type from which MyNumbers is derived will generate an error, thus the statement

```
MyNumbers := [6..32];
```

is illegal, because MyNumbers is derived from the base type numberset, which is a set of integer

values ranging from 1 to 10. Any values outside this range are considered illegal.

Set Union

Set union is essentially the addition of sets. Consider the following statements :

```
MyNumbers := [1, 2];
MyNumbers := MyNumbers + [4];
```

MyNumbers now contains the elements 1, 2 and 4.

Set Difference

Set difference is essentially the subtraction of sets. Consider the following statements :

```
MyNumbers := [1, 2];
MyNumbers := MyNumbers - [1];
```

MyNumbers now contains the only the element 2.

Determining if a value is in a set

To determine whether an element is in a set, the "in" operator is used :

```
MyNumbers := [1, 2];
Test := 1 in MyNumbers;
```

In the above case, the boolean variable Test will be true.

Set Commonality (Intersection)

The set commonality or intersection operator (*) is used to determine which elements are common to two sets. For example :

```
Set1 := [1..5];
Set2 := [4..9];
Set3 := Set1 * Set2;
```

Will result in Set2 containing [4, 5], as these are the only two elements common to both sets.

4.32.7.2 Set Example

If you wanted to record the state that the home security system was in, you could define an [enumerated type](#) :

```
TSecurityMode = (Disarmed, Armed, Away, Home, Night);
```

You could then define a variable of this type and use it :

```
{ var section }
SecurityMode : TSecurityMode;

{ module section }
if (Time = sunset) and (SecurityMode in [Away, Night]) then ...
```

You could define a [set](#) of modes :

```
{ var section }
PartiallyArmed : set of TSecurityMode;
```

and assign a set to it :

```
{ initialisation section }
PartiallyArmed := [Armed, Away, Home, Night];
```

Then it can be used in your Logic :

```
{ module section }
if SecurityMode in PartiallyArmed then ...
```

4.32.8 Tutorial 11

Question 1

What is the output of the following :

```
{ var }
numbers : ARRAY [1..5] of integer;
loop : integer;

{ main program }
numbers[1] := 7;
numbers[2] := 13;
numbers[3] := numbers[2] - 1;
numbers[4] := numbers[3] DIV 3;
numbers[5] := numbers[3] DIV numbers[4];
for loop := 1 to 5 do
  writeln('Numbers[' ,loop,'] is', numbers[loop] );
```

Tutorial Answers

4.33 Files



A file is a collection of information, usually stored on a computer hard disk. This information is accessed via means of a file variable.

Before a file variable can be used, it must be associated with an external file through a call to the [AssignFile](#) procedure. The external file stores the information written to the file or supplies the information read from the file. For security reasons, only files in the project directory (the directory where your project is stored) can be accessed.

Once the association with an external file is established, the file variable must be "opened" to prepare it for input or output. An existing file can be opened via the [Reset](#) procedure, and a new file can be created and opened via the Rewrite procedure. Text files opened with Reset are read-only and text files opened with [Rewrite](#) and [AppendFile](#) are write-only.

When a program completes processing a file, the file must be closed using the standard procedure [CloseFile](#). After a file is closed, its associated external file is updated. The file variable can then be associated with another external file.

The Logic Engine only supports text files, not typed files. Text files are arranged as a sequence of

variable length lines :

- Each line consists of a sequence of characters.
- Each line is terminated with a special character, called END-OF-LINE ([EOLN](#))
- The last character in the file is another special character, called END-OF-FILE ([EOF](#))

The Logic Engine pre-defines two file variables for use. They are called file1 and file2. Additional file variables can not be created. This means that you can have a maximum of two files open at once. It does not limit the number of different files which can be accessed though.

Note : Files used by logic will not automatically be included in the Project Archive. See the Exporting to an Archive topic for details of adding files to the archive.

See also [UTF-8 Example](#)

4.33.1 AssignFile Procedure

The AssignFile procedure assigns a file name to a file variable.

Applicability

Colour C-Touch only.

Syntax

```
AssignFile(FileVariable, FileName);
```

FileVariable is a file variable (either file1 or file2)

FileName is a [string](#)

Description

This associates the name of a file on the disk (in the project directory) with the file variable.

Example

To associates the file1 variable with the file "MyFile.txt" :

```
AssignFile(file1, 'MyFile.txt');
```

4.33.2 Reset Procedure

The Reset procedure opens an existing file for reading.

Applicability

Colour C-Touch only.

Syntax

```
Reset(FileVariable);
```

FileVariable is a file variable (either file1 or file2)

Description

Reset opens the existing external file with the name [assigned](#) to FileVariable. An error results if no existing external file of the given name exists. If FileVariable is already open, it is first closed and then reopened. The current file position is set to the beginning of the file.

Example

To reset the file associated with the file1 variable :

```
Reset(file1);
```

4.33.3 Rewrite Procedure

The ReWrite procedure creates a new file and opens it for writing.

Applicability

Colour C-Touch only.

Syntax

```
ReWrite(FileVariable);
```

FileVariable is a file variable (either file1 or file2)

Description

Rewrite creates a new external file with the name assigned to FileVariable. FileVariable is a variable associated with an external file using [AssignFile](#).

If an external file with the same name already exists, it is deleted and a new empty file is created in its place.

If FileVariable is already open, it is first closed and then re-created. The current file position is set to the beginning of the empty file.

After calling Rewrite, Eof(FileVariable) is always True.

Example

To ReWrite the file associated with the file1 variable :

```
ReWrite(file1);
```

4.33.4 Reading from Files

The syntax for reading file data is:

```
Read(FileName, Variable_List);
```

FileName is the name of a file variable (either file1 or file2)

Variable_List is a series of variable identifiers separated by commas.

The read procedure, however, does not go to the next line. This can be a problem with character input, because the end-of-line character is read as a space.

To read data and then go on to the next line, use :

```
ReadLn(FileName, Variable_List);
```

Examples

Suppose you had the following data in a file, and variables a, b, c, and d were all integers.

```
45 97 3
1 2 3
```

This would be the result of various statements:

Statement(s)	a	b	c	d
read (file1, a); read (file1, b);	45	97		
readln (file1, a); read (file1, b);	45	1		
read (file1, a, b, c, d);	45	97	3	1
readln (file1, a, b); readln (file1, c, d);	45	97	1	2

The read statement does not skip to the next line unless necessary, whereas the readln statement is just a read statement that skips to the next line at the end of reading.

When reading integers, all spaces are skipped until a numeral is found. Then all subsequent numerals are read, until a non-numeric character is reached (including, but not limited to, a space).

Variables of any [Type](#) can be read from a file.

4.33.5 AppendFile Procedure

The AppendFile procedure prepares an existing file for adding text to the end.

Applicability

Colour C-Touch only.

Syntax

```
AppendFile(FileVariable);
```

FileVariable is a file variable (either file1 or file2)

Description

Call Append to ensure that a file is opened with write-only access with the file pointer positioned at the end of the file. FileVariable is a text file variable and must be associated with an external file using [AssignFile](#). If no external file of the given name exists, an error occurs. If FileVariable is already open, it is closed, then reopened. The current file position is set to the end of the file.

Example

To prepare the file associated with the file1 variable for having text appended to it :

```
AppendFile(file1);
```

4.33.6 Writing to Files

Writing to files is the same as using the [Write and WriteLn](#) procedures for displaying data, except that the file name is included as a parameter :

```
Write(FileName, Argument_List);
WriteLn(FileName, Argument_List);
```

Instead of writing to the Logic Engine results screen, the data will be written to the file.

Examples

To write "the result is" followed by the value of variable i to file1 :

```
WriteLn(file1, 'the result is', i);
```

4.33.7 CloseFile Procedure

The CloseFile procedure closes a file.

Applicability

Colour C-Touch only.

Syntax

```
CloseFile(FileVariable);
```

FileVariable is a file variable (either file1 or file2)

Description

This closes the file and its associated external file is updated. The file variable can then be associated with another external file.

Example

To close the file associated with file1 :

```
CloseFile(file1);
```

4.33.8 EOF Function

The EOF function tests whether the file position is at the end of a file.

Applicability

Colour C-Touch only.

Syntax

```
EOF(FileVariable);
```

FileVariable is a file variable (either file1 or file2)

Description

Eof tests whether the current file position is the end-of-file. Eof(FileVariable) returns True if the current file position is beyond the last character of the file or if the file is empty; otherwise, Eof(F) returns False.

Example

To perform an action if the file1 is not at the end :

```
if not Eof(file1) then ...
```

4.33.9 EOLN Function

The EOLN function tests whether the file pointer is at the end of a line.

Applicability

Colour C-Touch only.

Syntax

```
EOLN(FileVariable);
```

FileVariable is a file variable (either file1 or file2)

Description

Eoln tests whether the current file position is the end-of-line of a text file. Eoln(FileVariable) returns True if the current file position is at an end-of-line or if Eof(FileVariable) is True; otherwise, Eoln (FileVariable) returns False.

Example

To perform an action if the file1 is not at the end of a line :

```
if not Eoln(file1)then ...
```

4.33.10 FileExists Function

The FileExists function returns whether the file exists.

Applicability

Colour C-Touch only.

Syntax

```
FileExists(FileName)
```

FileName is a [String](#) containing the file name (without the path).

Description

This returns a [Boolean](#) value. If the file exists in the project directory, the result is true, otherwise it is false.

Example

To check if a file exists before opening the file for reading :

```
if FileExists('MyFile.txt') then
begin
  AssignFile(file1, 'MyFile.txt');
  Reset(file1);
  ...
  CloseFile(file1);
end;
```

4.33.11 File Example

For this example, we have an array of data which we need to store and recover after a power failure.

An array called LevelArray stores 100 integers. Calling the SaveData [Procedure](#) below will save the data. This needs to be done on a regular basis, but not too often. A period of around 10 to 30 minutes will provide a reasonable balance between ensuring that relatively recent data is available following a power failure, but without excessive use of the computer hard disk (or flash disk in Colour

C-Touch).

```

procedure SaveData;
var
  i : integer;
begin
  AssignFile(file1, 'data.txt');
  Rewrite(file1);
  for i := 1 to 100 do
    WriteLn(file1, LevelArray[i]);
  CloseFile(file1);
end;

procedure ReadData;
var
  i : integer;
begin
  AssignFile(file1, 'data.txt');
  Reset(file1);
  for i := 1 to 100 do
    if not eof(file1) then
      ReadLn(file1, LevelArray[i]);
  CloseFile(file1);
end;

```

You need to call ReadData when the logic first start. The [Initialisation](#) section of the code is a good place for this.

Note that the first time you run this, the data file will not exist and so the ReadData procedure will fail. There are three ways around this :

1. Add the SaveData procedure to the code first. Wait for it to save the data, then add the ReadData procedure.
2. Create a dummy 'data.txt' file.
3. Use the [FileExists Function](#) to check that the file exists before trying to read it.

4.33.12 Tutorial 12

Question 1

A file called "data.txt" contains 10 lines, each containing a single integer. Read the integers into an integer array called Data.

[Tutorial Answers](#)

4.34 ZigBee Functions

the Logic Engine is to provide control and monitoring of ZigBee.

There are a series of functions provided for access to ZigBee levels and states :

[SetZigbeeEndpointLightingLevel](#)
[SetZigbeeGroupLightingLevel](#)
[SetZigbeeScene](#)
[SetZigbeeEndpointCurtainLevel](#)
[SetZigbeeEndpointCurtainStop](#)
[SetZigbeeGroupCurtainLevel](#)
[SetZigbeeGroupCurtainStop](#)

[GetZigbeeEndpointLightingLevel](#)
[GetZigbeeGroupLightingLevel](#)
[StopZigbeeEndpointLightingRamp](#)
[StopZigbeeGroupLightingRamp](#)

4.34.1 **SetZigbeeEndpointLightingLevel**

The SetZigbeeEndpointLightingLevel procedure sets the level of a ZigBee Endpoint.

Syntax

```
SetZigbeeEndpointLightingLevel(Network, Node, Endpoint, NewLevel, RampRate);
```

Network is an [Integer](#) or [Network Tag](#).

Node is an integer or tag

Endpoint is an Integer or tag.

NewLevel is an Integer, Percent or Level Tag

RampRate is an integer (number of seconds) or [Ramp Rate Tag](#)

Description

The Endpoint on the selected Node and Network gets set to the NewLevel, with a specified Ramp Rate. If you select a ramp rate other than the [standard ramp rates](#), it will choose the closest one.

Example

To set the value of the Endpoint called "Ulti 2 Gang Dimmer 1 Channel" on the "Ulti 2 Gang Dimmer 1" Node on "My Network" to level 128 over 4 seconds :

```
SetZigbeeEndpointLightingLevel("My Network", "Ulti 2 Gang Dimmer 1", "Ulti 2 Gang Dimmer 1 Channel 1", 128, 4);
```

4.34.2 **SetZigbeeGroupLightingLevel**

The SetZigbeeGroupLightingLevel procedure sets the level of a ZigBee Group.

Syntax

```
SetZigbeeGroupLightingLevel(Network, Group, NewLevel, RampRate);
```

Network is an [Integer](#) or [Network Tag](#).

Group is an integer or tag

NewLevel is an Integer, Percent or Level Tag

RampRate is an integer (number of seconds) or [Ramp Rate Tag](#)

Description

The Group on the selected Network gets set to the NewLevel, with a specified Ramp Rate. If you select a ramp rate other than the [standard ramp rates](#), it will choose the closest one.

Example

To set the value of the Group called "Group 3784" on "My Network" to level 128 over 4 seconds :

```
SetZigbeeGroupLightingLevel("My Network", "Group 3784", 128, 4);
```

4.34.3 **SetZigbeeScene**

The SetZigbeeScene procedure triggers the ZigBee scene.

Syntax

```
SetZigbeeScene(Network, Group, Scene);
```

Network is an [Integer](#) or [Network Tag](#).

Group is an integer or tag

Scene is an integer or tag

Description

The Scene on the selected Network gets triggered.

Example

To trigger the scene called "Scene 3784.1" on ZigBee Group "Group 3784" on "My Network" :

```
SetZigbeeScene("My Network", "Group 3784", "Scene 3784.1");
```

4.34.4 **SetZigbeeEndpointCurtainLevel**

The SetZigbeeEndpointCurtainLevel procedure sets the position of a ZigBee endpoint curtain controller.

Syntax

```
SetZigbeeEndpointCurtainLevel(Network, Node, Endpoint, NewLevel);
```

Network is an [Integer](#) or [Network Tag](#).

Node is an integer or tag

Endpoint is an Integer or tag.

NewLevel is an Integer, Percent or Level Tag

Description

The curtain controller Endpoint on the selected Node and Network gets set to position given by NewLevel. The rate at which the curtain controller moves is programmed in the curtain controller.

To get the position of the curtain use the function [GetZigbeeEndpointLightingLevel](#).

Example

To set the position of the Endpoint curtain controller called "Ulti 2 Gang Dimmer 1 Channel" on the "Ulti 2 Gang Dimmer 1" Node on "My Network" to position 128 :

```
SetZigbeeEndpointLightingLevel("My Network", "Ulti 2 Gang Dimmer 1", "Ulti 2 Gang Dimmer 1 Channel 1", 128);
```

4.34.5 **SetZigbeeEndpointCurtainStop**

The SetZigbeeEndpointCurtainStop procedure tells the ZigBee endpoint curtain controller to stop moving the curtain.

Syntax

```
SetZigbeeEndpointCurtainStop(Network, Node, Endpoint);
```

Network is an [Integer](#) or [Network Tag](#).

Node is an integer or tag

Endpoint is an Integer or tag.

Description

The curtain controller Endpoint on the selected Node and Network gets told to stop moving. The

position of the curtain will be broadcast by the curtain controller when it stops moving.
To read the position of the curtain use the function [GetZigbeeEndpointLightingLevel](#).

Example

To stop the Endpoint curtain controller called "Ulti 2 Gang Dimmer 1 Channel" on the "Ulti 2 Gang Dimmer 1" Node on "My Network" :

```
SetZigbeeEndpointCurtainStop("My Network", "Ulti 2 Gang Dimmer 1", "Ulti 2
Gang Dimmer 1 Channel 1");
```

4.34.6 SetZigbeeGroupCurtainLevel

The SetZigbeeGroupCurtainLevel procedure sets the position of a ZigBee group curtain controller.

Syntax

```
SetZigbeeGroupCurtainLevel(Network, Group, NewLevel);
```

Network is an [Integer](#) or [Network Tag](#).

Group is an integer or tag

NewLevel is an Integer, Percent or Level Tag

Description

The curtain controller Group on the selected Network gets set to position given by NewLevel. The rate at which the curtain controller moves is programmed in the curtain controller.

To get the position of the curtain use the function [GetZigbeeGroupLightingLevel](#).

Example

To set the position of the Endpoint curtain controller called "Group 17493" on "My Network" to position 128 :

```
SetZigbeeGroupCurtainLevel("My Network", "Group 17493", 128);
```

4.34.7 SetZigbeeGroupCurtainStop

The SetZigbeeGroupCurtainStop procedure tells the ZigBee group curtain controller to stop moving the curtain.

Syntax

```
SetZigbeeGroupCurtainStop(Network, Group);
```

Network is an [Integer](#) or [Network Tag](#).

Group is an integer or tag

Description

The curtain controller Group on the selected Node and Network gets told to stop moving. The position of the curtain will be broadcast by the curtain controller when it stops moving.

To read the position of the curtain use the function [GetZigbeeGroupLightingLevel](#).

Example

To stop the Group curtain controller called "Group 17493" on "My Network" :

```
SetZigbeeGroupCurtainStop("My Network", "Group 17493");
```

4.34.8 GetZigbeeEndpointLightingLevel

The GetZigbeeEndpointLightingLevel function returns the [level](#) of a ZigBee Endpoint.

Syntax

```
GetZigbeeEndpointLightingLevel(Network, Node, Endpoint)
```

Network is an [Integer](#) or [Network Tag](#).

Node is an Integer or Application Tag.

Endpoint is an Integer or Group Address Tag.

Description

The integer result is the [level](#) of the Endpoint.

Example

To perform an action if the value of Endpoint called "Ulti 2 Gang Dimmer 1" on the Node "Ulti 2 Gang Dimmer 1 Channel 1" on "My Network" is 255 :

```
if GetZigbeeEndpointLightingLevel("My Network", "Ulti 2 Gang Dimmer 1",  
"Ulti 2 Gang Dimmer 1 Channel 1") = 255 then ...
```

4.34.9 GetZigbeeGroupLightingLevel

The GetZigbeeGroupLightingLevel function returns the [level](#) of a ZigBee Group.

Syntax

```
GetZigbeeGroupLightingLevel(Network, Group)
```

Network is an [Integer](#) or [Network Tag](#).

Group is an Integer or Application Tag.

Description

The integer result is the [level](#) of the Group.

Example

To perform an action if the value of Group called "Group 17493" on "My Network" is 255 :

```
if GetZigbeeGroupLightingLevel("My Network", "Group 17493") = 255 then ...
```

4.34.10 StopZigbeeEndpointLightingRamp

The StopZigbeeEndpointLightingRamp procedure tells the ZigBee lighting endpoint stop ramping the endpoint.

Syntax

```
StopZigbeeEndpointLightingRamp(Network, Group);
```

Network is an [Integer](#) or [Network Tag](#).

Group is an integer or tag

Description

The lighting Endpoint on the selected Node and Network gets told to stop ramping. The start ramp is done using the [SetZigbeeEndpointLightingLevel](#) command.

To read the level of the endpoint use the function [GetZigbeeEndpointLightingLevel](#).

Example

To stop ramping the Endpoint called "Ulti 2 Gang Dimmer 1 Channel" on the "Ulti 2 Gang Dimmer 1" Node on "My Network" :

```
StopZigbeeEndpointLightingRamp("My Network", "Group 17493");
```

4.34.11 StopZigbeeGroupLightingRamp

The StopZigbeeGroupLightingRamp procedure tells the ZigBee lighting group stop ramping the group.

Syntax

```
StopZigbeeGroupLightingRamp(Network, Group);
```

Network is an [Integer](#) or [Network Tag](#).

Group is an integer or tag

Description

The lighting Group gets told to stop ramping. The start ramp is done using the

[SetZigbeeGroupLightingLevel](#) command.

To read the position of the curtain use the function [GetZigbeeGroupLightingLevel](#).

Example

To stop ramping the Group called "Group 17493" on "My Network" :

```
StopZigbeeGroupLightingRamp("My Network", "Group 17493");
```

5 Debugging Programs

Most programs, even those created by professionals, contain a certain amount of errors (or "bugs"). The process of discovering and removing these bugs is called "debugging".

5.1 Error Types

There are three types of errors that can occur in a program :

- Syntax errors
- Run-time errors
- Logical Errors

Syntax Errors

Syntax Errors are where the user has entered code that does not follow the correct format (syntax). An example would be is the user entered :

```
Level = 0;
```

instead of

```
Level := 0;
```

Syntax Errors are found by the [Compiler](#) and generate [Compilation Error Messages](#).

Syntax Errors must be fixed before the program can be compiled and run.

Run-Time Errors

Run-time Errors occur when the program is [Running](#). They generally occur because a value is outside of the allowable range. For example, the following will cause [Run Time Errors](#) :

```
x := 5 / y;           when y = 0
x := MyArray[y];     when y is outside of the bounds of the array
StartTimer(1000000);
```

Logical Errors

Logical Errors occur when there is an error in the logic of the code, but it still compiles correctly and runs.

An example would if the user wanted something to occur only at night (between sunset and sunrise). If they wrote the code :

```
if (time < sunrise) and (time > sunset) then ...
```

then the condition would never be true (as the time can never be both before sunrise and after sunset). The correct code would be :

```
if (time < sunrise) or (time > sunset) then ...
```

Logical Errors will not generate error messages. The only way you can find them is by thoroughly testing the system and observing the behaviour. Once a program is running. Logical Errors are the most common, and the hardest to find.

Logical errors are the subject of the rest of this chapter.

5.2 Debugging Support Features

The Logic Engine provides the following features to support the debugging of programs.

Run Once

Using the [Run Once](#) option allows you to see what happens in one run through the logic engine. This is easier than trying to see what is happening when it is running full speed.

Displaying Status

At any part of the program it is possible to [Display](#) the status of any variable, or just write a message to the [Output Window](#). This can be useful for determining where the program is going, and what the values are.

Alternatively, you can write data directly to the [screen](#) which can be easier than having to refer to the Output Window.

Logging

Messages can also be [written](#) to the Log file. This is helpful if problems only occur infrequently, and it is not possible to sit watching the screen continuously, or if it happens to quickly to see what happened.

Resources

If your program will not run due to a lack of memory, the [Resources](#) window can be used to see how much is being used by what.

Debug Compilation

The Debug Compilation [Option](#) can be used to ensure that all array parameters etc are within bounds.

Halt and Restart Statements

The [Halt](#) and [Restart](#) statements can be used to terminate the Logic Engine if the user program has detected some kind of anomaly, or if an error condition arises.

LED Control

The PAC can use the [LED](#) to indicate the status of a parameter for debugging.

5.3 Debugging Methods

To debug programs, there are various techniques which can be used. In general, techniques which are used for debugging software apply equally well to debugging Logic Engine Programs :

- [Condition Testing](#)
- [Tracking What Your Program is Doing](#)
- [Intermittent Errors](#)

5.3.1 Condition Testing

The main aspect of Logic Engine Programs which needs to be tested are the [Conditional Logic](#)

statements. It is easy to make a mistake with the logic, and it is important that every condition be tested.

It is necessary to test both that the condition is true when you expect it to be true, and to also test that it is false when you expect it to be false.

To test whether a conditional statement works or not, it is necessary to have the Logic Engine [Running](#), and to set up the necessary conditions to make the statement be true or false.

As a simple case, consider the code :

```
if (time = "9:00PM") and (DayOfWeek = "Monday") then
  SetLightingState("Porch Light", ON);
```

The simplest way to test this is to wait until 9PM on Monday night and see if the Porch Light come on. This is obviously not a practical solution. A better way is to set the PC time and date to 8:59PM on a Monday and wait for 1 minute to see if the Porch Light comes on.

Remember to also test the negative condition. You have already tested that the light comes on when it is 9PM and it is Monday. You should also test 9PM on a Tuesday, and make sure that it doesn't come on.

The type of things you may need to do in order to test a condition are :

- Set the system time and/or date to just before an event is to occur
- Set C-Bus levels to a particular level
- Set [System IO Variables](#) to particular values

5.3.2 Tracking What Your Program is Doing

Sometimes it is difficult to work out what your program is doing. It may look like a condition is true, but the statement is not being executed. The most common way to solve these problems is to put statements in your code to display or log some data to indicate what is happening.

For example, if you had some code :

```
if counter = 5 then
  SetLightingState("fan", OFF);
```

If the fan is not going off, it may be hard to know why. Putting a [WriteLn](#) or [LogMessage](#) statement immediately before had will tell you what is happening. For example, you could add the following line of code :

```
WriteLn('Counter = ', Counter);
if counter = 5 then
  SetLightingState("fan", OFF);
```

In this case, the value of the Counter will be displayed in the [Output Window](#) every time the Logic Engine is [Run](#). If you can see that the Counter variable has a value of 124, then the chances are that you may not have initialised the value properly, or may not have reset it.

When you have debugged the code, remember to remove any of these debugging statements.

5.3.3 Intermittent Errors

Sometimes problems (errors) only occur occasionally, and they are hard to observe and debug. These are referred to as "intermittent" problems, and are the most difficult to track down.

The best method of trying to identify the source of the problems is to try to determine under what circumstances the problem occurs. If there is a definite pattern, then this may provide the necessary information to locating the source of the problem. Waiting for the problem to occur enough times to find a pattern can be a very time consuming and expensive exercise.

Another technique to support the locating of intermittent errors is to review the Logs to find out what was happening at the time of the problem. If necessary, use the [LogMessage](#) procedure to write data to the log to make this process simpler.

6 Error Messages

Error messages may occur during [Compilation](#) or at [Run Time](#). Error messages are displayed in the [Output Window](#). Compile errors start with a "C" followed by a number. Run-time errors start with an "R".

To find the source of an error, double click on the error message in the Output Window. The code will be displayed in the [Code Window](#), with the cursor at the position of the error. For run-time errors, the cursor will be placed at the start of the line.

See also [Compilation Errors \(C000 - C999\)](#), [Run Time Errors \(R000 - R999\)](#)

6.1 Compilation Errors

Compile errors occur during [Compiling](#) when there is a [syntax error](#) in the entered code.

To find where in the code an error is, just double click on the error message, and the cursor will be positioned at the position in the code where the compiler detected the error. It may be necessary to look either side of the cursor position, or even on the lines above to find the root cause of the error.

Note that a single error may result in more than one error message, depending on whether the compiler can determine what was intended. If more than one error message is reported, it is always best to fix the first error in the list of errors first. If the following error messages make sense, then fix them at the same time, otherwise [Compile](#) the code again and fix the first error in the new list.

Compilation Errors must be fixed before the program can be compiled and run.

The Compilation Errors are listed below.

Error C001 : Error in simple type

A [Simple Type](#) was expected, but was not found.

Error C002 : Identifier expected

An [Identifier](#) was expected, but was not found.

Error C003 : "program" expected

The reserved word "program" was expected, but was not found. This error should never be encountered.

Error C004 : ")" expected

A right parenthesis ")" was expected, but was not found.

Error C005 : ":" expected

A colon ":" was expected, but was not found.

Error C006 : Illegal symbol

An illegal symbol was found. Check the code syntax.

Error C007 : Error in parameter list

An illegal symbol was found in a [Parameter List](#).

Error C008 : "of" expected

The reserved word "of" was expected, but was not found.

Error C009 : "(" expected

A left parenthesis "(" was expected, but was not found.

Error C010 : Error in type

An error occurred in a [Type](#) declaration.

Error C011 : "[" expected

A left parenthesis "[" was expected, but was not found.

Error C012 : "]" expected

A right parenthesis "]" was expected, but was not found.

Error C013 : "end" expected

The reserved word "[end](#)" was expected, but was not found.

Error C014 : ";" expected

A semi-colon ";" was expected, but was not found.

Error C015 : Integer expected

An [Integer](#) constant was expected, but was not found.

Error C016 : "=" expected

An equals symbol "=" was expected, but was not found.

Error C017 : "begin" expected

The reserved word "[begin](#)" was expected, but was not found.

Error C018 : Error in declaration part

An error was found in the declaration part of a [Block](#).

Error C019 : Error in field-list

An error was found in the field list of a [Record](#).

Error C020 : "," expected

A comma "," was expected, but was not found.

Error C021 : "." expected

A period "." was expected, but was not found. The end of the [Program](#) was expected. Most likely

cause is begin/end statements not matched up.

Error C050 : Error in constant

A [Constant](#) was expected, but was not found.

Error C051 : "!=" expected

An [assignment](#) symbol "!=" was expected, but was not found.

Error C052 : "then" expected

The reserved word "[then](#)" was expected, but was not found.

Error C053 : "until" expected

The reserved word "[until](#)" was expected, but was not found.

Error C054 : "do" expected

The reserved word "[do](#)" was expected, but was not found.

Error C055 : "to"/"downto" expected

The reserved word "[to](#)" or "downto" was expected, but was not found.

Error C058 : Error in factor

An error occurred in a [factor](#). Check that brackets match up.

Error C059 : Error in variable

An error occurred in a variable selector ([array](#) index, [record](#) field or [pointer](#) reference).

Error C101 : Identifier declared twice

An [Identifier](#) has been declared twice. Remove one of the declarations.

Error C102 : Low bound exceeds high bound

The low bound of a [sub-range](#) is greater than the high bound.

Error C103 : Identifier is not of appropriate class

The [Identifier](#) is of the wrong [Type](#).

Error C104 : Identifier is not declared

The [Identifier](#) has not been declared. Check that the identifier has been declared and that the spelling is correct.

Error C105 : Sign not allowed

A sign "-" or "+" is not allowed.

Error C106 : Number expected

A number was expected, but was not found.

Error C107 : Incompatible subrange types

A [Sub-Range](#) of a different type was expected.

Error C108 : File not allowed here

A [File](#) variable is not allowed at this point in the code.

Error C109 : Type must not be real

A [Real Type](#) is not allowed. Change to an appropriate type.

Error C110 : Tagfield type must be scalar or subrange

Only a [scalar](#) or [Sub-Range](#) is allowed.

Error C111 : Incompatible with tagfield type

Constant type does not match field type.

Error C113 : Index type must be scalar or subrange

[Array](#) index must be [scalar](#) or [Sub-Ranges](#).

Error C114 : Base type must not be real

[Set](#) base [Type](#) must be [scalar](#) or [Sub-Range](#).

Error C115 : Base type must be scalar or subrange

[Set](#) base [Type](#) must be [scalar](#) or [Sub-Range](#).

Error C116 : Error in type of standard procedure parameter

Error in [Parameters](#) of a standard procedure.

Error C117 : Unsatisfied forward reference

A [Forward Declaration](#) of a procedure, function or pointer was not completed.

Error C119 : Forward declared; repetition of parameter list not allowed

A [Forward Declaration](#) of a procedure, function or pointer was repeated.

Error C120 : Function result type must be scalar, subrange or pointer

A function can only return a result which is [scalar](#), [Sub-Range](#) or [pointer](#).

Error C121 : File value parameter not allowed

A [File](#) variable can not be passed as a value parameter.

Error C122 : Forward declared function repetition of result type not allowed

The [function](#) result [Type](#) can not be repeated for [Forward Declared](#) functions.

Error C123 : Missing result type in function declaration

The [function](#) result [Type](#) was missing in the [Forward Declaration](#).

Error C124 : Field precision format for real only

Field precision in [Write](#) statement only allowed for [Real](#) types.

Error C125 : Error in type of standard function parameter

Error in [type](#) of a standard function/procedure [Parameter](#).

Error C126 : Number of parameters does not agree with declaration

Error in number of [Parameters](#) in a function/procedure. Either too many or not enough parameters have been entered.

Error C128 : Result type of parameter function does not agree with declaration

[Function](#) result [type](#) does not match the function declaration.

Error C129 : Type conflict of operands

The two [operands](#) are not [Type](#) compatible.

Error C130 : Expression is not of set type

A [set](#) expression was expected. For example, a set variable or a set constant (eg [1, 2, 4, 7]).

Error C131 : Tests on equality allowed only

Only the equality [operator](#) "=" can be used for this type. Other operators are not allowed.

Error C132 : Strict inclusion not allowed

The ">" and "<" operators can not be used with [sets](#).

Error C133 : File comparison not allowed

[Files](#) can not be compared.

Error C134 : Illegal type of operand(s)

The multiply (*) [Operator](#) can not be used with these [operands](#).

Error C135 : Type of operand must be boolean

The [operands](#) must be [boolean](#).

Error C136 : Set element type must be scalar or subrange

[Set](#) elements must be [scalar](#) or [Sub-Range](#).

Error C137 : Set element types not compatible

The [set](#) element is not compatible with the set type.

Error C138 : Type of variable is not array

An [Array](#) variable type was expected, but the variable is not an array type.

Error C139 : Index type is not compatible with declaration

The [Array](#) index type does not match the array declaration.

Error C140 : Type of variable is not record

A [Record](#) variable type was expected, but the variable type is not a record.

Error C141 : Type of variable must be pointer

A [Pointer](#) variable type was expected, but the variable is not a pointer.

Error C142 : Illegal parameter substitution

The [Parameter](#) type is not valid.

Error C143 : Illegal type of loop control variable

Only [scalar](#) or [Sub-Range](#) types can be used as [loop](#) variables.

Error C144 : Illegal type of expression

Expression must be [scalar](#).

Error C145 : Type conflict

The [types](#) are not compatible.

Error C146 : Assignment of files not allowed

[Files](#) can not be assigned.

Error C147 : Label type incompatible with selecting expression

[Case](#) label type is not compatible with the case selecting expression.

Error C148 : Subrange bounds must be scalar

The [Sub-Range](#) bounds must be [scalar](#).

Error C149 : Index type must not be integer

[Array](#) index type must be [scalar](#), but not [Integer](#).

Error C150 : Assignment to standard function is not allowed

Can not assign a value to a standard [Function](#).

Error C151 : Assignment to formal function is not allowed

Can not assign to a formal [Function](#).

Error C152 : No such field in this record

The specified [Record](#) field does not exist.

Error C154 : Actual parameter must be a variable

The [Parameter](#) must be a variable.

Error C155 : Control variable must neither be formal nor non local

[For](#) statement control variable must be a local variable.

Error C156 : Multidefined case label

[Case](#) label has been defined more than once. Remove the duplicate case label.

Error C157 : Too many cases in case statement

The range of the [case](#) labels has exceeded the maximum. The difference between the lower bound and the upper bound of the case statement must be less than 1000.

Error C158 : Missing corresponding variant declaration

Error in [Pointer](#) "new" statement.

Error C159 : Real or string tagfields not allowed

Error in [Pointer](#) "new" statement.

Error C160 : Previous declaration was not forward

Function / Procedure was previously declared without a [Forward Declaration](#).

Error C161 : Again forward declared

Function / Procedure was previously declared with a [Forward Declaration](#).

Error C162 : Parameter size must be constant

Error in [Pointer](#) "new" statement.

Error C165 : Multidefined label

Label defined more than once. Labels are not available to the user in the logic engine, but are used in the implementation of the Modules. Remove any label definitions.

Error C166 : Multideclared label

Label declared more than once. Labels are not available to the user in the logic engine, but are used in the implementation of the Modules. Remove any label declarations.

Error C167 : Undeclared label

Label has not been declared. Labels are not available to the user in the logic engine, but are used in the implementation of the Modules. Remove the use of any labels.

Error C168 : Undefined label

Label has not been defined. Labels are not available to the user in the logic engine, but are used in the implementation of the Modules. Remove the use of any labels.

Error C169 : Error in base set

Base [Set](#) type is illegal.

Error C177 : You may only assign to the identifier of a function in the body of that function

A [Function](#) result can only be assigned within the body of the function.

Error C178 : Duplicated variant part in record declaration

[Record](#) variant part duplicated. Remove the duplication.

Error C179 : Function not supported for this Project type

This function is not supported for the selected Project type (HomeGate, Schedule Plus, Colour C-Touch, PAC, C-Touch Mark II). Some Project types only support particular functions. Refer to the relevant section of the help file for details of which functions are supported for which project types.

Error C180 : Once statements can only be used inside Modules

[Once Statements](#) only work within a [Module](#) and can not be used in the [Initialisation](#) section, [Functions](#) or [Procedures](#).

Error C181 : Use of reserved words "input" or "output"

The reserved word "input" or "output" has been used in the logic code. These words can not be used for [Identifiers](#).

Error C201 : Error in real constant: digit expected

A digit is missing from a [Real](#) constant.

Error C202 : String constant must not exceed source line

A [string](#) constant has gone onto the next line, or the closing ' is missing.

Error C203 : Integer constant exceeds range

An [Integer](#) constant has exceeded the allowed range (-2147483648 to 2147483647).

Error C250 : Too many nested scopes of identifiers

The nesting of [identifiers](#) has exceeded the limit (20).

Error C251 : Too many nested procedures and/or functions

The nesting of [procedures](#) and/or [functions](#) has exceeded the limit (20).

Error C254 : Too many long constants

The maximum number of long constants has been exceeded. A maximum of 1000 is allowed. A long constant is a non-integer constant, such as a [Real](#) number, a [Set](#) or a [String](#). These are not necessarily constants defined in the [Constants](#) section, but can be any constant value used in the

code, such as the string in :

```
WriteLn('Value = ', v);
```

Error C261 : Tag Error

There is an error in a [Tag](#). Ensure that the spelling and case is exactly correct.

Error C262 : Percent constant out of range (0 - 100)

A [percentage constant](#) can only be from 0 to 100.

Error C263 : System IO Type is invalid

The [System IO](#) variable type is invalid.

Error C265 : Tag must not exceed source line

A [Tag](#) has gone onto the next line, or the closing " is missing.

Error C266 : String constant too long

A [String Constant](#) is longer than the maximum allowed (255 characters).

Error C267 : Special Function not supported in logic

The selected [Special Function](#) is not supported by the logic engine.

Error C268 : Too many ConditionStaysTrue statements

There is a limit to how many [ConditionStaysTrue](#) functions can be used.

Error C269 : This in-built system IO variable can not be set

The [In-Built System IO Variable](#) is "read only" and can not be set.

Error C270 : Semicolon not allowed before "else"

A semicolon is not allowed immediately before an [else](#) statement.

Error C271 : Too many HasChanged statements

There is a limit to how many [HasChanged Functions](#) can be used..

Error C272 : Property Type is invalid

The [Page Property](#) or [Component Property](#) type does not match the function type.

Error C273 : Only Trigger Control Application can be used

Only the Trigger Control Application can be used for some C-Bus functions.

Error C274 : Invalid Level

C-Bus levels can only be between 0 and 255 (100%) inclusive.

Error C275 : Invalid Measurement Application Unit/Channel

The C-Bus Measurement Application Unit Id and/or Channel does not exist. Use the Measurement Application Manager to add this channel.

Error C304 : Element expression out of range

The [set](#) expression is outside of the allowable range.

Error C305 : Procedure only allowed within Modules

This procedure is only allowed with a [Module](#).

Error C399 : Feature not implemented

This [Pascal](#) feature has not been implemented.

Error C400, C500, C501 : Internal compiler error

An error has occurred within the compiler. Contact technical support for advice.

Scalar Types

A Scalar type is either an [Integer](#), [Boolean](#), [Char](#) or [Enumerated Type](#). It does not include [Real](#) or [String](#) types.

Compilation Warnings

A Compilation Warning is a warning that something is most likely a problem, but the logic will be allowed to run anyway. The compiler warnings are listed below :

Warning W001 : Tag too long (more than 31 characters) to be downloaded to Colour C-Touch.

The [tag](#) is too long to be able to be downloaded to Colour C-Touch. The tag will be shortened before being transferred to the Colour C-Touch. This will cause a compile error in the Colour C-Touch because the tag will no longer be recognised. To fix this, edit the tag using the C-Bus ToolKit and make it no more than 31 characters in length.

Warning W004 : Scene will use more than 75% of the PAC scan time.

Executing the Scene will use more than 75% of the PAC capacity. It is possible that this will cause the PAC to reset. See [How Much Logic Is Possible](#).

Warning W005 : It is recommended that a Scene be used for sending multiple C-Bus commands.

Sending a series of C-Bus commands as separate messages can be inefficient and makes the logic code longer and harder to read. It is recommended that if more than three C-Bus commands are to be sent, then a [Scene](#) should be used.

Warning W006 : Function not supported for this Project type

This function is not supported for the selected Project type (HomeGate, Schedule Plus, Colour C-Touch, PAC). Some Project types only support particular functions. Refer to the relevant section of the help file for details of which functions are supported for which project types. This warning is generated when the **Allow use of all Functions for Testing** [option](#) is selected instead of generating error C179 to allow the function to be used temporarily for testing purposes.

Warning W007 : Tag has spaces at start or end

A [Tag](#) has one or more spaces at the start or end of the Tag. These spaces are ignored, but they should be removed for "correctness".

Warning W008 : String constants longer than 50 characters may cause problems.

By default, [Strings](#) are 50 characters. Unless you have defined a string variable to be longer than this, using a string constant which is longer than 50 characters will cause it to be truncated.

6.2 Run Time Errors

There are three classes of run-time errors :

- Errors R000 - R099 are errors which can optionally be ignored.
- Errors R100 - R199 are critical errors, which can cause a re-start of the Logic Engine
- Errors R200 - R299 are load errors, and can not be recovered from

Errors

The errors in the range R001 - R099 are non-critical errors which the Logic Engine can often safely ignore. They will result in the Logic Engine not doing what you want, but the Logic Engine can recover and continue. The [Logic Engine Options](#) allow you to ignore these errors and continue, or to re-start the Logic Engine.

Error R002 : Invalid System IO number

The [System IO](#) number is invalid.

Error R003 : Invalid Timer number

The [Timer](#) number is invalid.

Error R004 : Invalid Module number

The [Module](#) number is invalid.

Error R005 : Invalid Scene number

The [Scene](#) number is invalid.

Error R006 : Invalid Page number

The [Page](#) number is invalid.

Error R007 : Invalid System I/O Type

The [System IO](#) type is not compatible.

Error R008 : Exceeded maximum number of Instructions

The maximum number of [instructions per scan](#) has been exceeded.

Error R009 : C-Bus Messages are being sent on every scan

[C-Bus](#) messages are being transmitted on every scan. This can cause problems on C-Bus by using too much bandwidth. See [Logic Engine Options](#).

Error R010 : Too many graphics commands are being executed

The maximum number of [graphics](#) commands have been exceeded. Check that a [ClearScreen](#) procedure is being used.

Error R011 : Minimum delay time is 0.2 second

The minimum [Delay](#) is 0.2 second. It is not possible to delay for less than this time.

Error R012 : Program execution failed

The [Execution](#) of a program failed. The reason for the failure will usually be given which will provide a clue as to the resolution of the problem.

Error R013 : Logic engine stalled. Catch-up started.

This error only occurs in a PAC or C-Touch. It results from the logic exceeding 100% of capacity for an extended period of time. See [How Much Logic Is Possible](#).

Error R014 : Socket Error

A socket function ([TCP/IP](#), [UDP](#), [Ping](#) or [DNS](#)) has failed. The error functions can be used to obtain socket error details:

- [ClientSocketError Function](#)
- [ServerSocketError Function](#)
- [UDPSError Function](#)
- [GetDNSLookupResult Function](#)

A message with some details will appear in the log.

Critical Errors

The errors in the range R100 - R199 are Critical Errors which the Logic Engine can not ignore. These errors will cause the Logic Engine to stop immediately. The user can select for the Logic Engine to automatically [re-start](#) following a Critical Error if required.

Error R100 : Other error

An unlisted type of error has occurred.

Error R101 : Reading from an output file

An attempt was made to read from an output [file](#).

Error R102 : Writing to an input file

An attempt was made to write to an input [file](#).

Error R103 : eof / eoln used on an output file

An [EOF](#) or [EOLN](#) function was used on an output [file](#).

Error R104 : Memory overflow

The amount of memory allocated has been fully used.

Error R105 : Divide by zero

A division by zero was attempted.

Error R106 : Illegal pointer value

A [pointer](#) was pointing to an illegal memory value.

Error R107 : Value out of range

A value was outside of the allowable range.

Error R108 : <, <=, >, >= used with an address

Illegal operations were attempted on an address.

Error R109 : File Not Found

The specified [file](#) could not be found.

Error R110 : Illegal set operation

An illegal operation was applied to a [set](#).

Error R111 : Invalid Serial Port number

The [Serial Port](#) number was invalid.

Error R199 : Logic has taken too long

You have too much logic to run in the PAC. See the [How Much Logic Is Possible](#) topic and the [Efficient Code](#) topic for details of reducing the size of logic code.

Load Errors

The errors in the range R200 - R299 are Load Errors which the Logic Engine can not ignore, and can not recover from. These errors occur during the loading of the logic into the [Interpreter](#). The user program must be changed in order for the Logic Engine to be able to run.

Error R200 : Duplicated label

A label was duplicated. This is an internal compiler error. Contact technical support for assistance.

Error R201 : Illegal instruction

An illegal instruction was found. This is an internal compiler error. Contact technical support for assistance.

Error R202 : Integer table overflow

Too many [integer](#) constants have been used.

Error R203 : Real table overflow

Too many [real](#) constants have been used.

Error R204 : Illegal character

An illegal character was found. This is an internal compiler error. Contact technical support for assistance.

Error R205 : Set table overflow

Too many [set](#) constants have been used.

Error R206 : Boundary table overflow

Too many [arrays](#) have been used.

Error R207 : String table overflow

Too many [string](#) constants have been used.

Error R208 : Too much logic to run

You have too much logic to run in the logic engine. See the [Efficient Code](#) topic for details of reducing the size of logic code.

6.3 Resolving Compilation Errors

When a [Compilation Error](#) occurs, there will be an error message in the [Output Window](#). The error number will be displayed, along with the line it which it occurred. To find the source of an error, double click on the error message in the Output Window. The code will be displayed in the [Code Window](#), with the cursor at the position of the error.

Always start resolving the error with the first error in the list. A single error can cause a cascade of error messages, and the later ones may be difficult to identify the cause.

Sometimes it is not clear exactly where the error occurs, and the actual cause of the error may be some way before the point at which the compiler reported the error.

EXAMPLES

If you had :

```
procedure Proc1;  
begin  
  { some code goes here }  
end      { missing semi-colon }
```

```
procedure Proc2;  
begin  
  { some code goes here }  
end;
```

You would get an error pointing to the start of the Proc2 declaration, even though the error is a missing semi-colon at the end of the line in the previous procedure (Proc1).

The following code is an example of a very common cause of errors :

```
if GetLightingLevel("Outside") > 50% and Time = "9:00PM" then
begin
  { some code goes here }
end;
```

An error C004 ("illegal symbol") will indicate that there is a problem at the "=" sign. The root cause of the problem is that the brackets around the two terms are missing. The code should read :

```
if (GetLightingLevel("Outside") > 50%) and (Time = "9:00PM") then...
```

The reason for the compiler reporting the error that it does is related to the way the expression gets evaluated. The "and" has a higher [Operator Precedence](#) than the ">" or "=" operators. This means that the expression is evaluated as :

```
if GetLightingLevel("Outside") > (50% and Time) = "9:00PM" then...
```

In this case, the "and" will be interpreted as a [Bitwise Operator](#). This is perfectly legal code up until the point of the "=", which is why the compiler reports the message there.

7 FAQ

The following sections cover some Frequently Asked Questions regarding the use of the Logic Engine.

7.1 When to use logic

You should generally only use logic when it is needed. If your requirements can be met through the use of Scenes, Schedules, Irrigation or other in-built software features, then you should use the built-in features. There are several reasons for this :

1. The in-built features are much faster and more efficient
2. The in-built features have editors which make them easier to view and edit
3. The in-built features have been thoroughly tested. Writing your own code introduces the possibility of bugs.

7.2 Using Counters

A counter is just a [Variable](#) which is used for counting some event. For example, to count how often the "Scene Control" group address goes on and then trigger three different Scenes, you would need to do the following steps.

Declare the counter variable

In the Variables node of the [Logic Tree](#), enter the declaration for an integer [Variable](#) :

```
PressCounter : integer;
```

Initialise the counter

In the [initialisation](#) node of the Logic Tree, initialise the value of the counter :

```
PressCounter := 0;
```

Increment the counter

When the "Scene Control" Group gets switched on, increment (add one to) the counter variable (this code is in a [Module](#)) :

```
once GetLightingState("Scene Control") = ON then
begin
  PressCounter := PressCounter + 1;
  ...
```

Use the counter

Use the value of the counter variable to determine which Scene to trigger. When the counter gets to the last value, it is important to reset the counter again.

```
once GetLightingState("Scene Control") = ON then
begin
  PressCounter := PressCounter + 1;
  if PressCounter = 1 then
    SetScene("Scene 1");
  if PressCounter = 2 then
    SetScene("Scene 2");
  if PressCounter = 3 then
begin
```

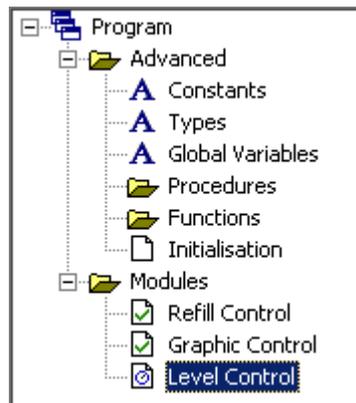
```

SetScene("Scene 3");
PressCounter := 0;
end;
end;

```

7.3 Program Execution

When the logic first starts, it runs the [Initialisation](#) code. From then on, each time the logic is run (each [scan](#)), the code in each of the [Modules](#) is run in the order in which they appear in the [Logic Tree](#). In the case of a logic tree like :



the order of the Modules will be :

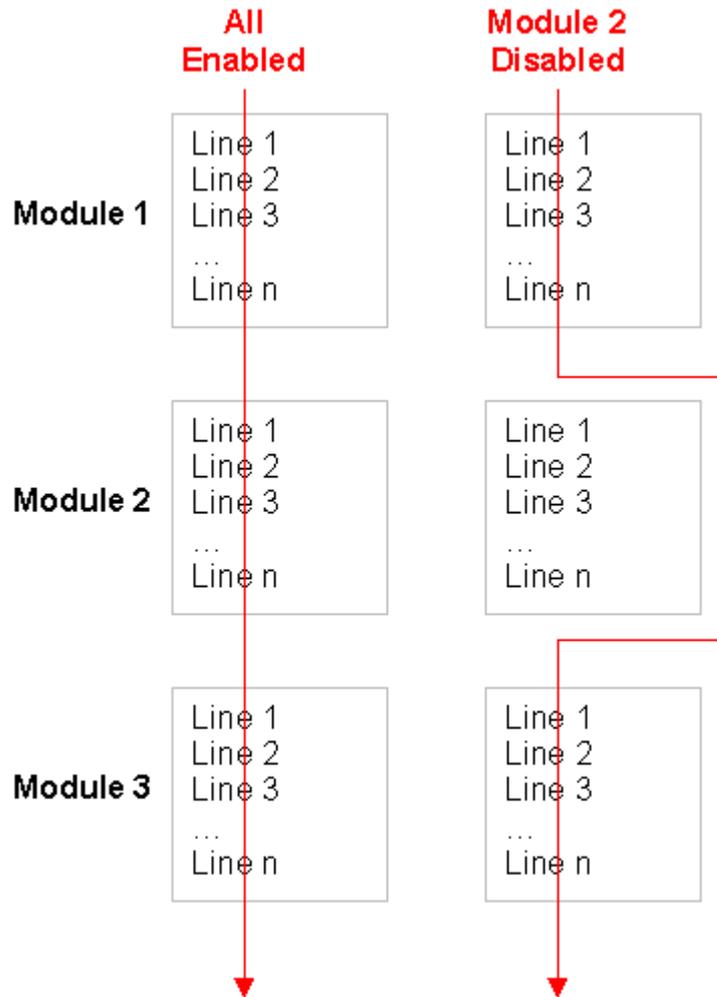
1. Refill Control
2. Graphic Control
3. Level Control

The code in each Module is run from the top line of code to the bottom line of code each scan. The only exceptions to this are :

1. Disabled Modules : when a Module has been [disabled](#), it does not get run until it has been [enabled](#) again. This does not affect the operation of other Modules.
2. When a Module is paused : When a module is paused due to a [Delay Procedure](#) or a [WaitUntil Procedure](#), the module will stop at that point of the code until the delay is complete. When the delay is complete, the code will continue again. A delay in one Module does not affect any other Modules.
3. The [ExitModule Procedure](#) causes all of the rest of the code in the Module to be skipped. This does not affect any other Modules.

Enabled and Disabled Modules

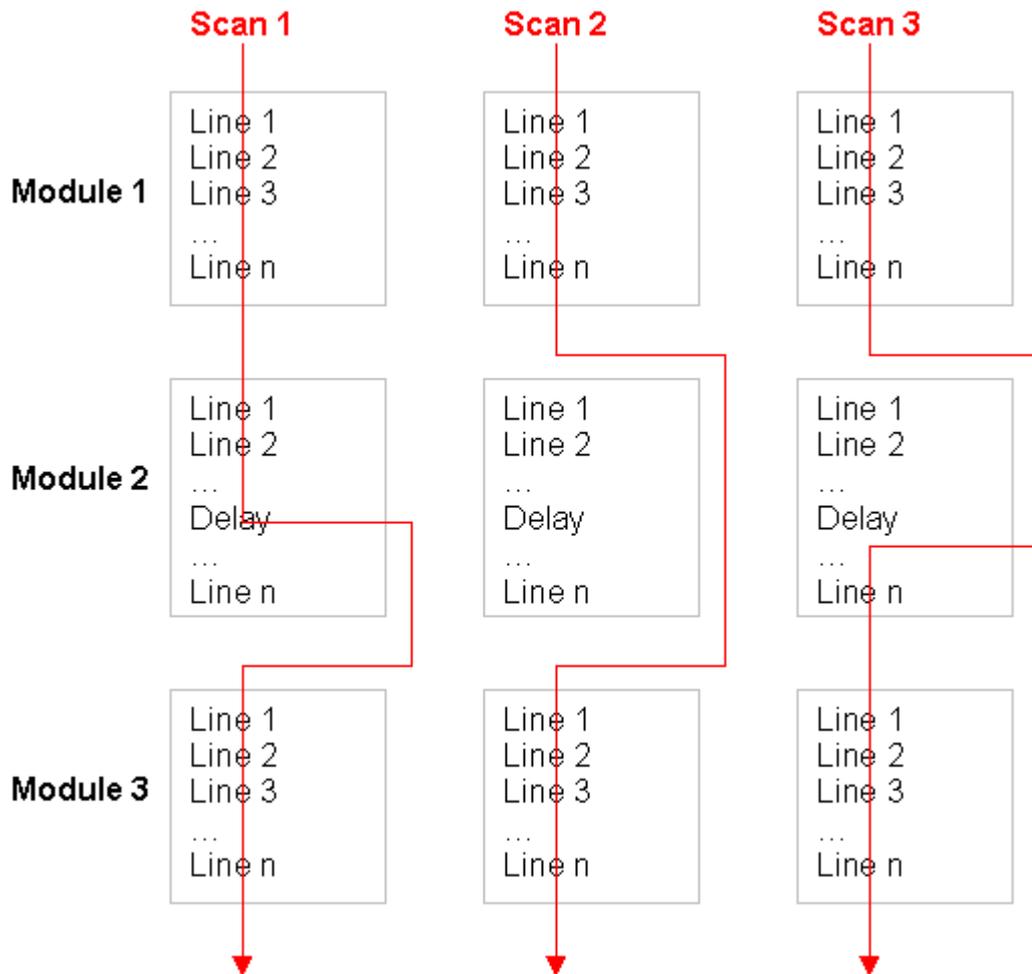
If a Module is [disabled](#) that Module will not be run until it is [enabled](#) again. A disabled Module does not affect any other Modules. The diagram below shows this. The red line shows the order of execution of the program for a single scan.



Delayed Modules

If there is a [delay](#) (or [WaitUntil Procedure](#)) in a Module, the rest of that Module will not be run until the delay is finished. This delay does not affect any other Modules. The diagram below shows this.

All of Module 1 and 3 runs on every scan. On the first scan, the code in Module 2 up to the point of the delay runs, then the rest of Module 2 is ignored and the execution jumps to Module 3. On the second scan, all of Module 2 is ignored, because it is still delaying. When the delay has finished (scan 3), the rest of Module 2 is run. The next scan will be the same as scan 1 again.



7.4 Random Event Times

Implementing Random Event Times

The random number function can be used to perform an action at a random time. A typical example is to switch lights on at random times when nobody is at home to make a house look "lived in".

To switch on the lounge room light for two hours starting at a random time between 7:00PM and 8:00PM, the following code could mistakenly be used :

```

if time = "7:00:00PM" + random("1:00:00") then    { This will not work ! }
begin
  SetLightingState("Lounge", ON);
  Delay("2:00:00");
  SetLightingState("Lounge", OFF);
end;

```

Since the random value will be different each scan, it is possible that the condition will never be true and the statements will not be executed. To make this style of code work properly, a variable needs to be used which contains the random value and is not changed each time the expression is evaluated. For example :

In the [Initialisation](#) section :

```

RandomTime := random("1:00:00");

```

In the [Module](#) section :

```

if time = "7:00:00PM" + RandomTime then
begin
  SetLightingState("Lounge", ON);
  Delay("2:00:00");
  SetLightingState("Lounge", OFF);
  RandomTime := random("1:00:00");
end;

```

The above code still has a weakness in that if the 2 hour delay was changed to be less than an hour, then the code could be run more than once each day. This is probably not what is required.

For example, if there was only a 10 minute delay and if the RandomTime variable was 300 (5 minutes), then the code would be executed at 7:05PM. After a delay of 10 minutes (7:15PM), the RandomTime variable gets calculated again. If it gets a new value of 900 (15 minutes), then the code will be run again at 7:30PM.

The Easy Way of Doing Random Start Times

The simplest way to implement the above requirements is :

```

if time = "7:00:00PM" then
begin
  Delay(random("1:00:00") + 1);
  SetLightingState("Lounge", ON);
  Delay("2:00:00");
  SetLightingState("Lounge", OFF);
end;

```

In this case, once the time gets to 7PM, there is a random delay of up to an hour, followed by the other actions. During the [delay](#) period, this [Module](#) will wait for the delay to be complete, but other Modules will continue.

Note that the value of `random("1:00:00")` could be anywhere between 0 and 3599. A delay of zero is not allowed, so a value of 1 is added in the code above.

If you want to only do the random timing if a flag called AwayMode is set, then you could use :

```

if AwayMode and (time = "7:00:00PM") then
begin
  Delay(random("1:00:00") + 1);
  SetLightingState("Lounge", ON);
  Delay("2:00:00");
  SetLightingState("Lounge", OFF);
end;

```

In this case, once the delay has started, the rest of the events will occur even if the AwayMode flag changes during the delay. This is probably not what is desired. A better method would be :

```

if time = "7:00:00PM" then
begin
  Delay(random("1:00:00") + 1);
  if AwayMode then
    SetLightingState("Lounge", ON);
  Delay("2:00:00");
  if AwayMode then

```

```

    SetLightingState("Lounge", OFF);
end;
```

7.5 Logic Engine Security

The Logic Engine operates within a "sand box" which means that it has a restricted environment to protect other applications from it and vice versa. Specifically, the Logic Engine restrictions are :

- It can only access a section of memory allocated exclusively to it
- It can only access files in the project directory
- It is only allowed to use a certain amount of processor time

However there are potential "back doors" which could cause security weaknesses :

- The Logic Engine can start other applications, which could potentially be malicious. Ensure that any program or file that you open from the Logic Engine is trustworthy.
- TCP/IP sockets and Serial (RS232) connections can be used to communicate with the Logic Engine. The code performing this communication is entirely at the user's discretion, so care should be taken as to what can be controlled via these connections.

7.6 Handling Triggers

Because the Logic Engine is not event driven, if two successive C-Bus commands set a C-Bus Group Address to the same level, then the second one can be missed. If it is important to react to each of these events, then it is necessary to set the Group Address to a different level after dealing with an event.

Consider the following code :

```

    once GetTriggerLevel("Logic Triggers") = 255 then
        Macro1;
```

In this case, if the "Logic Triggers" Group Address is set to level 255, then the Macro1 will be executed. If the "Logic Triggers" Group Address is set to level 255 again, nothing will happen. The "Logic Triggers" Group Address needs to be set to a value other than 255 and then back to 255 before Macro1 will be executed again.

If the code is changed to :

```

    once GetTriggerLevel("Logic Triggers") = 255 then
    begin
        Macro1;
        SetTriggerLevel("Logic Triggers", 0);
    end;
```

then the level will be reset to 0 after Macro1 is executed. The next time that the Logic Triggers Group Address is set to level 255, Macro1 will be executed again.

The other possible problem with this code is that if the trigger is set on the first scan of the Logic Engine, then Macro1 will never run unless something externally sets the trigger to level 0 then back to level 255 again. The [once](#) statement is only executed when an false to true change is seen. A more reliable way of writing the above code would be :

```

    if GetTriggerLevel("Logic Triggers") = 255 then
    begin
        Macro1;
        SetTriggerLevel("Logic Triggers", 0);
    end;
```

Because the level has been set back to 0 each time, there is no problem with the code being executed on every [scan](#).

See also [Controlling Modules from Components or Schedules](#)

7.7 Logic Catch-up

If the PC clock is adjusted forward (for example to set the time correctly, or for Daylight Savings), there could be the possibility of events being missed. The Schedule Catch-up process ensures that any Schedules or Logic which should have been executed during this time will be executed.

7.8 Handling Sets of Loads

Sometimes it is necessary to handle a set of Loads to do things like :

- Finding whether any lights are on
- Setting some loads to a particular level

These functions can often be handled through the use of [C-Bus Scene Functions](#). To handle a set of loads, a Scene is created containing all of the Group Addresses to be used. The Scene is then used to contain the set of Group Addresses and to control or monitor them together.

Examples

To set the level of a set of Loads (in a Scene), use the [SetSceneLevel Procedure](#).

To determine whether all/any of the Group Addresses in a Scene are on/off, use the [GetSceneMaxLevel](#) or [GetSceneMinLevel](#) functions.

To record the level of a series of loads, use the [StoreScene Procedure](#) and then you can later use the [SetScene Procedure](#) to restore the levels to what they were.

7.9 Controlling Modules from Components or Schedules

There are several mechanisms for controlling a Module from a PICED Component or Schedule. This section describes the alternative methods for executing a series of actions (in a Module) when it is "triggered" by a Schedule or by a user clicking on a Component.

Using Group Addresses

A Component or Schedule can set a C-Bus Group Address to a particular level which can then be used within a Module to initiate some actions. The problem with this approach is that C-Bus commands get sent out onto C-Bus, which is not always appropriate. To make this work :

- The Schedule or Component sets a C-Bus Group to a particular level (usually 100%)
- The Module has an [If Statement](#) to check whether the Group is at the correct level
- The Module resets the Group level when it is complete

See also [Handling Triggers](#)

Using System I/O Variables

Almost the same thing can be done using System I/O Variables (with Components, not with Schedules). This has the advantage that no messages are sent onto C-Bus. To make this work :

- The Component sets a System I/O variable to a particular level (usually a boolean variable set to TRUE)
- The Module has an [If Statement](#) to check whether the System I/O variable is at the correct level
- The Module resets the System I/O variable value when it is complete

Using Special Functions

Special Functions can be used to enable and disable a Module. This approach is often the best where a series of actions have to be performed by a Schedule or when a Component is clicked. To make this work :

- The Schedule or Component has the "Enable Module" Special Function
- The Module is disabled in the [Initialisation](#) section
- When the Schedule or Component enables the Module, it will be executed on the next [scan](#)
- The Module then disables itself when it is complete

If a Module is controlled by a Special Function in a Schedule or from a Component, the [module icon](#) will have a + symbol in it when in editing mode.

7.10 Running Modules Infrequently

If a module only needs to be run occasionally (say once per minute), the simplest method is to just place a [Delay](#) at the end of the Module code. When the Module is complete, it will delay for the specified duration before running again.

7.11 Simplifying Logic Conditions

There are various ways that [boolean](#) expressions can be simplified. A boolean expression with less terms is quicker to evaluate, and is less likely to have errors. The rules for logic expressions are shown below.

Distributive Rule

A and (B or C) = (A and B) or (A and C)

DeMorgan's Rules

not (A and B) = not A or not B

not (A or B) = not A and not B

Applying the Rules

By looking at a boolean expression, it is often possible to see the above patterns and use these to simplify the expression.

Example

For example, consider the condition :

```
if (date = ScheduleDate) and (time = "7:00") or (date = ScheduleDate) and
(time = "19:00") then ...
```

The above expression can use the Distributive Rule to simplify it, since it is of the form :

$$(A \text{ and } B) \text{ or } (A \text{ and } C)$$

In this case :

```
A is (date = ScheduleDate)
B is (time = "7:00")
```

```
C is (time = "19:00")
```

So by applying the rule, the above is equivalent to

```
A and (B or C)
```

which is :

```
if (date = ScheduleDate) and ((time = "7:00") or (time = "19:00")) then ...
```

7.12 Efficient Code

There are many things that can be done to make your Logic code more efficient, and hence to run faster.

Simplify Logic

Logic Expressions can be [Simplified](#) to make them evaluate more quickly. This can make the code easier to read too.

Run Less Frequently

Modules can be [Run Less Frequently](#) where appropriate. If a module only needs to run every minute or so, then do so.

Nesting Conditions

Where there is a complex condition, it is more efficient to nest the condition if possible. For example, if you had a complex Schedule like :

```
if (time = StartTime) and (Day >= 1) and (Day <= 7) and (Month >= "June") and
(Month <= "August") and (GetLightingLevel("Porch") > 50%) then ...
```

Each [scan](#), all of the 6 conditions that make up the expression have to be evaluated. If the condition was re-written as :

```
if (time = StartTime) then
  if (Day >= 1) and (Day <= 7) and (Month >= "June") and (Month <= "August") and
  (GetLightingLevel("Porch") > 50%) then ...
```

Then only the first condition is evaluated each scan, and when the condition is true (only once per day), then the rest will be evaluated. This reduces the amount of processing time to 1/6. Ideally, the first condition should be the one that occurs the [least](#) often. If it had been written as :

```
if (GetLightingLevel("Porch") > 50%) then
  if (time = StartTime) and (Day >= 1) and (Day <= 7) and (Month >= "June") and
  (Month <= "August") then ...
```

then the whole thing would be evaluated every scan while the "porch" light is on.

Using Variables

If you have a calculation that is used many times throughout you code, it can be more efficient to assign it to a variable, then use the variable.

For example, if you have logic that depends on whether it is currently work hours, then you may have code like :

```
if (DayOfWeek >= "Monday") and (DayOfWeek <= "Friday") and not IsSpecialDayType
(Date, "Public Holiday") and (Time >= "9:00AM") and (Time <- "5:30PM") and
GetTriggerState("Scene 1") then ...
```

```
if (DayOfWeek >= "Monday") and (DayOfWeek <= "Friday") and not IsSpecialDayType
(Date, "Public Holiday") and (Time >= "9:00AM") and (Time <- "5:30PM") and
GetTriggerState("Scene 2") then ...
```

This can be made faster and easier to read by defining a boolean variable called WorkingHours and writing :

```
WorkingHours := (DayOfWeek >= "Monday") and (DayOfWeek <= "Friday") and not
IsSpecialDayType(Date, "Public Holiday") and (Time >= "9:00AM") and (Time <-
"5:30PM");
```

```
if WorkingHours and GetTriggerState("Scene 1") then ...
```

```
if WorkingHours and GetTriggerState("Scene 2") then ...
```

Using Scenes

If you have code which is setting a series of Group Addresses, it is much more efficient to [Set a Scene](#). This has the advantages that :

- It makes the code more compact
- It makes the code more readable
- It is much easier to edit a scene than to change lines of logic
- It executes much faster

7.13 Fixing Errors

Most programs have [errors](#) in them. The most common problem is [Compilation Errors](#).

The logic engine provides various means of [finding and fixing](#) errors.

There are various [Debugging Methods](#) which can be used to track down [logical errors](#) within your program.

7.14 Tracking a Group Address

One-way tracking

To get one Group Address to track another, the simplest method is to use the [TrackGroup Procedure](#). For example, to get Group Address 2 to follow the value of Group Address 1 if the Track variable is true :

```
if Track then
    TrackGroup("Local Network", "Lighting", 1, 2);
```

Alternatively, the [GetCBusTargetLevel](#) and [GetCBusRampRate](#) functions can be used. The code to do the same as the above would be :

```
if Track then
    if GetCBusTargetLevel("Local Network", "Lighting", 2) <>
GetCBusTargetLevel("Local Network", "Lighting", 1) then
        SetLightingLevel(2, GetCBusTargetLevel("Local Network", "Lighting", 1),
GetCBusRampRate("Local Network", "Lighting", 1));
```

Two-way Tracking

To get two Group Addresses to track each other, the simplest method is to use the [TrackGroup2 Procedure](#). For example, to get Group Address 1 and Group Address 2 to follow each other if the Track variable is true :

```
if Track then
    TrackGroup2("Local Network", "Lighting", 1, "Local Network", "Lighting",
2);
```

Example

A common use for this requirement is where two rooms are joined by a removable partition. In this case, when the partition is closed, you want the lights in the two halves of the room to be independent. When you open the partition, you want the lights to operate together. If we had a sensor on the partition controlling a group "partition" which is on when closed, then you could get the lights in room 1 and those in room 2 to track each other as follows :

```
if GetLightingState("partition") = false then
    TrackGroup2("Local Network", "Lighting", "Room 1", "Local Network",
"Lighting", "Room 2");
```

If there were three rooms joined by two partitions, the code would be :

```
{ Rooms 1 and 2 joined, Room 3 separate }
if (GetLightingState("partition 1") = false) and (GetLightingState
("partition 2") = true) then
    TrackGroup2("Local Network", "Lighting", "Room 1", "Local Network",
"Lighting", "Room 2");

{ Rooms 2 and 3 joined, Room 1 separate }
if (GetLightingState("partition 1") = true) and (GetLightingState("partition
2") = false) then
    TrackGroup2("Local Network", "Lighting", "Room 2", "Local Network",
"Lighting", "Room 3");

{ Rooms 1, 2 and 3 joined }
if (GetLightingState("partition 1") = false) and (GetLightingState
("partition 2") = false) then
begin
    TrackGroup2("Local Network", "Lighting", "Room 1", "Local Network",
"Lighting", "Room 2");
    TrackGroup2("Local Network", "Lighting", "Room 1", "Local Network",
"Lighting", "Room 3");
end;
```

7.15 Logic Templates

Logic code can be saved as a Template. This enables code to be re-used on future projects. When writing code which is designed to be re-used, there are several things to consider, as listed below.

Using Constants

Where possible, numbers should be represented with [constants](#). This means that the value of the constant can be changed in one place, and the rest of the code will work correctly. For example, if you had some code which used some zones, your code might be :

```

{ constants section }
ZoneCount = 5;

{ var section }
ZoneState : array[1..ZoneCount] of integer;

{ module section }
for ZoneNo := 1 to ZoneCount do
  if ZoneState[ZoneNo] > 0 then
    ...

```

In the above code, you only need to change the value of ZoneCount in one place if you want to change the number of zones. It also makes the code easier to understand.

Similarly, constants can be used for C-Bus Group Addresses. For example, if you have some template code to control a bathroom exhaust fan, you could use a constant for the fan Group Address instead of a [Tag](#) :

```

{ constants section }
BathroomFan = 22;

{ module section }
...
SetLightingState(BathroomFan, On);
...

```

When someone uses a template with the above code in it, they just need to change the value for the BathroomFan and the code will work correctly.

Comments

When you work on code which was written by someone else, or even by yourself some time ago, it can be quite difficult to work out what the code is doing. Frequent [comments](#) throughout the code will make it easier to re-use code.

Name Space

If you have a series of logic templates which use the same [variable](#), [procedure](#) or [function](#) names, you will not be able to load them at the same time. This is because you will have multiple declarations of the same variable, procedure or function. To avoid this, you should use names that will be unique for the template you are designing. For example, if you are designing a template to handle a bathroom fan timer, you may want to prefix all variables with some letters which relate to the bathroom fan timer :

```

{ var section }
bft_LightOnTime : integer;
bft_FanDuration : integer;
bft_FanIsOn : boolean;

```

The chances are that these names will not be used in any other templates, and you will not get a conflict between the names.

7.16 How Much Logic Is Possible

There is no definite answer to how much logic code is possible, as it depends on many factors including :

- The speed of your computer
- The logic functions used

- How the code has been written

You can use the [Resource Window](#) to see how much of the resources are being used.

There are several basic limits to the amount of code :

- The amount of [memory](#) available
- The amount of code storage space available
- How long the code takes to run (in the PAC, if it takes too long to run, it will be terminated)
- Software Limits for some features

See the [Efficient Code](#) topic for ideas on how to get more code into the logic engine.

Note that it is possible to run a lot more logic on your computer than in a Colour C-Touch or particularly in a PAC.

PAC

Estimating Usage

You can get an idea of how much of a scan typical statements take in a PAC by looking at the table below :

Statement	Approximate % of a scan taken	Comment
C-Bus Commands		
SetLightingLevel("light", 100%, 4);	1.2%	
SetCBusLevel("local", "lighting", "light", 100%, 4);	1.3%	The SetLightingLevel is faster
SetScene("All On");	1.0% per scene item	Faster than sending individual commands. Less code is needed too.
TrackGroup("Local", "Lighting", "Group 1", "Group 2");	0.4% / 2.4%	Larger value is if the tracking group needs to be changed
Conditions		
once / if	0.1%	"if" is slightly faster than "once"
GetLightingLevel("light") = 100%	0.2%	
GetCBusLevel("local", "lighting", "light") = 100%	0.3%	
GetLightingState("light") = ON	0.2%	
GetLightingState("light")	0.1%	This is quicker than above because no comparison (= ON) is used
GetSceneLevel("floor 1") = 0%	0.1% per scene item	For 10 scene items in the scene. Much faster than a series of GetCBusLevel commands.
time = "7:00 PM"	0.2%	
ConditionStaysTrue(condition, "0:01:00")	0.3%	Not including the evaluation of the condition
Variables		
Flag := true;	0.1%	

Counter := 0;	0.1%	
Counter := Counter + 1;	0.2%	
Other		
Each additional module used	1%	
begin / end	0%	
Comments or blank lines	0%	

By adding up the individual items that occur in the worst case scan, you can get an idea of whether a particular amount of logic is likely to fit in a PAC. Consider the code :

```

once GetLightingLevel("Start") = 100% then
begin
  SetLightingLevel("Room 1", 100%);
  SetLightingLevel("Room 2", 100%);
  SetLightingLevel("Room 3", 100%);
  SetLightingLevel("Room 4", 100%);
end;

```

On the scan when "Start" goes to 100%, this will take approximately 9.4% of the PAC capacity (including 1% for the module it is in). On the other scans, it will only take 1.4% of the PAC capacity.

When the logic is running, the [Resource Window](#) shows a rough estimate of how much of PAC capacity the logic will use. You will need to leave the logic running and test that it doesn't exceed 75% (to allow some spare capacity). The [Logic Engine Options](#) form allows you to select whether you want to receive warnings if the PAC usage becomes excessive. To be confident that the PAC has the capacity to run your logic, you will need to test all aspects of the logic. This requires exercising every function of the logic and ensuring that the PAC usage does not exceed 75%. To be completely sure, you will need to transfer the project to the PAC and repeat the tests on the actual site.

Measuring the Actual Usage

The actual usage in the PAC can be monitored by connecting to the PAC and selecting the **Transfer | Control Unit | Log PAC Messages** option. This will log the PAC usage whenever it exceeds 75%. It is necessary to test the worst-case conditions to be sure that the PAC will be OK under all circumstances. See Controlling the PAC in the main help file.

Getting more code to run

If you have tasks which do not need to be executed often, you can place them in a module with a delay at the end. This will ensure that the code is not executed on every scan.

Alternatively, if you have a series of tasks which are too much to execute all in one scan, you can write some code like :

```

{ var section }
ScanCount : integer;

{ initialisation section }
ScanCount := 1;

{ Module }
case ScanCount of
  1 : begin
      { code for the first scan here }
    end;
  2 : begin

```

```

        { code for the second scan here }
    end;
3 : begin
    { code for the third scan here }
    end;
4 : begin
    { code for the fourth scan here }
    end;
end;
{ increment the scan counter }
ScanCount := ScanCount + 1;
if ScanCount = 5 then
    ScanCount := 1;

```

The above code can be generated automatically using the **Structures | Alternate Code** pop-up menu item.

The disadvantage of using this technique is that each part of the code is run less often and hence there will be an increased delay between a given event and logic reacting to it.

Setting lots of Scenes

Setting large scenes can use up a lot of the PAC resources. If there is a possibility of several scenes being set in the same scan, then the PAC may even reset itself.

If you have code like :

```

once GetLightingState("Group 1") then
    SetScene("Scene 1");
once GetLightingState("Group 2") then
    SetScene("Scene 2");
once GetLightingState("Group 3") then
    SetScene("Scene 3");

```

you could conceivably have 3 scenes executed in the same scan, if Group 1, 2 and 3 went on at the same time.

If you changed it to :

```

once GetLightingState("Group 1") then
begin
    SetScene("Scene 1");
    ExitModule;
end;
once GetLightingState("Group 2") then
begin
    SetScene("Scene 2");
    ExitModule;
end;
once GetLightingState("Group 3") then
begin
    SetScene("Scene 3");
    ExitModule;
end;

```

then you could never have more than one scene set on a particular scan. If Group 1, 2 and 3 went true at the same time, on the next scan Scene 1 would get set and then it would exit the module. One the following scan, Scene 2 would get set and on the third scan, Scene 3 would get set.

Short and Long Term Maximum Usage

The PAC and C-Touch do allow the code to use more than 100% of a scan occasionally, as long as the longer term average (over several seconds) does not exceed 100%.

The PAC will allow the logic to run for up to 333% of a scan occasionally and the C-Touch will allow the code to run for up to 200% occasionally.

Most logic code has occasions when it needs to run considerably more logic than normal. An example of this is when a large Scene is being set. In this case, the logic may only require 20% for most of the time, but may need over 100% of a scan when the Scene gets set.

C-Touch Mark 2

The C-Touch Mark 2 unit can run around 5 times as much code as the PAC.

7.17 Function indices start from 0, not 1

The following can use a [Tag](#) or an [Integer](#) as their argument ([parameter](#)):

- [EnableModule](#) and [DisableModule](#) procedures
- [ModuleEnabled Function](#)
- [ShowingPage Function](#)
- [ShowPage Procedure](#)
- [SetSceneLevel](#), [SetSceneOffset](#) and [SetScene](#) procedure
- [System IO Functions](#)
- [ExecuteSpecialFunction Procedure](#)
- [IsSpecialDayType Function](#)

In all cases, the parameter index starts from 0, not from 1. For example, if you have 3 scenes, called "Scene 1", "Scene 2" and "Scene 3". If you wanted to set "Scene 2", you could use :

```
SetScene("Scene 2");
```

or

```
SetScene(1); // note it is not 2
```

It is recommended that you always use tags, rather than numbers. The reasons include :

- You are more likely to get the number wrong than the tag
- If the order of the items changes, the number will be wrong, but the tag will still be right

7.18 Displaying logic data

There are two ways of displaying logic engine data (for example, the content of a [String](#) variable):

1. Using [DrawText](#) is very flexible, but requires code to be written.
2. A Level / Value indicator can be used to show the text for a string [User System IO variable](#). This requires no coding and you can drag it to wherever you want and set the font, size etc. You place it over the top of a button to make it look like the text is part of the button if required. You can use the "word wrap" option if the text may be wider than the button.

8 Appendix

8.1 Hexadecimal Numbers

A Hexadecimal number is a number represented in "base 16". Everyday numbers are represented in decimal, which is base 10. In the decimal system numbers are expressed with 10 symbols; the familiar digits 0-9. The hexadecimal system uses 16 symbols, the ten digits plus five letters (A to F) to stand for additional "digits".

In the decimal system, once the number 9 (the last digit) is reached, a symbol has to be placed in the next column (the "tens" column) to create a number with two digits. In the hexadecimal system, in exactly the same way, once the number "F" (the last digit) is reached a symbol must be placed in the next column (the "sixteens" column).

Hexadecimal [Constants](#) are expressed with a \$ sign at the front.

A comparison between the two is shown below (decimal = hexadecimal) :

Decimal	Hexadecimal (2 digits)
0	\$00
1	\$01
:	:
9	\$09
10	\$0A
11	\$0B
12	\$0C
13	\$0D
14	\$0E
15	\$0F
16	\$10
17	\$11
18	\$12
:	:
254	\$FE
255	\$FF

8.2 Binary Numbers

Binary numbers are numbers represented as base 2. The symbols used in a binary number are 0 and 1. Each bit (binary digit) of a binary number represents a power of 2. For example :

Decimal	Binary (8 bits)
0	00000000
1	00000001
2	00000010
3	00000011
4	00000100
5	00000101
:	:
254	11111110
255	11111111

8.3 Character and String Formats

There are many possible ways of representing characters and strings, including:

- [ASCII](#)
- [Unicode](#)
- [UTF-8](#)
- [UTF-16](#)

Prior to V4.7 of the PICED software, only ASCII characters could be used in logic. From V4.7 onwards, all characters are [Unicode](#) (UTF-16), and can also be converted to [UTF-8](#) for use with:

- Reading and writing data from [files](#)
- Reading and writing data from [serial ports](#)
- Reading and writing data from [sockets](#)

Logic in C-Touch, PAC and Wiser only support ASCII.

 Note that Unicode characters can not be used in C-Bus Tags.

8.3.1 ASCII

ASCII stands for American Standard Code for Information Interchange. Computers can only understand numbers, so an ASCII code is the numerical representation of a character such as 'a' or '@' or an action of some sort.

ASCII characters are often expressed as [hexadecimal](#) values. To determine the ASCII character for a hexadecimal value of 0A (for example), convert the value to decimal (10) and then look it up in the tables below (line feed).

Non-printing characters

ASCII was developed a long time ago and now the non-printing characters are rarely used for their original purpose.

Value	Symbol	Meaning
0	NUL	
1	SOH	start of header
2	STX	start of text
3	ETX	end of text
4	EOT	end of transmission
5	ENQ	enquiry
6	ACK	acknowledge
7	BEL	bell
8	BS	backspace
9	HT	horizontal tab
10	LF	line feed
11	VT	vertical tab
12	FF	form feed
13	CR	carriage return
14	SO	shift out
15	SI	shift in
16	DLE	data link escape
17	DC1	no assignment, but usually XON
18	DC2	
19	DC3	no assignment, but usually XOFF
20	DC4	
21	NAK	negative acknowledge

22	SYN	synchronous idle
23	ETB	end of transmission block
24	CAN	cancel
25	EM	end of medium
26	SUB	substitute
27	ESC	escape
28	FS	file separator
29	GS	group separator
30	RS	record separator
31	US	unit separator

Printable Characters

Value	Symbol	Value	Symbol	Value	Symbol
32	space	64	@	96	`
33	!	65	A	97	a
34	"	66	B	98	b
35	#	67	C	99	c
36	\$	68	D	100	d
37	%	69	E	101	e
38	&	70	F	102	f
39	'	71	G	103	g
40	(72	H	104	h
41)	73	I	105	i
42	*	74	J	106	j
43	+	75	K	107	k
44	,	76	L	108	l
45	-	77	M	109	m
46	.	78	N	110	n
47	/	79	O	111	o
48	0	80	P	112	p
49	1	81	Q	113	q
50	2	82	R	114	r
51	3	83	S	115	s
52	4	84	T	116	t
53	5	85	U	117	u
54	6	86	V	118	v
55	7	87	W	119	w
56	8	88	X	120	x
57	9	89	Y	121	y
58	:	90	Z	122	z
59	;	91	[123	{
60	<	92	\	124	
61	=	93]	125	}
62	>	94	^	126	~
63	?	95	_	127	delete

8.3.2 Unicode

Unicode is a computing industry standard allowing computers to use text expressed in most of the world's writing systems.

Unicode can be represented by different character encodings. The most commonly used encodings

are:

- [UTF-8](#) : this uses 1 byte for all [ASCII](#) characters (which have the same code values as in the standard ASCII), and up to 4 bytes for other characters
- [UTF-16](#) : this uses 2 or 4 bytes to encode characters. The Logic Engine uses UTF-16.

8.3.3 UTF-8

UTF-8 (8-bit UCS/Unicode Transformation Format) is a variable-length character encoding for [Unicode](#). It is able to represent any character in the Unicode standard, but is backwards compatible with [ASCII](#).

The logic engine provides two functions for converting to and from UTF-8 encoded strings:

- [StringToUTF8 Procedure](#)
- [UTF8ToString Procedure](#)

See [UTF-8 Example](#)

8.3.4 UTF-16

UTF-16 (16-bit UCS/Unicode Transformation Format) is a variable-length character encoding for [Unicode](#). The encoding form maps each character to a sequence of 16-bit words.

The logic engine stores all characters as UTF-16. This allows non-ASCII (English) characters to be used anywhere in the logic engine.

8.3.5 UTF-8 Example

We have a string variable called s, and it has been assigned the following text:

```
s := 'cost = €12';
```

The Euro symbol (€) is not an [ASCII](#) character. It is stored as a [Unicode](#) character and has a [value](#) of 8364 (\$20AC [Hexadecimal](#)).

Only ASCII data can be [written](#) to a [file](#). If this string was to be saved to a file, the non-ASCII character would be corrupted. The [UTF-8](#) format consists of a series of bytes, which can be written to a file, so the string needs to be converted to UTF-8 first, using the [StringToUTF8 Procedure](#):

```
s := 'cost = €12';
StringToUTF8(s);
AssignFile(file1, 'test.txt');
ReWrite(file1);
WriteLn(file1, s);
CloseFile(file1);
```

When the Euro character is encoded in UTF-8, it is stored as three characters with values of \$E2, \$82 and \$AC. If you look at the file text.txt you will see the Euro character if the program recognises it as being UTF-8. If not, you will see the characters â , ¯

When reading data from a UTF-8 encoded file, the reverse process is used:

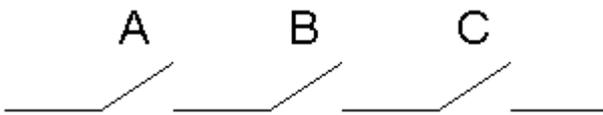
```
AssignFile(file1, 'test.txt');
Reset(file1);
ReadLn(file1, s);
CloseFile(file1);
UTF8ToString(s);
```

A similar process will be needed to reading/writing to [serial ports](#) and [sockets](#).

8.4 Ladder Logic

A common method of describing logic is to use Ladder Logic. This section briefly describes the conversion between a Ladder diagram and the equivalent logic statements.

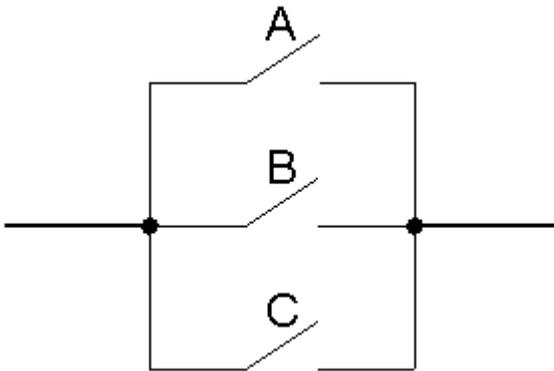
Switches connected in series are equivalent to the logic [AND](#) operation. The Ladder Diagram below :



is equivalent to :

A and B and C

Switches connected in parallel are equivalent to the logic [OR](#) operation. The Ladder Diagram below :

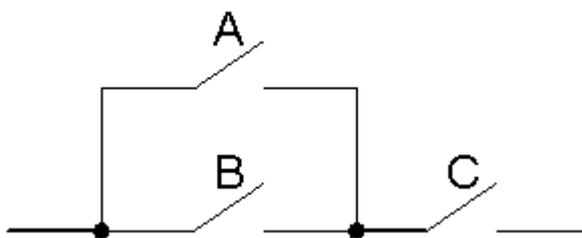


is equivalent to :

A or B or C

Examples

The Ladder Diagram below :

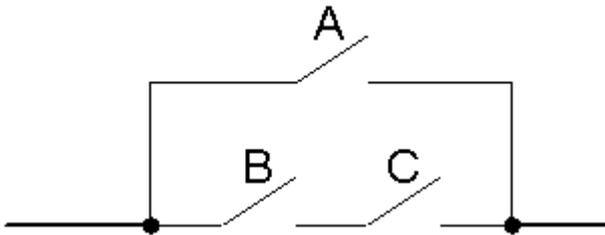


is equivalent to :

(A or B) and C

Note that the OR condition is in brackets because the AND has a higher [operator precedence](#) than the OR.

The Ladder Diagram below :



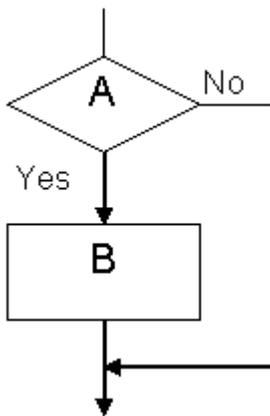
is equivalent to :

A or B and C

8.5 Flow Charts

A Flow Chart can be used to represent a sequence of actions and decisions. For more details on the use of flow charts, refer to a book on software programming.

The simple decision box is equivalent to the logic [If Statement](#). The flowchart below :



is equivalent to :

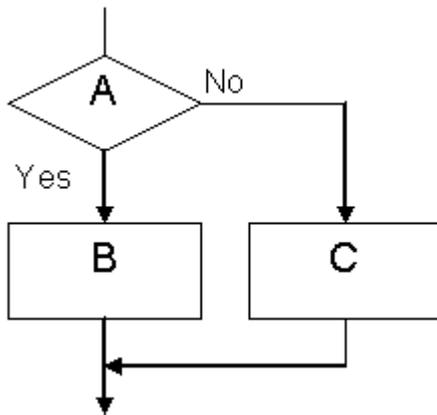
if A then B;

or

once A then B;

depending on whether the statement is to be [edge triggered](#) or not.

The flowchart below :

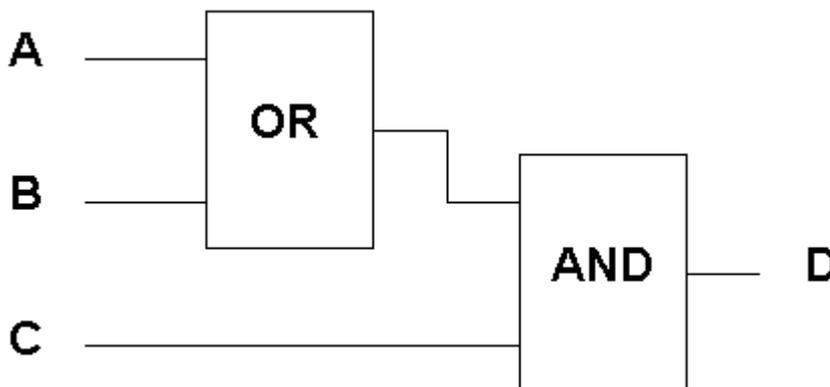


is equivalent to :

```
if A then B else C;
```

8.6 Functional Blocks

Many functional blocks have an equivalent in Logic. For example, the function block below :



is equivalent to :

```
D := (A or B) and C;
```

Note that the OR condition is in brackets because the AND has a higher [operator precedence](#) than the OR.

8.7 Pascal

For more details on the Pascal language see :

[Brian Brown / Peter Henry](#)

[Coronado Enterprises](#)

[Roby](#)

[List of Tutorials](#)

The Logic Engine also has a series of example projects which can be used as a source of information.

The Logic Engine language differs from standard Pascal in the following ways :

- The Logic Engine supports variable length [strings](#)
- Many non-standard built-in functions have been added

- [Modules](#) have been incorporated
- [Tags](#) are supported
- [Once Statement](#)
- [Delay Procedure](#)
- [WaitUntil Procedure](#)
- Alternative [Integer](#) constants

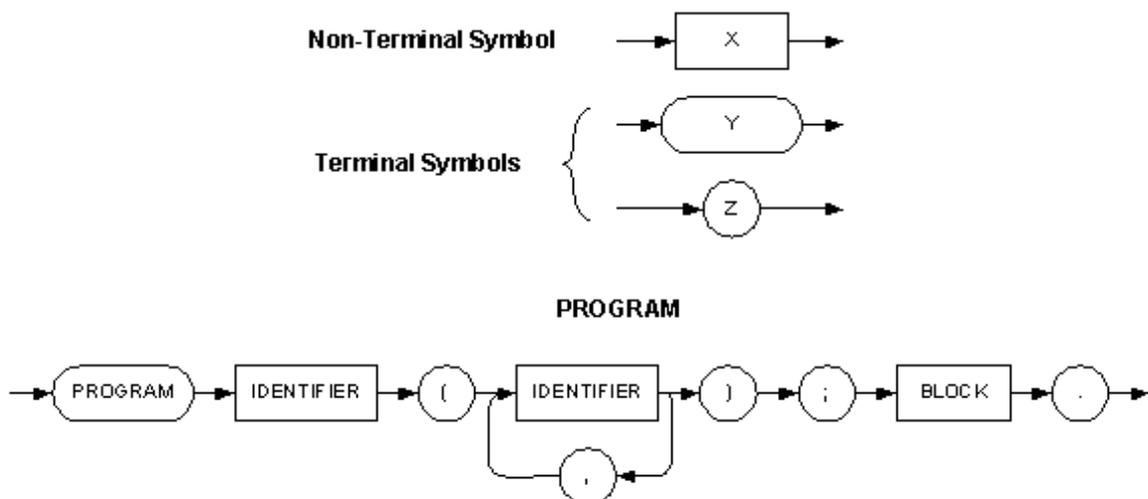
8.7.1 Syntax Diagrams

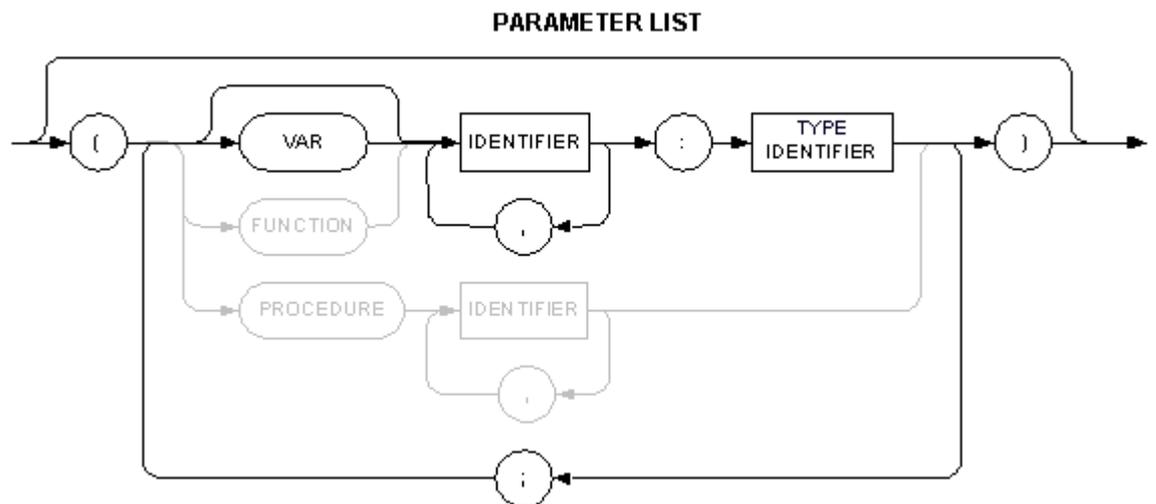
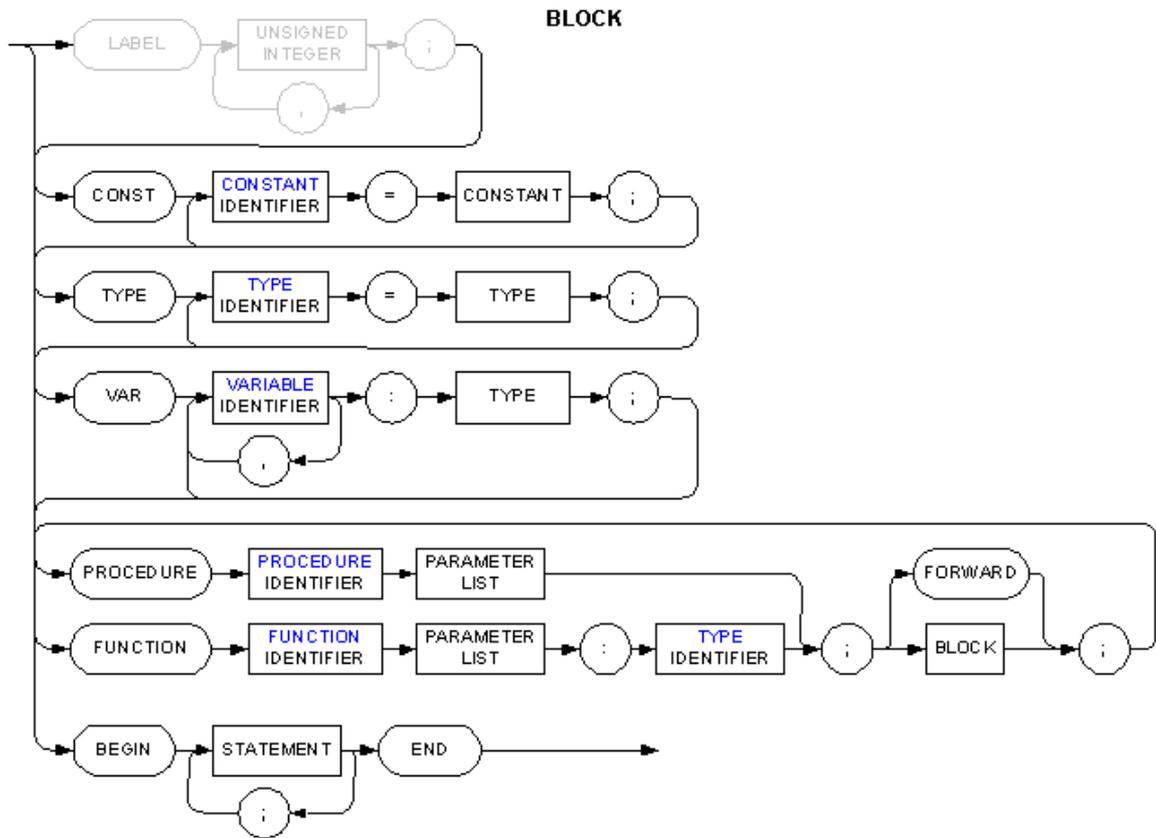
A few rules for Syntax Diagrams

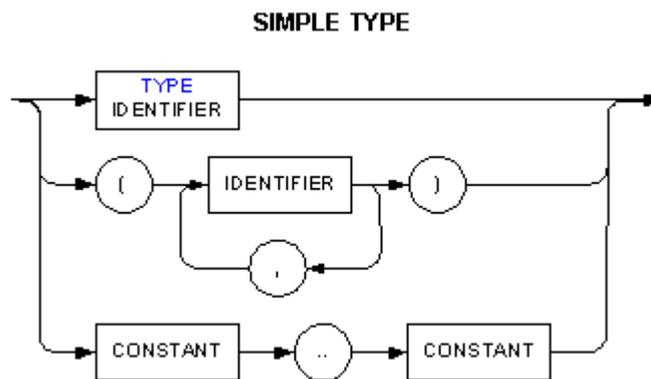
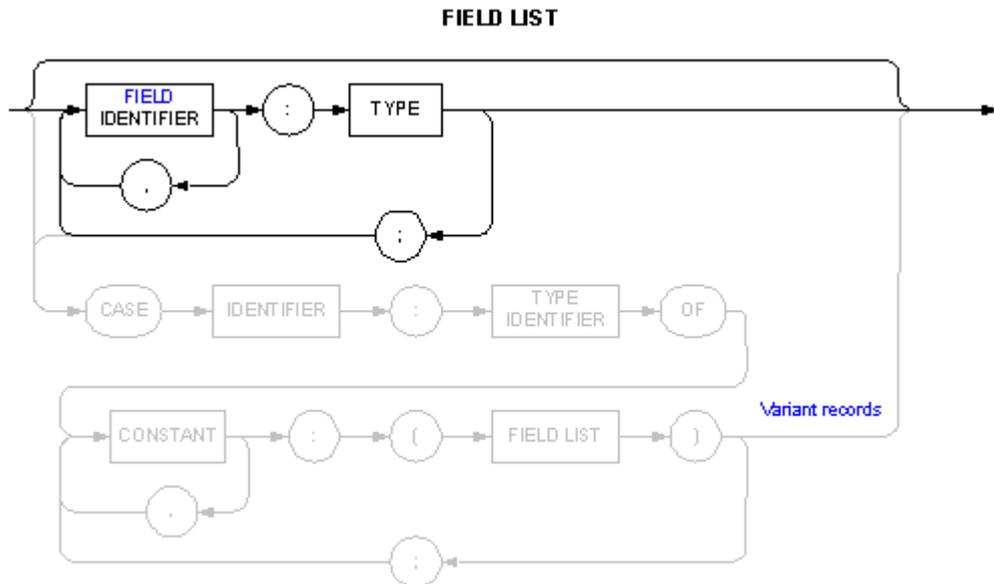
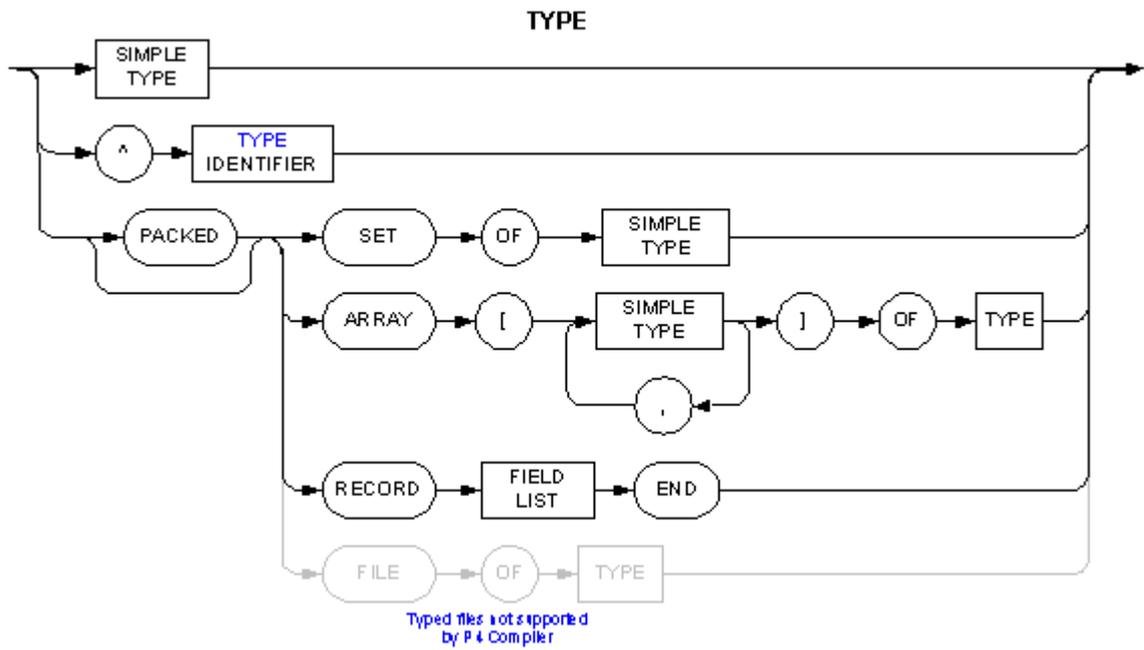
- Non-terminal symbols can be "expanded" and represent another syntax diagram with the name in the box.
- Terminal symbols can not be expanded and the content of the circle / oval is the text which has to be typed.
- The diagrams are traversed in the direction of the arrows.
- Where there is a choice of directions to follow, this means that the various paths are alternatives.

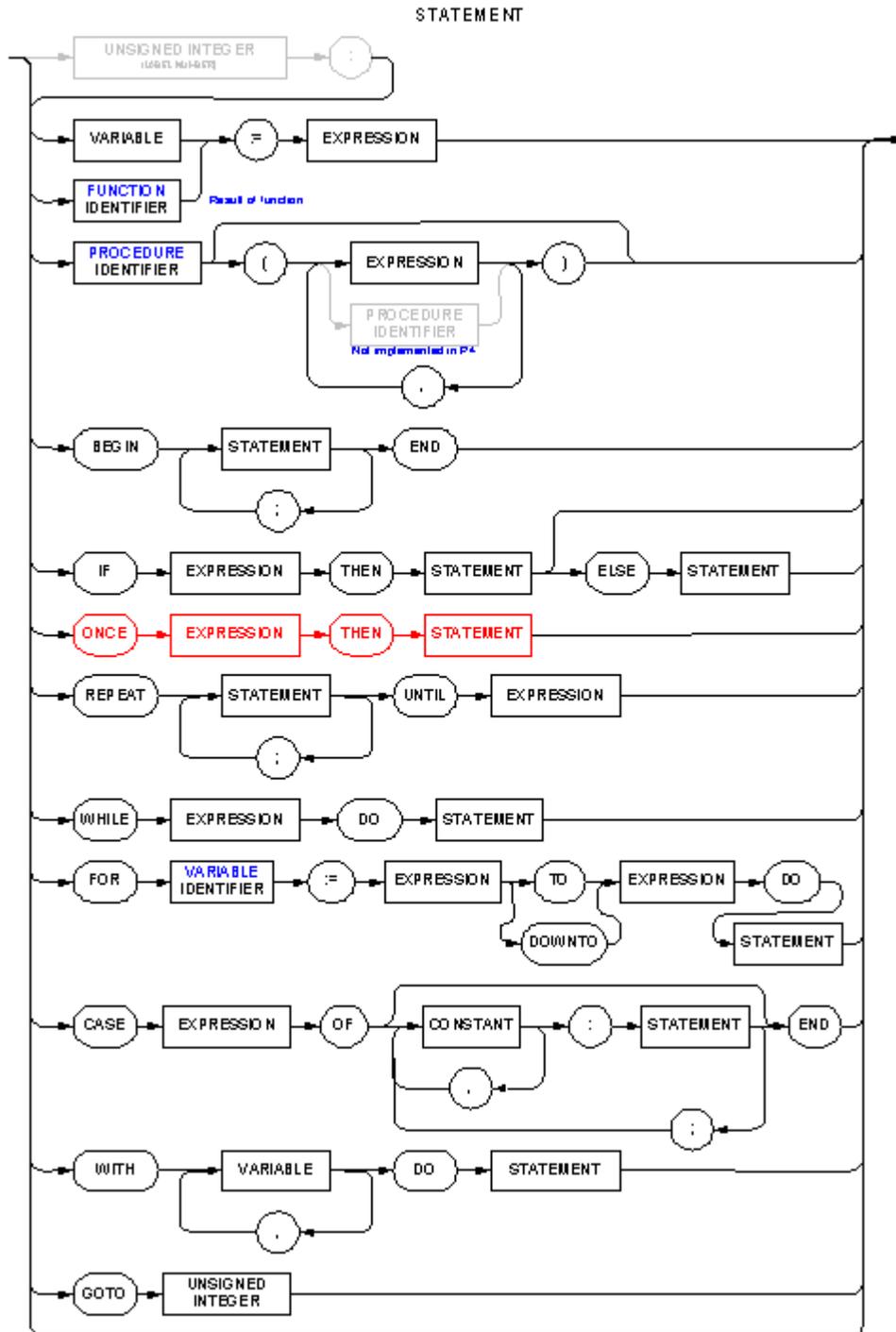
To help to clarify the syntax diagrams, there is some colour coding used :

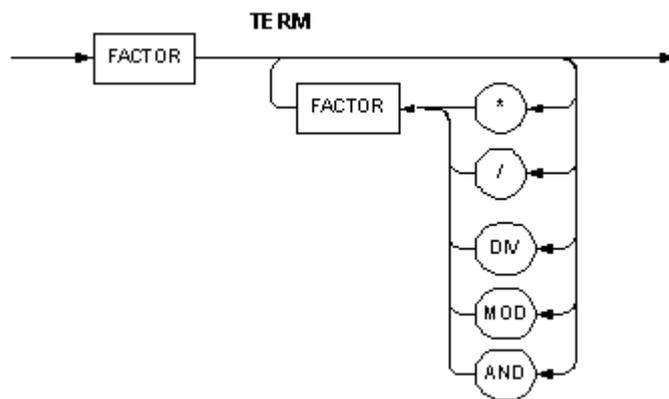
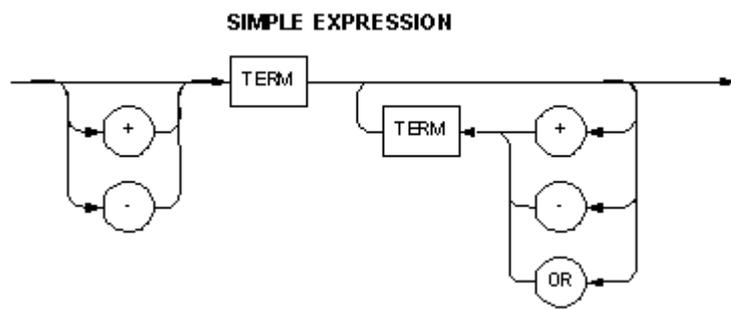
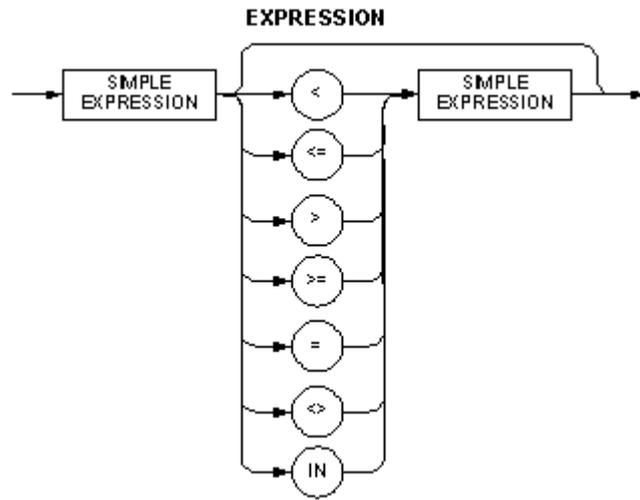
- Black : standard Pascal
- Grey : standard, but uncommon Pascal (or not implemented)
- Red : extensions for the Logic Engine
- Blue : comments

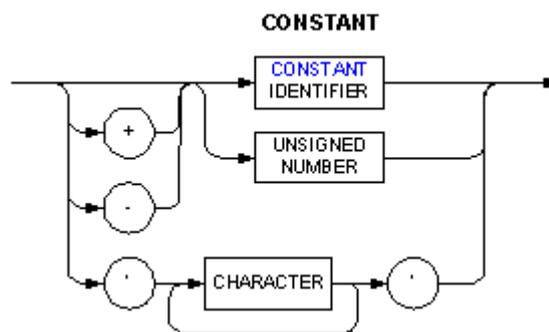
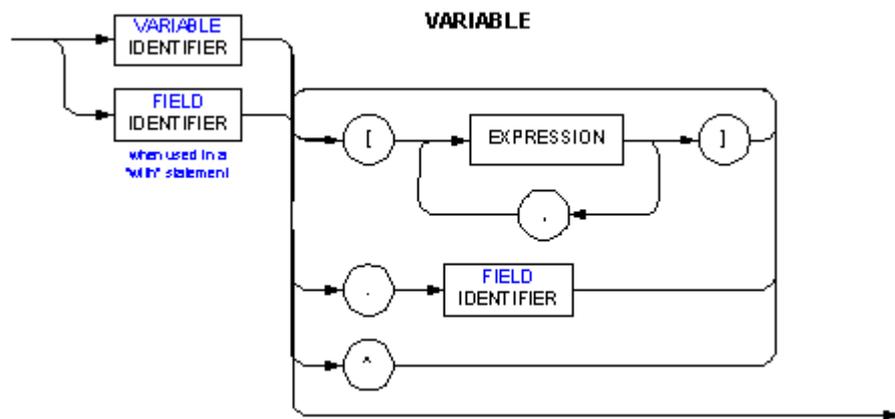
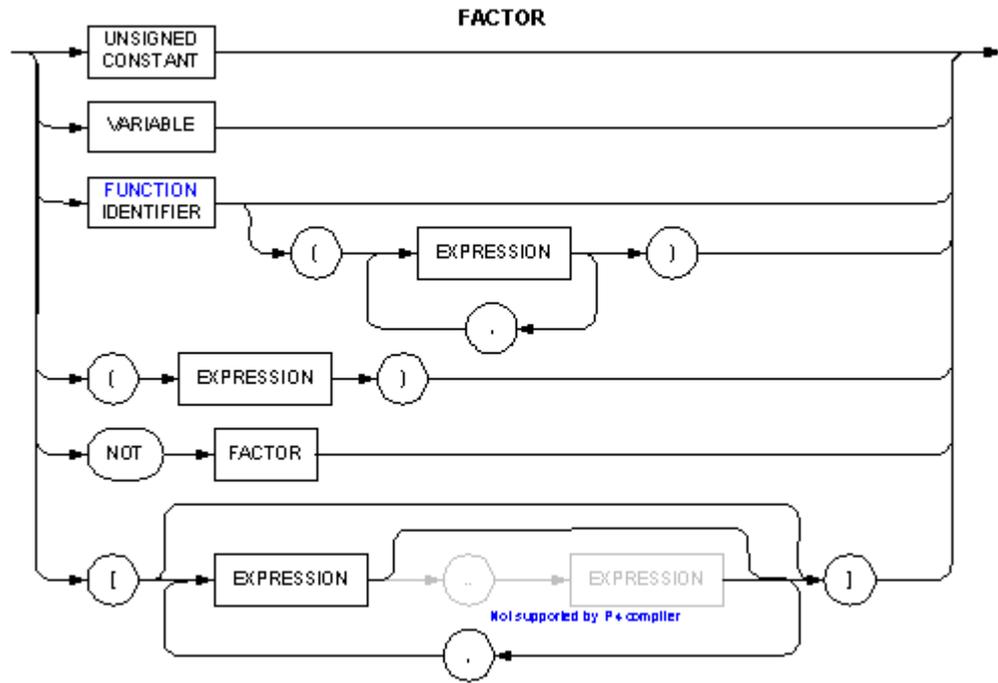




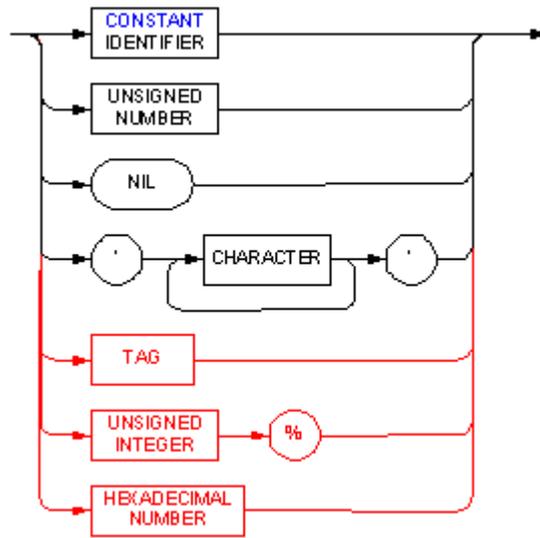




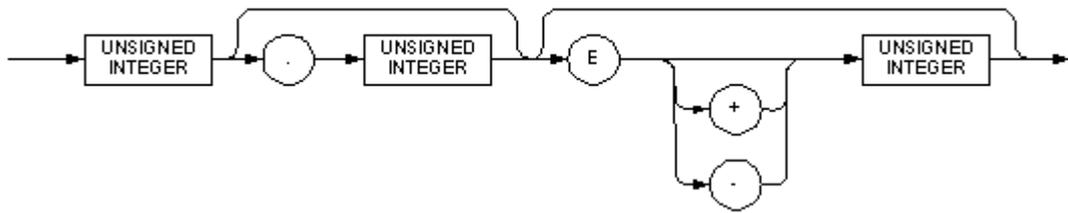




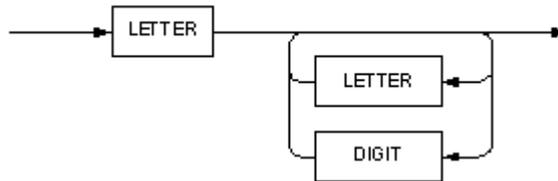
UNSIGNED CONSTANT



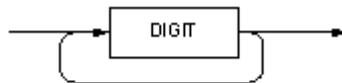
UNSIGNED NUMBER



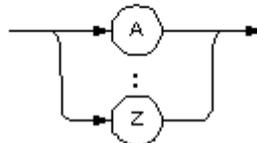
IDENTIFIER

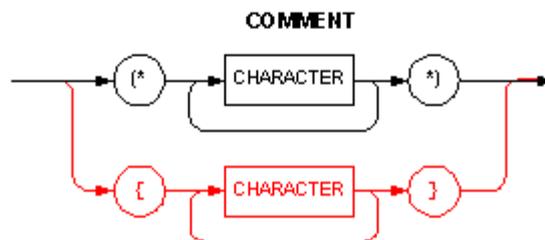
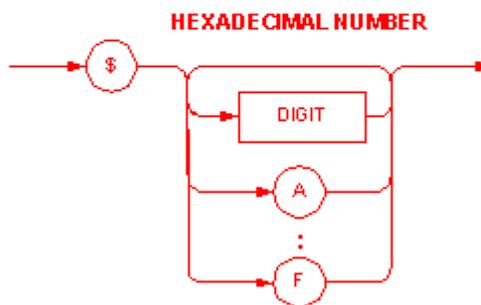
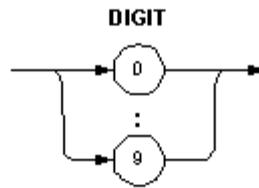


UNSIGNED INTEGER



LETTER





Comments can be placed anywhere in the code

8.8 Tutorial Answers

In the code answers below, the sections that the code segments belong in are shown in comments. For example, a snippet of code starting with { var } belongs in the Global Variables.

There are many ways of solving some of these questions. The only way to be sure that your solution works is to write the code in the Logic Engine and test it. You may want to reduce the duration of any delays for testing purposes.

Tutorial 1

Question 1

name is valid

case is not a valid identifier - it is a reserved word

light level is not a valid identifier - it has a space in it

light_level is valid
group# is not a valid identifier - it has an illegal character (#) in it

Question 2

```
WriteLn('Level = ', Level);
```

Question 3

1200.0
18
255

Question 4

Real
Integer
Boolean
Char
String

Question 5

```
total : integer;  
message : string;  
cost : real;  
error : boolean;
```

Question 6

There is a space in the name of number2 :

```
number1, number 2; integer;
```

There is a semicolon instead of a colon before the word integer :

```
number1, number 2; integer;
```

The = should be a :=

```
number1 = 12;
```

number2 is an integer, and can not be assigned a real value :

```
number2 := 2.3
```

There is a missing semicolon at the end of the statement :

```
number2 := 2.3
```

The quote at the end of the string is missing :

```
WriteLn('number2 =, number2);
```

Question 7

```
WriteLn('Level = ', Level * 100 / 255:6:1, 'W');
```

Question 8

```
Count := Count + 1;
```

Tutorial 2

Question 1

0.125
 1.5
 8
 -2
 7
 2
 0.5
 -1

Question 2

```
Z := X + (Y * Y);
Z := (X + Y) * (X + Y);
Z := (A + B + E) / (D + E);
Z := A + (B / C);
Z := (A + B) / C;
Z := A + (B / (D - C));
```

Question 3

```
Y := 2X + A;
4 := X - Y;
A := 1 / (X + (Y - 2));
-J := K + 1;
S := T / * 3;
Z + 1 := A;
```

should be $2 * X + A$ {missing operator}
 4 is a constant
 missing bracket
 rewrite as $j := -(k + 1);$
 one too many operators
 rewrite as $z := A - 1;$

Question 4

1 false
 2 false
 3 true
 4 true
 5 false
 6 true
 7 false
 8 true
 9 true

Tutorial 3

1. int1 = 10
2. bool1 = true
3. int1 = 5 to 15
4. int1 = 4
5. int1 = 3
6. char1 = 'C'
7. int1 = 5

Tutorial 4

Question 1

```
once (time = Sunset + "0:30:00") and (DayOfWeek <> "Sun") and (DayOfWeek <>
"Sat") then
    SetLightingState("Porch Light", ON);
```

OR

```
once (time = Sunset + "0:30:00") and (DayOfWeek in ["Mon", "Tue", "Wed", "Thu",
"Fri"]) then
    SetLightingState("Porch Light", ON);
```

Question 2

```
once (time = "7:00PM") and (DayOfWeek = "Fri") and (Day <= 7) then
    SetScene("Party");
```

Question 3

The following will not work as required :

```
OffTime := Time + "2:00:00";
```

If the time is after 10PM, then the OffTime variable will be assigned a value of greater than midnight, which is meaningless. The following will ensure that the value is always between 0 and 86399 (11:59:59PM):

```
OffTime := Time + "2:00:00";
if OffTime >= 86400 then
    OffTime := 0;
```

OR

```
OffTime := (Time + "2:00:00") mod 86400;
```

Question 4

To increment Counter every 20 seconds, any of the following will work :

```
if (Second = 0) or (Second = 20) or (Second = 40) then
    Counter := Counter + 1;
```

OR

```
if Second in [0, 20, 40] then
    Counter := Counter + 1;
```

OR

```
if Second mod 20 = 0 then
    Counter := Counter + 1;
```

OR

```
if Time mod 20 = 0 then
    Counter := Counter + 1;
```

To increment Counter every 45 seconds, only the following will work :

```
if Time mod 45 = 0 then
    Counter := Counter + 1;
```

Question 5

```
once GetTriggerLevel("Nudge Up") = 100% then
    NudgeSceneLevel("Living Area", 10%);
```

Tutorial 5

Question 1

The differences between a System IO Variable and a regular Logic [Variable](#) are :

- the user can not directly monitor or control a regular variable.
- with System IO variables, the "get" and "set" functions must be used to use them
- the values of System IO variables are not initialised when the Logic Engine runs
- the values of System IO variables are saved when the Project is saved

Question 2

1. An integer System IO variable value can only be read using the GetIntSystemIO function. The correct code should be :

```
if GetIntSystemIO("Counter") = 10 then ...
```

2. An integer System IO variable can only have its value set using the SetIntSystemIO function. The correct code should be :

```
SetIntSystemIO("Counter", 0);
```

Question 3

```
once GetLightingState("Bathroom Light") then
  TimerStart(1);

if TimerRunning(1) and (TimerTime(1) = "0:30:00") then
begin
  SetLightingState("Bathroom Light", OFF);
  TimerStop(1);
end;
```

Question 4

Write a statement to set the "Switch On Time" System IO variable to the time that the "Spa Pump" Group Address was switched on.

```
once GetLightingState("Spa Pump") then
  SetIntSystemIO("Switch On Time", Time);
```

Tutorial 6

Question 1

```
Copy(String1, String2, 1, 3);
```

Question 2

```
String1 := 'Date =';
DateToString(Date, String2);
Append(String1, String2);
```

Question 3

```
Copy(String2, String1, 6, 3);
x := StringToInt(String2);
```

Question 4

```
if Time = "5:30PM" then
  Execute('HomeTime.wav', '');
```

Question 5

```
Once GetBoolSystemIO("Set Now") then
  SetLightingLevel("Lounge Light", PercentToLevel(GetIntSystemIO("Desired
Level"))));
```

Note that the "Desired Level" has to be converted from a percent to a level before it can be used.

Question 6

Write a statement to set the "All On" Scene at 8:30AM on weekdays (Monday to Friday) which are not a public holiday.

```
once (time = "8:30AM") and (DayOfWeek <> "Sun") and (DayOfWeek <> "Sat") and
not IsSpecialDayType(date, "Public Holiday") then
  SetScene("All On");
```

Tutorial 7

Question 1

One possible solution is :

```
{ var }
  i, total : integer;
...
{ Module 1 }
total := 0;
for i := 1 to 10 do
begin
  total := total + i;
  WriteLn('Triangle Number ', i:3, ' is ', total);
end;
```

Question 2

```
if A > B then
  WriteLn(A)
else
  WriteLn(B);
```

Question 3

B
E
H
I
J

Question 4

A
D

F
H

Question 5

```
if letter < 'A' then
  writeln('Too low')
else
  writeln('Too high');
```

Question 6

2
4
8
16
32
64
128

Question 7

```
case i of
  0 : WriteLn('A');
  1 : WriteLn('B');
  2 : WriteLn('C');
  3 : WriteLn('D');
end;
```

Question 8

```
{ initialisation }
count := 0;
{ ... }
{ main program }
once GetLightingState("Bathroom Light") then
  count := count + 1;
```

Note that a "once" statement must be used. An "if" statement will count the number of [scans](#) executed while the light is on.

Question 9

The following are two simple solutions.

```
once GetLightingState("Lounge Light") = ON then
  SetLightingState("Lounge Lamp", ON);
```

```
once GetLightingState("Lounge Light") = OFF then
  SetLightingState("Lounge Lamp", OFF);
```

OR

```
once GetLightingState("Lounge Light") then
  SetLightingState("Lounge Lamp", ON);
```

```
once not GetLightingState("Lounge Light") then
  SetLightingState("Lounge Lamp", OFF);
```

The following code is nearly the same, but will not allow the lamp to be switched on and off independently, which is probably undesirable :

```
once GetLightingState("Lounge Light") <> GetLightingState("Lounge Lamp") then
  SetLightingState("Lounge Lamp", GetLightingState("Lounge Light"));
```

The following code is NOT acceptable, as it will result in multiple commands being sent to C-Bus :

```
if GetLightingState("Lounge Light") = ON then
  SetLightingState("Lounge Lamp", ON);

if GetLightingState("Lounge Light") = OFF then
  SetLightingState("Lounge Lamp", OFF);
```

OR

```
SetLightingState("Lounge Lamp", GetLightingState("Lounge Light"));
```

Question 10

```
once GetLightingState("Bedroom Switch") then
  if time > "9:00PM" then
    SetLightingLevel("Bedroom Light", 70%, "8s")
  else
    SetLightingState("Bedroom Light", ON);

once not GetLightingState("Bedroom Switch") then
  SetLightingLevel("Bedroom Light", OFF);
```

Question 11

```
once GetLightingState("Room 1 Switch") then
  if GetLightingState("Divider Closed") then
    SetScene("Room 1 On")
  else
    SetScene("All On");

once not GetLightingState("Room 1 Switch") then
  if GetLightingState("Divider Closed") then
    SetScene("Room 1 Off")
  else
    SetScene("All Off");
```

Question 12

```
once GetLightingState("Outside PIR") = ON then
  if (Time > "9:00:00PM") and (Time < "11:59:59PM") then
    begin
      StoreScene("Restore Scene");
      SetLightingState("Room 1", ON);
      delay(2);
      SetLightingState("Room 2", ON);
      delay(2);
      SetLightingState("Room 3", ON);
    end;

once GetLightingState("Outside PIR") = OFF then
  if (Time > "9:00:00PM") and (Time < "11:59:59PM") then
```

```
SetScene("Restore Scene");
```

Question 13

Here is a simple solution using a module called "Lived In", which controls the lived-in look.

```
{ initialisation }
CurrentRoom := 0;
NextRoom := 0;
DisableModule("Lived In");
{ ... }
{ module "Lived In" }
{ pick the next room to switch on }
repeat
  NextRoom := random(4) + 1;
until NextRoom <> CurrentRoom;
{ switch off the light in the current room }
case CurrentRoom of
  0 : ;
  1 : SetLightingState("Room 1", OFF);
  2 : SetLightingState("Room 2", OFF);
  3 : SetLightingState("Room 3", OFF);
  4 : SetLightingState("Room 4", OFF);
end;
{ switch on the light in the next room }
case NextRoom of
  0 : ;
  1 : SetLightingState("Room 1", ON);
  2 : SetLightingState("Room 2", ON);
  3 : SetLightingState("Room 3", ON);
  4 : SetLightingState("Room 4", ON);
end;
CurrentRoom := NextRoom;
delay("0:05:00" + random("0:15:00"));
{ ... }
{ another module }
once GetLightingLevel("away mode") and (time >= sunset + "1:00:00") then
  EnableModule("Lived In");
once not GetLightingLevel("away mode") or (time = "11:00PM") then
  DisableModule("Lived In");
```

Question 14

When the "Night" Scene is set, there is a delay for three hours. During this time, the Module will not be run again. By the time the Module is run again, the time has changed (to 11PM).

Question 15

Line 1 :

- It should be a once statement
- There should be brackets around the individual parts of the condition
- 9PM is not a valid tag, it should be "9PM"

Line 2 :

- Kitchen Light should be within quotes (assuming it is supposed to be a tag)
- There is no semi-colon at the end of the line

The correct code is :

```
once (day = 14) and (month = 7) and (time = "9PM") then
```

```
SetLightingLevel("Kitchen Light", 100%);
```

Tutorial 8

Question 1

```
procedure Multiply( number1, number2 : integer );
var
  Result : integer;
begin
  Result := number1 * number2;
  writeln( Result )
end;
```

Question 2

```
2 0
1 2
```

Question 3

```
function Multiply2( number1, number2 : integer ) : integer;
var
  Result : integer;
begin
  Result := number1 * number2;
  Multiply2 := Result
end;
```

Question 4

```
3
2
3
3
```

Question 5

Errors :

- The begin and end statements are not needed around individual statements
- The assignment operator is := not =
- The WriteLn needs a comma separating the arguments

The correct code is :

```
x := x + 1;
WriteLn('x = ', x);
```

Tutorial 9

Question 1

Using a Delay procedure :

```
once time = "7:00PM" then
begin
  SetLightingState("Porch Light", on);
```

```

    Delay("4:00:00");
    SetLightingState("Porch Light", off);
end;
```

Using a WaitUntil procedure :

```

once time = "7:00PM" then
begin
    SetLightingState("Porch Light", on);
    WaitUntil(time = "11:00PM");
    SetLightingState("Porch Light", off);
end;
```

Using neither :

```

once time = "7:00PM" then
    SetLightingState("Porch Light", on);
once time = "11:00PM" then
    SetLightingState("Porch Light", off);
```

Question 2

```

If Counter >= 100 then
begin
    DisableModule("Module 2");
    WaitUntil(Counter < 50);
    EnableModule("Module 2");
end;
```

Tutorial 10

Question 1

```

{ var }
reply : string;
count : integer;
{ ... }
{ initialisation }
OpenSerial(1, 4, 9600, 8, 1, 0, 0);
{ ... }
{ main program }
count := 0;
WriteSerial(1, 'AT'#13#10);
repeat
    count := count + 1;
    Delay(1);
    ReadSerial(1, reply, #13#10);
until (reply = 'OK') or (count = 10);
if Reply = 'OK' then
    LogMessage('Modem Connected')
else
    LogMessage('Modem Connection Failed');
```

Tutorial 11

Question 1

Numbers[1] is 7
Numbers[2] is 13
Numbers[3] is 12
Numbers[4] is 4
Numbers[5] is 3

Tutorial 12Question 1

```
AssignFile(file1, 'data.txt');  
Reset(file1);  
i := 0;  
while not eof(file1) do  
begin  
  i := i + 1;  
  ReadLn(file1, Data[i]);  
end;  
CloseFile(file1);
```

Index

' 40

-

- 45, 50, 259

"

" 62

#

40

\$

\$ 39, 308

%

% 39

(

(* 37

*

* 45, 50, 259

*) 37

▪

. (record fields) 255

... 5

/

/ 45, 50

// 37

:

:= 42

^

^ 256

{

{ 37

}

} 37

+

+ 45, 50, 259

<

< 46, 50

<= 46, 50

<> 46, 50

=

= 46, 50

>

> 46, 50

>= 46, 50

A

Abs 54

Access Control 107

Air Conditioning 107, 117

Alarm 107, 149

Alignment

horizontal 196

vertical 195

and 9, 47, 49, 50

Append 134

AppendFile 264

Appendix 308
 Applications 62
 Archive 261
 ArcTan 58
 Arithmetic Operators 45
 Arrays 254
 ASCII 309, 311
 AssignFile 262
 Assignment 42
 Audio
 In-built System IO 107
 Auto restart 31
 auto-completion 25

B

Backlight 149
 Basic 8
 Beep 144
 Binary 308
 bits 49
 Blocks 172
 Boolean 40
 Operators 47, 299
 order 50
 precedence 50
 Rules 299
 Simplifying Expressions 299

C

case 161
 when to use 162
 Catch-up 298
 C-Bus
 Conditions 9
 Functions 71
 Get Enable Level 76
 Get Enable State 77
 Get Level 74
 Get Lighting Level 77
 Get Lighting State 77
 Get Ramp Rate 74
 Get State 75
 Get Target Level 75
 Get Trigger Level 79
 Labels 120
 Level 72
 Network state 107

Pulse 82
 Scene 73, 83, 86, 87, 88
 Set Enable Level 84
 Set Enable State 85
 Set Level 83
 Set Lighting Level 85
 Set Lighting State 86
 Set State 84
 Set Trigger Level 88
 State 72
 statements 11
 Tags 73, 90
 Timer 76
 Track Group 89
 Voltage 80, 81
 C-Bus Unit 152
 C-Bus Unit Functions 150
 Char 40, 59, 60
 Char Operators 46
 Characters 309, 310, 311
 Chr 59
 ChrW 60
 ClearScreen 184
 Click
 Example 197
 Get 196
 Get X Coordinate 196
 Get Y Coordinate 197
 ClientSocketConnected 210
 ClientSocketError 211
 CloseClientSocket 211
 CloseFile 265
 CloseSerial 198
 CloseServerSocket 212
 CloseUDPSocket Procedure 218
 Code
 Efficient 300
 code window 25
 Colour 184
 Comments 37
 use of 302
 Compile
 Errors 277
 Warnings 277
 Compiling 30
 Components
 Properties 234, 237, 238, 239, 240, 241, 242, 243
 Compound Statement 11

Conditional Logic 8
 Conditions 8, 9
 ConditionStaysTrue 159
 Const 23, 34
 Constants 38
 in templates 302
 Controlling Modules 298
 Conventions
 Typographic 5
 Conversion
 Char to Integer 60
 Integer to Char 59, 60
 Coordinates 183
 Copy 134
 Cos 58
 Counters 292
 CrossFadeScene 73
 C-Touch 151
 C-Touch Functions 150
 CurrentPage 144

D

Data
 Displaying 307
 Date 63
 Conditions 9
 DateToString 135
 day of month 64
 day of week 64
 day of year 65
 decode 65
 encode 65
 In-built System IO 107
 month 66
 today 63
 year 66
 Date Functions 63
 Date Types 250
 DateToString 135
 Day 64
 Daylight Savings 107
 DayOfWeek 64
 DayOfYear 65
 Debug Data 31
 Debugging 273, 274
 condition testing 274
 error types 273
 intermittent errors 275

 support features 273
 tracking program 275
 Debugging Serial 207
 DecodeDate 65
 DecodeTime 67
 Delay 11, 178
 ConditionStaysTrue 159
 DeleteEMail Procedure 231
 DisableModule 180
 Display 42
 Displaying Data 307
 div 45, 50
 DLT 107, 115, 117
 Labels 120
 DNS 223
 Errors 224
 Example 225
 IP Address 224
 Lookup 223
 Result 224
 DNSLookup Procedure 223
 do 164
 Domain Names 223
 DrawImage 185
 DrawText 185, 307
 DrawTextBlock 186
 DTR 201
 DurationToString 141

E

Efficient Code 300
 Ellipse 187
 Ellipsis 5
 else 154, 161
 E-Mail 227
 Body 228, 230
 Count 228
 Delete 231
 Send 231
 Sender Address 229
 Sender Name 229
 Subject 230
 Enable
 Get State 77
 Set Level 84
 Set State 85
 EnableModule 179
 EncodeDate 65

EncodeTime 67
 Energy 121
 Enumerated Types 250
 EOF 265
 EOLN 265
 Error Messages 277
 Error Reporting 107
 Error Types 273
 Errors 31
 Compilation 277
 Debugging 273, 274
 fixing 301
 Run Time 287
 types 273
 Ethernet 209, 232
 Even 55
 Execute 144
 ExecuteSpecialFunction 149
 ExitModule 179
 Exp 54
 Exponentiation 57

F

False 40
 FAQ 292
 FileExists 266
 Files 261, 311
 AppendFile 264
 AssignFile 262
 CloseFile 265
 End of File 265
 End of Line 265
 Example 266
 FileExists 266
 Reading 263
 Reset 262
 Rewrite 263
 Writing 264
 Flag 40
 Floating Point Numbers 40
 Flow Charts 313
 Flow Control 153
 Fonts 31
 For 164
 Format 136
 Formatting 35
 Forward Declarations 174
 Frequently Asked Questions 292

Function 34
 Functional Blocks 314
 Functions 23, 171
 Standard 53

G

GetAccessLevel 145
 GetBoolIBSystemIO 113
 GetBoolSystemIO 104
 GetCBusApplicationAddress 93
 GetCBusApplicationCount 92
 GetCBusApplicationFromIndex 93
 GetCBusApplicationTag 94
 GetCBusGroupAddress 95
 GetCBusGroupCount 94
 GetCBusGroupFromIndex 95
 GetCBusGroupTag 96
 GetCBusLevel 74
 GetCBusLevelAddress 97
 GetCBusLevelCount 96
 GetCBusLevelFromIndex 97
 GetCBusLevelTag 98
 GetCBusNetworkAddress 91
 GetCBusNetworkCount 90
 GetCBusNetworkFromIndex 91
 GetCBusNetworkTag 92
 GetCBusRampRate 74
 GetCBusState 75
 GetCBusTargetLevel 75
 GetCBusTimer 76
 GetClick 196
 GetClickX 196
 GetClickY 197
 GetCompBooleanProp Function 237
 GetCompIntegerProp Function 237
 GetCompRealProp Function 238
 GetCompStringProp Procedure 238
 GetCompType Function 239
 GetDNSLookupIPAddress Procedure 224
 GetDNSLookupResult Function 224
 GetEMailAddress Procedure 229
 GetEMailBodyLine Procedure 230
 GetEMailBodyLineCount Function 228
 GetEMailCount Function 228
 GetEMailSender Procedure 229
 GetEMailSubject Procedure 230
 GetEnableLevel 76

GetEnableState 77
 GetHTTPData Procedure 225
 GetIntIBSystemIO 114
 GetIntSystemIO 104
 GetIPAddress Procedure 232
 GetLightingLevel 77
 GetLightingState 77
 GetNetworkAdaptorCount Function 232
 GetPageCompCount Function 239
 GetPageIntegerProp Function 233
 GetPingResult Function 223
 GetProfile Function 244
 GetRealIBSystemIO 114
 GetRealSystemIO 104
 GetSceneLevel 78, 83
 GetSceneMaxLevel 79
 GetSceneMinLevel 79
 GetStringIBSystemIO 115
 GetStringSystemIO 105
 GetTransportControlData Procedure 246
 GetTriggerLevel 79
 GetUnitParameter 80
 GetUnitParamStatus 81
 GetUnitStatus 80
 GetZigbeeGroupLightingLevel 270
 Graphics 183
 Brush Colour 190
 Brush Style 190
 Circle 187
 Clear Screen 184
 Click 196, 197
 Colours 184
 coordinates 183
 Ellipse 187
 Font Colour 191
 Font Name 191
 Font Size 192
 Font Style 192
 Image 185
 Line 188
 Move to 188
 Pen Colour 193
 Pen Style 194
 Pen Width 194
 positions 183
 Rectangle 189
 Round Rectangle 189
 Text 185
 Text Block 186

Text Height 195
 Text Position 195
 Text Width 196
 Group
 track 89
 Group Address 72
 Ramp Rate 74
 Ramping 75
 Target Level 75
 Timer 76
 track level 301
 Group Addresses 62, 302
 Group Labels 107

H

Halt 147
 Handshaking lines 201, 202
 HasChanged Function 160
 Heap 257
 Hexadecimal 39, 308
 string 142
 HexStringToInt 142
 Holidays 132, 133
 Hour 68
 HTTP 225, 226, 227
 HVAC 107
 HVAC Application 117

I

Identifiers 35
 if 8, 154
 ConditionStaysTrue 159
 when to use 157, 162
 in 50, 259
 In-Built System IO 107
 Energy 121
 HVAC 117
 Measurement Application 119
 Power 121
 Schedules 127
 Tariff 121
 Index
 parameter 307
 Initialisation 5, 23, 177, 292, 293
 Integer 39
 Internet 209, 218, 222, 223, 225, 232
 Introduction 4

IntToHexString 142
 IP Address 223, 224, 232
 Irrigation 107, 149
 IsCBusUnit 152
 IsCTouch 151
 IsMasterUnit 153
 IsPAC 151
 IsSpecialDayType 132
 IsWiser 152

J

Join Room 301

K

Keyboard Shortcuts 30

L

Labels 107, 115, 117, 120
 Ladder Logic 312
 Language 5, 34
 Labels 120
 LED 150, 151
 Length 136
 Level 72
 Target 75
 Levels 62
 LevelToPercent 145
 Light Level 80, 81
 Lighting
 Get Level 77
 Get State 77
 Level to Percent 145
 Percent to Level 148
 Scene 73, 83, 86, 87, 88
 Set Level 85
 Set State 86
 Track Group 89
 LineTo 188
 Ln 54
 Loads
 Sets 298
 Log 31
 natural 54
 Log in 149
 Log out 149

Logic
 Catch-up 298
 Editor 22
 Flow Charts 313
 Functional Blocks 314
 How Much 303
 In-built System IO 107
 Ladder 312
 quantity 303
 re-use 302
 Rules 299
 Simplifying Expressions 299
 speed 303
 Templates 302
 Using 22
 when to use 292
 Logic Engine 315
 Language 5
 operation 5
 Restart 147
 Stop 147
 Logic Report 30
 Logic Tree 23
 LogMessage 146
 LowerCase 136

M

Make on LAN 221
 Mark 257
 Master Unit 153
 Mathematical Functions 53
 Measurement Application 103
 System IO 119
 Media Transport Control 245
 Memory 29, 31
 Management 257
 Security 297
 Menu Items 22
 Minute 68
 mod 45, 50
 Module
 disabled 180
 enabled 181
 waiting 181
 Module Wizard 26
 Actions 28
 Conditions 27
 Details 27

ModuleDisabled 180
 ModuleEnabled 181
 Modules 5, 13, 23, 176
 controlling 298
 delay 293, 299
 disabled 293
 enabling 298
 execution 293
 Groups 177
 Tags 177
 Wizard 26
 ModuleWaiting 181
 Monitors 107
 Month 66
 MoveTo 188
 Music 245

N

Name space 302
 Names 62
 Network Adaptor 232
 Networks 62
 New 256, 257
 Next 61
 nil 256
 not 47, 49, 50
 NudgeSceneLevel 82

O

Odd 55
 of 161, 259
 Off 40
 On 40
 once 8, 156
 when to use 157
 OpenClientSocket 212
 OpenSerial 199
 OpenServerSocket 212
 OpenUDPSocket Procedure 218
 Operands 45
 Operation 5
 Operator Precedence 50
 Operators 45
 Options 31
 or 9, 47, 49, 50
 Ord 60

Ordinal
 functions 59
 types 59
 OrdW 60
 Other Functions 143
 output window 26

P

PAC 151
 PAC Functions 150
 Page
 current 144
 Properties 233, 234
 show 146
 showing 147
 which is showing 144, 147
 Page Transition effect 149
 Pages 107
 Properties 233
 Selecting 11
 Parameter
 index 307
 Parameters
 Strings 170
 Value 170
 Variable 170
 Pascal 314, 315
 Percent 39
 PercentToLevel 148
 Ping 222
 Example 223
 Result 223
 Send 222
 Pointers 256, 257
 pop-up menu 25
 Pos 137
 Pos2 137
 PostHTTPData Procedure 226
 Power 57, 121
 Power-up 298
 Pred 61
 Previous 61
 Procedure 34
 Procedures 23, 168
 ProfilelsSet Function 244
 Profiles 244, 245
 Program 34
 execute 144

Programs 5
 execution 293
 Protocols 202
 PulseCBusLevel 82

Q

Quick Start 8

R

Ramp Rate 74
 Ramp Rates 62
 Random 55
 Times 295
 Range Checking 31
 Read 42, 263
 ReadClientSocket 213
 ReadHTTPData Procedure 225
 ReadHTTPPostData Procedure 227
 ReadLn 42, 263
 ReadSerial 200
 ReadServerSocket 214
 ReadUDPSocket Procedure 219
 Real 40
 Records 255
 Rectangle 189
 Recursion 174
 Relational Operators 46
 Release 257
 Repeat 45, 163
 Report
 Logic 30
 Reset 262
 Resource Usage 31
 Resources 29
 Restart 147
 Restart Logic 31
 Rewrite 263
 right click 25
 Room Join 89, 301
 Round 56
 RoundRect 189
 RS232 197, 201, 202
 Debugging 207
 Example 202
 RTS 202
 Run Time Errors 287

Running Logic 31
 RunTime function 68

S

Scan 5
 Scene
 Cross-Fade 73
 Get Level 78, 83
 Get Max Level 79
 Get Min Level 79
 Is it set? 78, 83
 Levels 78, 83
 Nudge Levels 82
 Set 86
 Set Level 86
 Set Offset 87
 Store 88
 Scenes 62, 107, 298
 statements 11
 Schedules 107, 127
 Scope 173
 ScreenHeight 183
 ScreenWidth 183
 Second 69
 Security 107, 297
 SendEmail Procedure 231
 SendPing Procedure 222
 SendWOL Procedure 221
 Serial
 Close 198
 COM Ports 207
 Debugging 207
 DTR 201
 Embedded devices 207
 Errors 207
 Example 202
 Open 199
 Read 200
 RTS 202
 Security 297
 Write 201
 Serial Commands 197
 ServerSocketActive 214
 ServerSocketError 215
 ServerSocketHasClient 215
 SetBoolBSystemIO 116
 SetBoolSystemIO 105
 SetBrushColor 190

- SetBrushStyle 190
- SetCBusLevel 83
- SetCBusState 84
- SetCompBooleanProp Procedure 240
- SetCompCBusProp Procedure 240
- SetCompIntegerProp Procedure 241
- SetCompRealProp Procedure 241
- SetCompStringProp Procedure 242
- SetEnableLevel 84
- SetEnableState 85
- SetFontColor 191
- SetFontName 191
- SetFontSize 192
- SetFontStyle 192
- SetIntIBSystemIO 116
- SetIntSystemIO 106
- SetLEDState 150
- SetLength 138
- SetLightingLevel 85
- SetLightingState 86
- SetPageIntegerProp Procedure 234
- SetPenColor 193
- SetPenStyle 194
- SetPenWidth 194
- SetProfile Procedure 245
- SetRealIBSystemIO 117
- SetRealSystemIO 106
- Sets 258
 - Example 260
 - Operations 259
- Sets of Loads 298
- SetScene 86
- SetSceneLevel 86
- SetSceneOffset 87
- SetSerialDTR 201
- SetSerialRTS 202
- SetStringIBSystemIO 117
- SetStringSystemIO 107
- SetTriggerLevel 88
- shl 49, 50
- Shortcuts
 - Keyboard 30
- ShowingPage 147
- ShowingSubPage Function 243
- ShowPage 146
- ShowSubPage Procedure 243
- shr 49, 50
- Sin 58
- Sockets 209
 - Security 297
 - TCP/IP 209
 - UDP 218
- Solar Power 121
- Special Days 132, 133
- Special Function 149
- SpecialDayType 133
- Sqr 56
- Sqrt 56
- Stack 257
- Start time 68
- Start-up 298
- State 72
- Statement Wizard 29
- Statements 11
 - Compound 11
- Stop 147
- StoreScene 88
- String
 - Append 134
 - Copy 134
 - Date to String 135
 - Duration to String 141
 - Format 136
 - Hexadecimal to Integer 142
 - Integer to Hexadecimal 142
 - Length 136
 - Lower Case 136
 - Pos 137
 - Pos2 137
 - Set Length 138
 - String to Integer 138, 139
 - String to Real 140
 - Time to String 140
 - Type 40
 - Upper Case 141
 - UTF-8 143
- String Operators 46
- Strings 40, 309, 311
 - Functions 134
 - Parameters 170
- StringToInt 138
- StringToIntDef 139
- StringToReal 140
- StringToUTF8 143
- Structure 34, 35
- Sub-Page Frame 243
- Sub-Pages 243

Sub-Programs 168
 Sub-Ranges 253
 Succ 61
 Sunrise 69
 Sunset 70
 Syntax Diagrams 315
 Syntax Highlighting 31
 System
 In-built System IO 107
 System IO
 Displaying Data 307
 Editor 103
 Energy 121
 examples 130
 Functions 101
 GetBoolIBSystemIO 113
 GetBoolSystemIO 104
 GetIntIBSystemIO 114
 GetIntSystemIO 104
 GetRealIBSystemIO 114
 GetRealSystemIO 104
 GetStringIBSystemIO 115
 GetStringSystemIO 105
 In-Built 107, 117
 Measurement Application 119
 Power 121
 Schedules 127
 SetBoolIBSystemIO 116
 SetBoolSystemIO 105
 SetIntIBSystemIO 116
 SetIntSystemIO 106
 SetRealIBSystemIO 117
 SetRealSystemIO 106
 SetStringIBSystemIO 117
 SetStringSystemIO 107
 Tags 103, 113
 Tariff 121
 User 102
 using 102
 Variable Editor 103

Target Level 75
 Tariff 121
 TCP/IP 209
 Client Close 211
 Client Connected 210
 Client Error 211
 Client Open 212
 Client Read 213
 Client Write 216
 Example 216
 Server Active 214
 Server Close 212
 Server Error 215
 Server Has Client 215
 Server Open 212
 Server Read 214
 Server Write 216
 Telephony 107
 Temperature 80, 81
 Templates
 Logic 302
 Text 40
 Displaying 307
 TextHeight 195
 TextPos 195
 TextWidth 196
 Then 8, 154, 156
 Time
 Conditions 9
 decode 67
 DurationToString 141
 encode 67
 hour 68
 In-built System IO 107
 minute 68
 now 70
 Random 295
 RunTime 68
 second 69
 sunrise 69
 sunset 70
 time since start 68
 TimeToString 140
 Time Functions 66
 Timer 76
 Timer Functions 99
 TimerRunning 100
 TimerSet 100
 TimerStart 100

T

Tags 62
 C-Bus 73, 90
 C-Bus Application 92, 93, 94
 C-Bus Group 94, 95, 96
 C-Bus Level 96, 97, 98
 C-Bus Network 90, 91, 92
 using constants in place of 302

TimerStop 101
 TimerTime 101
 TimeToString 140
 Today 63
 ToggleLEDState 151
 Tool bar 23
 Track Group Address 301
 TrackGroup 89
 TrackGroup2 89
 Transport Control Application 245
 TransportControlData Procedure 247
 TransportControlDataCount Function 248
 TransportControlDataMLG Function 248
 TransportControlDataStart Function 248
 TransportControlDataType Function 247
 TransportControlDataValid Function 249
 TransportControlFlag Function 249
 Tree
 Logic 23
 Trigger
 Get Level 79
 Set Level 88
 Triggers 297
 Trigonometric Functions 57
 True 40
 Trunc 57
 Tutorial
 1 43
 10 209
 11 261
 12 267
 2 52
 3 62
 4 99
 5 131
 6 148
 7 165
 8 175
 9 182
 Answers 322
 Type 23, 34
 Types 39

U

UDP 218
 Active 220
 Close 218
 Error 220

Example 222
 Open 218
 Read 219
 WOL 221
 Write 219
 UDPSocketActive Function 220
 UDPSocketError Function 220
 Unicode 143, 309, 310, 311
 Unit
 C-Bus 152
 C-Touch 151
 Master 153
 PAC 151
 Parameter 80
 Parameter Status 81
 Status 80
 Wiser 152
 Until 163
 UpperCase 141
 User System IO 102
 Using Logic 22
 UTF-16 309, 311
 UTF-8 143, 309, 311
 UTF8ToString 143

V

Var 23, 34
 Variables 38, 292
 Voltage 80, 81

W

WaitUntil 182
 Warnings
 Compilation 277
 Web 209
 Web Data 225
 While 164
 Wiser Functions 150
 Wiser Unit 152
 Wizard
 Module 26
 Statement 29
 WOL 221
 Write 42, 264
 WriteClientSocket 216
 WriteLn 31, 42, 264
 WriteSerial 201

WriteServerSocket 216

WriteUDPSocket Procedure 219

X

xor 47, 49, 50

Y

Year 66

Z

Zones

HVAC 117