



OOPic Programmer's Guide

Index

Contents:

- [Chapter 1 - Introduction](#)
- [Chapter 2 - Your First OOPic Application](#)
- [Chapter 3 - Your First Virtual Circuit](#)
- [Chapter 4 - Your First Event Driven Program](#)
- [Chapter 5 - Programming Fundamentals](#)
- [Chapter 6 - Introduction to Objects](#)
- [Chapter 7 - Variables & Constants](#)
- [Chapter 8 - Interfacing with Hardware](#)
- [Chapter 9 - Virtual Circuits](#)
- [Chapter 10 - The OOPic System Object](#)
- [Chapter 11 - Mathematical Operations](#)
- [Chapter 12 - Dynamic Data Exchange \(DDE\)](#)

Copyright(c) 1999,2000 by Savage Innovations All rights reserved



OOPic Programmer's Guide

Chapter 1. - Introduction

Contents:

- [Defining OOPic](#)
- [How This Manual is Organized](#)
- [Compatibility with Other Languages](#)
- [Terms to Get You Started](#)
- [Notations](#)
- [Setting Up the OOPic Software Development Kit](#)
- [Sample Applications](#)
- [On Line Help](#)

Defining OOPic

OOPic (Pronounced OO-Pic) is an acronym for: **O**bject-**O**riented **P**rogrammable **I**ntegrate **C**ircuit.

OOPic helps you be more creative by providing you with an Object-oriented language model designed to

interact with the electrical hardware components that you attach to the OOPic. You design your hardware interface in software by creating Objects and setting their properties to define their behavior and interaction with hardware. These Objects also can be interconnected to form a Virtual Circuit. You then utilize this hardware interface and its associated Virtual Circuits by writing a program that controls and responds to hardware events that occur.

This chapter shows you how to set up the OOPic Software Development Kit on your computer and introduces other parts of the documentation

How This Manual is Organized

The chapters of this guide can be grouped into three parts.

Part 1 preliminary steps	Chapters 1-4 describe the preliminary steps to using this product, including Setup and three simple "First applications" to get you familiar with OOPic Basic syntax and it's use with the OOPic.
Part 2 Programming	Chapters 5-8 explain the fundamentals of programming OOPic , including Objects, variables, program control, and hardware I/O
Part 3 Virtual Circuits	Chapter 9-12 cover advanced topics such as Virtual Circuits, The OOPic system Object, Mathematical operations and networking.

Compatibility with Other Languages

The list below describes the compatibility of the OOPic language with other common languages.

BASIC:	The OOPic Basic syntax is 100% compatible with Microsoft's Visual Basic Language. The fundamental syntax comes directly from conventional Basic syntax and will be familiar to even the novice Basic programmer.
C:	The OOPic C syntax, as of this writing, is currently being developed.
Java:	The OOPic Java syntax, as of this writing, is currently being developed.

Terms to Get You Started

The list below defines some of the common terms used in the OOPic manuals to describe parts of the OOPic programming language used in this manual.

Keyword	Any reserved word in the OOPic language. Keywords may be a command, a function, a statement or any other word that OOPic uses for any other purpose. All keyword in OOPic Basic are case insensitive.
Identifier	A word that is defined by the programmer to represent an application specific item. An identifier cannot be spelled the same as any keyword and it must begin with an alphabetic letter
Constant	A predefined keyword that literally represents a value. Constants are used in place of numbers so that code is easier to read.
Variable	An Identifier that represents a value but, unlike a constant, it can be changed by assigning it a value. The value represented by a variable will remain the same until another value is assigned.
Command	A keyword that instructs the OOPic to take a particular action.
Function	A keyword that instructs the OOPic to perform a calculation and return a result. A function is a lot like a variable in that it represents a value, but the value that it represents is calculated at the time the function is executed.

Statement	A "sentence" of commands, functions, identifiers and constants that instruct the OOPic to take a particular action and describes how to do it. A Statement usually begins with a command but can also begin with an identifier.
Operator	Operators are special commands that perform calculations, evaluations and assignments.

If you are new to the Object-oriented and event-driven programming models, you may find some of the terminology a little strange at first. The list below defines some of the common terms used in the OOPic manuals to describe these programming models.

Event	An action recognized by an Object which code can be written to respond to. Events may occur as a result of a hardware condition or software manipulation of an Object.
Event-driven	A term used to describe a programming model. Unlike applications written in a procedural style, an event-driven application consists of code that remains idle until called upon by an Object to response to an event.
Method	A keyword (similar to a function or statement) that is part of the logical unit of an Object and whose operation acts directly upon that Object.
Object	A term used to describe a set of variables and code that acts as one logical unit.
Object-oriented	A term used to describe a programming model were certain variables and code act as a single logical unit.
Property	A variable that is part of the logical unit of an Object.

Virtual Circuits provide a means in which one can emulate electronic circuits that manipulate values. The list below defines some of the common terms used in the OOPic manuals to describe Virtual Circuits.

Default Value	The term used to describe the property of an object that will be used when none was specified by the program.
Flag	A term used to describe a 1 bit property that can be linked to a Flag-Pointer.
Link	A term used to describe a connection point in a Virtual Circuit.
Pointer	A term used to describe a property that is used to point to another property.
Virtual Circuit	A Virtual Circuit is a circuit in an OOPic that appears to be a physical discrete electronic circuit, but is actually the OOPic operating system emulating the functionality of the circuit.

Notations

Items in Brackets: Brackets, [], contain optional items which may be used but are not required. If the item enclosed in brackets is followed by three dots [exp...], it means that any number of expressions may be entered, but none are required.

The following table explains the abbreviations used throughout this manual:

{variable}	Any variable.
{object}	Any Object.
{class}	An Object class.
{label}	Any line label.
{constantlist}	A list of data items.
{subscript}	A numeric value.

{operator}	Arithmetic or logical operator.
{expression}	Arithmetic or logical Expression.

Setting Up the OOPic Software Development Kit

Before You Run Setup

Before you install the OOPic Software Development Kit, make sure your computer meets the minimum requirements, and be certain to read the README file, located at the root directory on our installation disk.

Check the Hardware and System Requirements.

To run the OOPic Software Development Kit, you must have certain hardware and software installed on your computer. The minimum system requirements include:

- Any IBM®-compatible machine running Windows® 95® or later.
- A hard disk with a minimum of 5 megabytes available space.
- A 5.25-inch or 3.5-inch disk drives.
- A mouse or other suitable pointing device.
- For the Command Line Compiler, you must have MS-DOS 3.11 or later.
- An available parallel Port if using the parallel programming cable.

Read the README File

The README file lists any changes to the OOPic documentation since its publication. Check the first section of the file for details and new information.

Setting Up

You install OOPic Software Development Kit on your computer using the Setup program. The Setup program installs the OOPic Software Development Kit itself, the Help system and sample applications.

Important:

You cannot simply copy files from the distribution disks to your hard disk and run the OOPic Software Development Kit. You must use the Setup program, which decompresses and installs the files in the appropriate directories.

When you run the Setup Program, a directory is created for OOPic. All of the required files will be installed within this directory with the exception of the Microsoft® Visual Basic® runtime files.

Follow this procedure to install the OOPic Software Development Kit on your computer.

1. Press the Windows Start Button.
2. Select **R**un from the Windows Start Menu.
3. Type "A:setup" and press return.
4. Follow the directions given in the installation program.

Sample Applications

In addition to the documentation, the OOPic Software Development Kit includes sample applications that you

can load into the OOPic. These applications are excellent learning tools. You can copy any part of them into your own applications, and modify them as necessary. You can find out more about these on our Internet world-wide-web page: <http://www.oopic.com>

On Line Help

You can access Help by choosing the Contents command from the Help menu.

The fastest way to find a particular topic in Help is to use the Search dialog box.

- From the Help menu, choose Search for Help On, or click the Search button from any Help topic window.
- In the Search dialog box, type a word, or select one from the list by scrolling up or down. Press ENTER or choose Show Topics to display a list of topics related to the word you specified.
- Select a topic name, and then press ENTER or choose "GO TO" to view the topic.

OOPic Internet Support Services

- Several services are available on our Internet world-wide-web page.
- You can access the OOPic Internet Support Services at: <http://www.oopic.com>

Copyright(c) 1999,2000 by Savage Innovations All rights reserved



OOPic Programmer's Guide

Chapter 2. - Your first OOPic application.

Contents:

- [What is an OOPic Application](#)
- [Steps to Creating Your First OOPic Application](#)
- [Starting the OOPic Software Development Kit](#)
- [First Application Concept](#)
- [Creating the Hardware Interface.](#)
- [Connecting the Hardware](#)
- [Downloading and Running the Application](#)

What is an OOPic Application

An OOPic application is a computer program written for the purpose of controlling electronic equipment connected to an OOPic microcontroller. The program itself is written and stored on a P.C. style computer where it is "compiled" into OOPic instructions and downloaded into an OOPic microcontroller. Once downloaded, the OOPic runs the program and the P.C. style computer is no longer needed for its continued operation.

In this chapter, instructions are given to create a very small program that can be easily implemented. The intent is to familiarize the new users with the procedures involved in the OOPic development environment.

Steps to Creating Your First OOPic Application

There are 5 steps involved in creating an application for the OOPic

1. Start the OOPic Software development kit.
2. Design a hardware interface.
3. Write code.
4. Connect the hardware.
5. Download and run the application.

Starting the OOPic Software Development Kit

Clicking the "OOPic Language Compiler" selection on the Windows Start Menu will start the OOPic's Software Development Kit. When the OOPic's Software Development Kit opens, it will be in full screen mode with a set of menus, a large program editor area for you to type in your program, and a list of current Objects.

The menu provides a set of operations that you will need while writing your application including a file save and load selection for you to save the applications that you have written.

First OOPic Application Concept

The first OOPic application is a Blinking LED. This device will turn an LED on and off once every second.

The materials required are as follows:

1. 1 OOPic
2. 1 red LED
3. 1 220 ohm resistor.
4. 1 battery connected to the OOPic.
5. 1 OOPic programming cable.

Creating the Hardware Interface.

In an OOPic application, a hardware interface is a series of Objects configured to interact with the physical hardware that is connected to the OOPic.

In our first application, an LED will be repeatedly turned on and off. This will be done by repeatedly outputting a voltage onto the I/O Line that the LED is connected to. A single Digital Input/Output line will be used for this purpose. Starting at the first line of code, type the following statement:

```
Dim LED As New oDio1
```

This instructs the OOPic to create an instance of an **oDio1** (1-bit digital I/O line) Object with the name "LED". The **oDio1** Object is classified as a "Hardware Object". This simply means that the Object will interact with the I/O Lines on the OOPic in some pre-defined way. The function of an **oDio1** Object is either to present a voltage onto one of the OOPic's I/O lines, or to read the voltage that is present on an I/O Line.

Once an instance of an Object has been created, all references to that instance are done by specifying its name, followed by a period, and then the property or method that you want to reference.

To specify which I/O line the LED Object uses for its voltage output, we will set some of its properties. At the bottom of the program, add the following statements:

```

Sub Main( )
  LED.IOLine = 31
  LED.Direction = cvOutput
End Sub

```

The statements [SUB MAIN\(\)](#) and [END SUB](#) define the beginning and the end of a procedure called MAIN. The procedure called "[MAIN](#)" is the procedure that will run first when the OOPic is turned on or reset.

The line "LED.[IOLine](#) = 31" sets the [IOLine](#) property of the Object named LED to 31 and the line "LED.[Direction](#) = [cvOutput](#)" sets the [Direction](#) property to 1 (the value of [cvOutput](#)). After the execution of these two statement, any value written to LED.[Value](#) will now be presented on I/O Line 31. If the [Value](#) property of the LED Object is set to 0, 0-Volts will be presented and if it is set to 1, 5-Volts will be presented.

To cycle the LED.[Value](#) property at a rate of 1 blink per second, we will add code that continuously assigns it to the [OOPic.Hz1](#) value. ([OOPic.Hz1](#) is a value that cycles once every second). At the bottom of the program, just before the line that reads "[END SUB](#)", Insert three lines and add the following statements:

```

Do
  LED.Value = OOPic.Hz1
Loop

```

The first statement sets up a looping construct. The second statement sets the [Value](#) property of the LED Object to [OOPic.Hz1](#), which cycles every second. The third statement finishes the looping construct and causes the program flow to return to the [Do](#) statement.

Assigning the LED Object's [Value](#) property to [OOPic.Hz1](#) when it is currently set to 1, results in the [IOLine](#) to be set to 5 Volts and when the [OOPic.Hz1](#) is currently set to 0, the [IOLine](#) will be set to 0 Volts.

By using the [Do...Loop](#) structure, around the assignment statement, the assignment statement is repeated indefinitely. The result of this is that the LED will turn off and on once every second.

The following code shows the complete code listing written as instructed in the previous paragraphs.

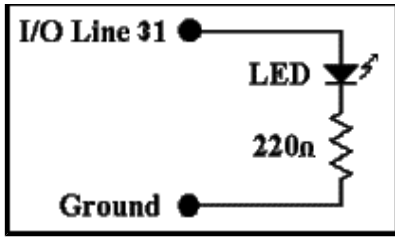
```

Dim LED As New oDiol
Sub Main( )
  LED.IOLine = 31
  LED.Direction = cvOutput
  Do
    LED.Value = OOPic.Hz1
  Loop
End Sub

```

Connecting the Hardware

Connect the LED together with the resistor to I/O Line 31 and Ground as shown.



Both I/O Line 31 and Ground can be found on the OOPic's 40-Pin I/O connector. A reference to the OOPic's 40-Pin I/O connector can be found at: <http://www.oopic.com/con40.htm>


Downloading and Running the Application

[Pressing the F5](#) key will compile the program and then download the "oex" file to the OOPic. If any errors are encountered, they will be reported and the compilation will stop. If no errors were encountered, an "oex" (OOPic Executable) file is generated. If you have not saved the program or you have changed it since the last time you have saved it, the compiler will ask you to save it. If this is the first time the program is going to be saved, the compiler will ask you for a program name. In this case, call the program FirstApp. Type "FirstApp" in the "File name" area of the "Save As" dialog box and hit Enter.

In order for the executable to be downloaded to the OOPic, the programming cable must be connected to the PC and [properly configured](#). It also needs to be connected to the OOPic's programming connector and an adequate power supply must be attached to the OOPic. See the "[Programming Cable](#)" section for more information.

Copyright(c) 1999,2000 by Savage Innovations All rights reserved

•

	<h1>OOPic Programmer's Guide</h1>
<h2>Chapter 3. - Your First Virtual Circuit.</h2>	
<p>Contents:</p> <ul style="list-style-type: none"> • What is a Virtual Circuit • Steps to Creating Your First Virtual Circuit • Creating an OOPic Application • Identify Functions Applicable to Virtual Circuits • Work out the Virtual Circuit Schematic • Add and Link the Virtual Circuit to the Application • Downloading and Running the Virtual Circuit Application 	

What is a Virtual Circuit

A Virtual Circuit is a function in an OOPic that appears to be a physical discrete electronic circuit, but is actually the OOPic operating system emulating the functionality of the circuit.

Virtual Circuits are created by pragmatically linking together a set of OOPic Objects in the same methodology

as one would physically link together a set of electronic components. Once created, each individual part of the virtual circuit can be manipulated or evaluated by the program providing total computerized control of the entire circuit.

Virtual Circuits are used to perform functions that provide continuous processes. A real-world example of a continuous process function would be the electrical circuit between a flashlight's switch and its light bulb. The function of the flashlight's switch is to provide continuous control over its light bulb. In an OOPic application, things that need to be continuously updated or monitored, can be done by using a Virtual Circuit.

Steps to Creating Your First Virtual Circuit

There are 5 steps involved in creating an application for the OOPic

1. Create an OOPic application
2. Identify functions applicable to Virtual Circuits
3. Work out the Virtual Circuits schematic
4. Add and link the Virtual Circuit to the application

Creating an OOPic Application

In Chapter 2, an OOPic application was created that blinks an LED by repeatedly assigning the [OOPic.Hz1](#) property to the [Value](#) property of an [oDio1](#) Object. The [OOPic.Hz1](#) property changes from 0 to 1 and then back to 0 once every second, while the [oDio1](#) Object was configured to apply a voltage to the I/O Line that an LED was connected to resulting in an application that blinked the LED..

In this chapter, instructions are given on how to modify the application in chapter 2 to use a Virtual Circuit to do some of the work.

Identify Functions Applicable to Virtual Circuits

In the OOPic application in Chapter 2, a process that continuously assigns the [OOPic.Hz1](#) property to an [oDio1](#) Object's [Value](#) property takes up the program's time. The function of this process is similar to that of the electrical circuit between a light switch and a light.

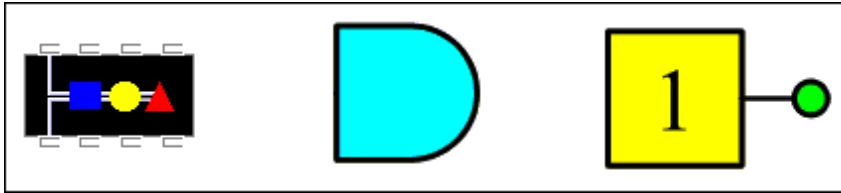
One of the OOPic's Processing Objects, the [oGate](#) Object, is capable of providing that same function that the OOPic application is now doing with code, thus modifying the program to use a Virtual Circuit with this Object will allow the application's program to go do other things.

Work out the Virtual Circuit Schematic

When creating a Virtual Circuit, a schematic is the best way to represent its functionality. Unlike a program, which uses a flowchart to represent the program's flow, a circuit Virtual Circuit uses a schematic to help clarify the function that the Virtual Circuit performs.

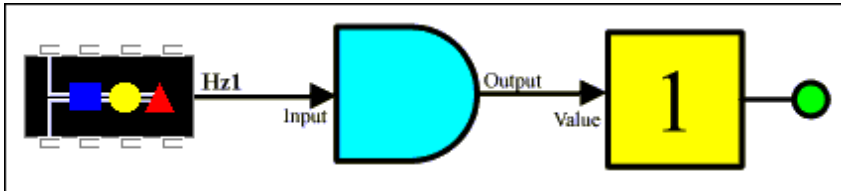
For our First Virtual Circuit, an [oGate](#) Object will be used to update an [oDio1](#) Object's [Value](#) property to the value of the [OOPic.Hz1](#) property.

Begin the schematic by drawing the Icon for the [OOPic](#), Next draw the icon for an [oGate](#) Object, and lastly, draw the Icon for the [oDio1](#) Object.



Next, draw arrows between the Object's properties to represent the data flow between the Objects.

First draw an arrow from the [OOPic](#) Object to the [oGate](#) Object. Label this arrow "Hz1" on the [OOPic](#) Object and "Input" on the [oGate](#) Object. Next draw an arrow from the [oGate](#) Object to the [oDio1](#) Object. Label this arrow "Output" on the [oGate](#) Object and "Value" on the [oDio1](#) Object.



Add and Link the Virtual Circuit to the Application

Adding the Virtual Circuit to the OOPic application is done progmatically with lines of code.

Because this application is a modification of the "First OOPic Application" found in Chapter 2, the final code listing for the OOPic application in Chapter 2, as shown below, will need to be modified.

```
Dim LED As New oDio1
Sub Main( )
    LED.IOLine = 31
    LED.Direction = cvOutput
    Do
        LED.Value = OOPic.Hz1
    Loop
End Sub
```

As previously detailed, the Virtual Circuit will use an [oGate](#) Object to provide the same function that the OOPic application is now doing with code. Starting at the first line of code, insert a blank line and type the following statement:

```
Dim WIRE As New oGate
```

This instructs the OOPic to create an instance of an [oGate](#) Object with the name "WIRE". The [oGate](#) Object is classified as a "Processing Object". This simply means that the Object will manipulate the values of other Objects in some pre-defined way. The function of an [oGate](#) Object is to read a property from one or more Objects, perform a specified logic function, and store the resulting value of that function into yet another Object's property

Next, the application's [Do...Loop](#) structure needs to be removed because its function is going to be replaced by the Virtual Circuit. Beginning with the Do statement, remove the following 3 lines of code.

```

Do
  LED.Value = OOPic.Hz1
Loop

```

Replace them with the following 3 lines of code.

```

WIRE.Input1.Link(OOPic.Hz1)
WIRE.Output.Link(LED.Value)
WIRE.Operate = cvTrue

```

The first line, "WIRE . [Input1](#) . [Link](#) ([OOPic](#) . [Hz1](#))", instructs the "WIRE" Object to link its [Input1](#) property to the [OOPic.Hz1](#) property. The second line, "WIRE . [Output](#) . [Link](#) (LED . [Value](#))", instructs the "WIRE" Object to link its [Output](#) property to the [Value](#) property of the "LED" Object. And the third line, "WIRE . [Operate](#) = [cvTrue](#)", sets the "WIRE" Object's [Operate](#) property to 1 (the value of [cvTrue](#)) which instructs it to start operating.

After the program executes these 3 lines of code, the Virtual Circuit begins to operate in the following manner;

1. The value of the [OOPic.Hz1](#) property will be loaded by the [oGate](#) Object.
2. The [oGate](#) will then use that value in a logical OR operation with its other inputs. In this application, only one input was used, so the result of the logical OR function will always equal the input.
3. The LED. [Value](#) is set to the result of the logical OR function.

This procedure operates in the background and will continue to operate until the WIRE. [Operate](#) property is set back to a value of 0 ([cvOff](#)).

The following code shows the complete code listing written as instructed in the previous paragraphs.

```

Dim WIRE As New oGate
Dim LED As New oDiol
Sub Main()
  LED.IOLine = 31
  LED.Direction = cvOutput
  WIRE.Input1.Link(OOPic.Hz1)
  WIRE.Output.Link(LED.Value)
  WIRE.Operate = cvTrue
End Sub

```

Downloading and Running the Virtual Circuit Application

Downloading and running the Virtual Circuit Application is no different than download and running any other OOPic application.

[Pressing the F5](#) key will compile the program and then download the "oex" file to the OOPic. If any errors are encountered, they will be reported and the compilation will stop. If no errors were encountered, an "oex" (OOPic Executable) file is generated. If you have not saved the program or you have changed it since the last time you have saved it, the compiler will ask you to save it. If this is the first time the program is going to be saved, the compiler will ask you for a program name. In this case, let's call the program "FirstVC". Type "FirstVC" in the "File name" area of the "Save As" dialog box and hit Enter.

In order for the executable to be downloaded to the OOPic, the programming cable must be connected to the PC and [properly configured](#). It also needs to be connected to the OOPic's programming connector and an adequate power supply must be attached to the OOPic. See the "[Programming Cable](#)" section for more information.
Copyright(c) 1999,2000 by Savage Innovations All rights reserved



OOPic Programmer's Guide

Chapter 4. - Your first Event Driven Program.

Contents:

- [What is an Event Driven Program](#)
- [Steps to Creating Your First Event Driven Program](#)
- [Creating an OOPic Application](#)
- [Identify Functions Applicable to Events](#)
- [Add the Event Object to the Application](#)
- [Create the Event Object's Subprocedure](#)
- [Downloading and Running the Virtual Circuit Application](#)

What is an Event Driven Program

An event is defined as any change in state that is recognized by an Object. An Event Driven Program is a program where any change in state can cause a subprocedure to be executed even when the program flow was not expecting to do so. Therefore, the order in which your code executes depends on which events occur.

In Event Driven Programs, the OOPic's [oEvent](#) Object is used to execute code in response to an event.

Steps to Creating Your First Event Driven Program

There are 4 steps involved in creating an application for the OOPic

1. Create an OOPic application
2. Identify functions applicable to events
3. Add the Event Object to the application
4. Create the Event Object's Subprocedure.

Creating an OOPic Application

In Chapter 3, an OOPic application was created that blinks an LED by using an [oGate](#) Object to assign the [OOPic.Hz1](#) property to the [Value](#) property of an [oDio1](#) Object.

In this chapter, instructions are given on how to modify the application in chapter 3 to be Event Driven.

Identify Functions Applicable to Events

In the OOPic application in Chapter 3, a Virtual Circuit continuously assigns the **OOPic.Hz1** property to an **oDio1** Object's **Value** property. The **OOPic.Hz1** property changes state from 0 to 1 and then back to 0 once every second which results in the LED blinking once every second.

The change in state of the **OOPic.Hz1** property can be recognized as an event by the program. Modifying the program to be Event Driven will allow the application's program to call a subprocedure which in turn can do more complex processing.

Add the Event Object to the Application

Adding the Event Driven routines to the OOPic application is done progmatically with lines of code.

Because this application is a modification of the "First Virtual Circuit" application, found in Chapter 3, the final code listing for the OOPic application in Chapter 3, as shown below, will need to be modified.

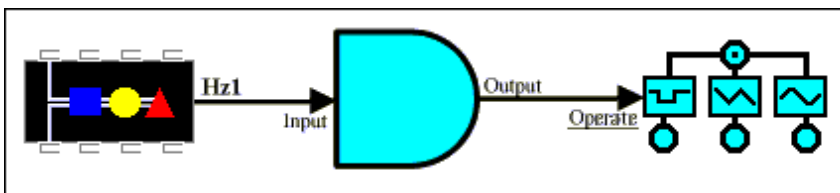
```
Dim WIRE As New oGate
Dim LED As New oDio1
Sub Main( )
    LED.IOLine = 31
    LED.Direction = cvOutput
    WIRE.Input1.Link(OOPic.Hz1)
    WIRE.Output.Link(LED.Value)
    WIRE.Operate = cvTrue
End Sub
```

As previously detailed, this application will use an **oEvent** Object to provide more complex processing. Starting at the first line of code, insert a blank line and type the following statement:

```
Dim BLINK As New oEvent
```

This instructs the OOPic to create an instance of an **oEvent** Object with the name "BLINK". The function of an **oEvent** Object is to call a subprocedure when its **Operate** property is set to 1.

Next, the Virtual Circuit needs to be changed. In the Chapter 3 application, the "WIRE" Object's output was linked to the **oDio1** Object's **Value** property. This needs to be changed so that it is linked to the **oEvent** Object's **Operate** property.



Locate the line that reads "WIRE . **Output** . **Link** (LED . **Value**)", and change it to read as show:

```
WIRE . Output . Link ( BLINK . Operate )
```

This instructs the "WIRE" Object to link its [Output](#) property to the [Operate](#) property of the "BLINK" Object.

After the program executes this line of code, the Virtual Circuit begins to operate in the following manner; and will continue to operate until WIRE.[Operate](#) is set back to a value of 0;

1. The value of the [OOPic.Hz1](#) property will be loaded by the [oGate](#) Object.
2. The [oGate](#) will then use that value in a logical OR operation with its other inputs. In this application, only one input was used, so the result of the logical OR function will always equal the input.
3. The BLINK.[Operate](#) property is set to the result of the logical OR function.
4. Each time the BLINK.[Operate](#) property changes from 0 to 1, the "BLINK" Object's Subprocedure is executed.

The following code shows the complete code listing written as instructed in the previous paragraphs.

```
Dim BLINK As New oEvent
Dim WIRE As New oGate
Dim LED As New oDio1
Sub Main()
    LED.IOLine = 31
    LED.Direction = cvOutput
    WIRE.Input1.Link(OOPic.Hz1)
    WIRE.Output.Link(BLINK.Operate)
    WIRE.Operate = cvTrue
End Sub
```

Create the Event Object's Subprocedure

Each [oEvent](#) Object has an associated subprocedure that is executed when the [oEvent](#) Object's [Operate](#) property is set to 1. The name of this routine is the name of the [oEvent](#) Object followed by "_CODE".

At the bottom of the program add the following lines of code:

```
Sub BLINK_CODE()
    LED.Value = 1
    LED.Value = 0
    LED.Value = 1
    LED.Value = 0
    LED.Value = 1
    LED.Value = 0
End Sub
```

The following code shows the complete code listing written as instructed in the previous paragraphs.

```
Dim BLINK As New oEvent
Dim WIRE As New oGate
Dim LED As New oDio1
Sub Main()
    LED.IOLine = 31
    LED.Direction = cvOutput
```

```
WIRE.Input1.Link(OOPic.Hz1)
WIRE.Output.Link(BLINK.Operate)
WIRE.Operate = cvTrue
End Sub
Sub BLINK\_CODE( )
  LED.Value = 1
  LED.Value = 0
  LED.Value = 1
  LED.Value = 0
  LED.Value = 1
  LED.Value = 0
End Sub
```

Downloading and Running the Virtual Circuit Application

Downloading and running the Virtual Circuit Application is no different than download and running any other OOPic application.

[Pressing the F5](#) key will compile the program and then download the "oex" file to the OOPic. If any errors are encountered, they will be reported and the compilation will stop. If no errors were encountered, an "oex" (OOPic Executable) file is generated. If you have not saved the program or you have changed it since the last time you have saved it, the compiler will ask you to save it. If this is the first time the program is going to be saved, the compiler will ask you for a program name. In this case, let's call the program "FirstEV". Type "FirstEV" in the "File name" area of the "Save As" dialog box and hit Enter.

In order for the executable to be downloaded to the OOPic, the programming cable must be connected to the PC and [properly configured](#). It also needs to be connected to the OOPic's programming connector and an adequate power supply must be attached to the OOPic. See the "[Programming Cable](#)" section for more information.

Copyright(c) 1999,2000 by Savage Innovations All rights reserved

-



OOPic Programmer's Guide

Chapter 5. - Programming Fundamentals.

Contents:

- [OOP](#)
- [Structure of an OOPic Application](#)
- [How an Event-Driven Application Works](#)
- [Objects](#)
- [Variables](#)
- [Subprocedures](#)
- [Functions](#)
- [Control Structures](#)
- [Adding Comments to Your Code](#)

OOP

OOPic Basic syntax is 100% Microsoft Visual Basic compatible. As with Visual Basic, you program in OOPic using an OOP (Object-oriented programming) model.

Object-oriented programming (OOP) is a method of programming that seeks to mimic the way we perceive things in the real world. OOP exploits our natural tendencies to generalize and classify and allows us to group logically related portions of an application. In most OOP languages, the OOP language extensions allow you to create extremely complex Object models that can closely mimic real life Objects. With OOPic however, the greatest advantage of using Object-oriented techniques is the ease at which you can interface with the physical hardware components present within the OOPic itself.

The software industry defines several fundamental principles that qualify a language as Object-oriented. These principles include encapsulation, inheritance, and polymorphism. Although the OOPic manuals do not refer to these principles often, it is useful to understand them.

Encapsulation simply means to group related items together. Rather than treating a number of logically related variables and functions dedicated to manipulating those variables as unrelated pieces of code, encapsulation collects them together in groups called classes. Other parts of the program can then use the variables and functions as a single Object. Once encapsulated, variables are referred to as properties of the Object and the functions are referred to as methods of the Object. Through encapsulation, much of an Object's complexity can be kept internal to the Object. And only the few select methods and properties that need to be available to other parts of a program are exposed. These exposed methods and properties, which can be accessed, are called the Object's interface. The term Data Abstraction refers to the way an Object's interface doesn't reveal the way the Object stores its data or performs its methods. The result of data abstraction is a simple Object interface that interacts with functionality that is considerably more complex.

Polymorphism is a term borrowed from the Greek language. It means literally "Having many shapes". In OOP this is the effect of being able to invoke an Object's method without knowing the Object's exact type and having the method perform a different action depending on that Object's type.

Inheritance allows you to base one Object type on another and build new derived Objects. Since OOPic's Objects are hardware based and the routines are written in optimized assembly language and stored in PROM, there is no way too physical create a new Object type to inherit the class of another. However, by creating Virtual Circuits, inheritance is accomplished.

Structure of an OOPic Application

OOPic applications can contain several different components.

The Main procedure. This is the main part of your program. It contains the primary program flow.

Sub-procedures. These provide your program with sections of code that can be executed from several different points in you program.

Objects. Objects can be thought of as self-contained multitasking code that runs in the background with its own set of variables. They are the way in which all OOPic programs interact with the hardware.

Events. Events are actions or situations recognized by an Object that cause subprocedures to get executed in response to the change in an Objects state. For more information on

Virtual Circuits. Virtual Circuits are emulated electronic circuits that run in the background. They can connect to, and manipulate any variable's values.

How an Event-Driven Application Works

An event is an action recognized by an Object. Event-driven applications execute programmer written code in response to an event. The OOPic's [oEvent](#) Object is the only Object that has the capability to execute code in response to an event action. If any instance of an event Object receives a signal to activate its event procedure, The OOPic application invokes the subprocedure with then name of the oEvent Object instance plus the phrase "_CODE".

Although the [oEvent](#) Object is the only Object in the OOPic language that has event capabilities, other Objects can still indirectly trigger events. Other Objects can be linked to the [oEvent](#) Object thereby giving any Object event capabilities. When you want your code to respond to an event, you first create an instance of an [oEvent](#) Object with the [Dim](#) command, write code for that event and point the [oEvent](#) Object to that code, and lastly "connect" the [oEvent](#) Object to the Object that is triggering the event.

The following example shows how a voltage change on one of the OOPic's I/O Lines can call an event using an [oDio1](#) Object connected to an event Object to invoke the basic syntax code subprocedure "MYEVENT".

```
Dim MyEvent As New oEvent
Dim ThingA As New oDio1
Dim Wire as New oFanout
Sub Main()
    ThingA.IOLine = 1
    ThingA.Direction = cvInput
    Wire.Input.Link(ThingA)
    Wire.Output1.Link(MyEvent.Operate)
END SUB

Sub MyEvent_Code
    event code goes here
End Sub
```

After executing these lines within Sub Main, the applications program flow is idle and you could add code that is free to go and do other things. A "Link" was made when the [Input](#) of the [oFanout](#) Object was linked to the

oDio1 Object and the **Output** of the **oFanout** Object was linked to the **Operate** property of the **oEvent** Object. Now, if the voltage on the I/O line specified by the **oDio1** Object's property changes from 0 to +5 the **Value** property of the Object named "ThingA" will change from 0 to 1, and thusly, The subprocedure named "MyEvent_Code" will be executed. When the **End Sub** statement is encountered at the end of the subprocedure's code, the program will continue where it left off previously at the time of the event.

Event-Driven vs. Traditional Programming

In a traditional or "procedural" application, the application itself rather than an event controls the portions of code that execute. Execution starts with the first line of the program and follows a defined path through the application, branching and calling subprocedures as needed.

In comparison with the example above, a traditional application must continuously check the state of the I/O Line in order to determine when to call the subprocedure "MYEVENT_CODE". This takes up processing time for your application and can complicate the program's flow control.

In event-driven programs, changes in the hardware's state can cause a subprocedure to be executed even when the program flow was not expecting to do so. Therefore, the order in which your code executes depends on which events occur, which in turn depends on what the hardware does. This is the essence of event-driven programming: your code responds when it needs to.

Because the your program's flow control does not dictate when an event will occur, your code must make a few assumptions about the "state of the world" when it executes. Your code may trigger additional events in response to its own interaction with the hardware and so, you should try to structure you application so those possibilities are anticipated.

Note: Avoid performing to many events at once. This could cause the stack to overflow.

Code that starts at power up.

In OOPic Basic Syntax, the subprocedure named "MAIN" in your program is designated as the starting point. When your application powers up or is reset, the program will start executing at the first line within this subprocedure. If you want to stop the program and wait for an event to occur, simply drop out of the "MAIN" subprocedure. Another way to stop the program is by setting the OOPic Object's **Operate** property to 0 (**cvOff**). This will also cause the OOPic to shut down all events and go into a low-power mode.

Prior to the "MAIN" subprocedure executing, When the OOPic is powered up or is reset, all of the application's Object will be created and their values initialized allowing the "MAIN" subprocedure to access them.

Ending an Event-Driven program.

An Event-Driven OOPic application stops running when all its **oEvent** Objects are disconnected from their triggers and no code is executing. Because no code is running and no events will occur that will run a subprocedure, there is no way to re-start the execution of the program code.

Even when no code is running, the Objects will still operate in the background until the power is shut off. Since the **Operate** property can be set to 0 by another Object, or even by another computer via the I²C network, the code does not need to be running to shut down the power.

Objects

The OOPic language is oriented towards the use of Objects. Objects are a collection of code and data that works together as a unit. [Chapter 6](#) gives a detailed explanation of Objects. There are several Object classes that are defined in OOPic and are available for use in your applications..

In most basic languages, One would find several Basic language statements and functions whose operations deal with a particular subject and are therefore related. For example, the Peek function and the Poke statement both deal with memory. The Peek function returns the value at a specified location and the Poke statement sets a specified memory location to a specified value as seen in the following example.

```
V = Peek(128) 'Old way to read memory location 128 into the variable V.
Poke 128,V   'Old way to write the variable V into memory location 128.
```

The syntax of these two procedures is completely dissimilar. In OOPic Basic, these two operations are combined into a single Object called "[oRAM](#)". With the [oRAM](#) Object, a specified memory location is treated as a single unit and is accessed as if it was a variable. The following example demonstrates the use of an [oRAM](#) Object dimensioned with the name "MYRAM".

```
V = MYRAM    'New way to read a memory location into the variable V.
MYRAM = V    'New way to write the variable V into a memory location.
```

This code reads better, is more consistent and can be understood more quickly. This language enhancement is one of the benefits of using an Object-oriented language.

In OOPic applications an Object is declared (or an instance of it is created) with the [Dim](#) statement as follows:

```
Dim Object As New type 'Syntax of the Dim statement
```

Each Object must have a unique name specified by the programmer. The Object's name is case insensitive and is the identifier that is used by the application to refer to that Object. The same rules apply to naming Objects as are applied to all other identifiers in the OOPic language.

Variables

When writing programs, it is often necessary to store values temporarily when performing calculations. For example; to compare the result of several calculations one would need to retain the result of each calculation for the comparison. Storing the values in variables will accomplish this.

The OOPic languages, like most programming languages, uses variables for storing values. Variables have a name that you use to refer to the variable. And they have a value, which contains the data. Unlike other programming languages however, OOPic's variables are also Objects. This gives OOPic applications some advantages. The most important advantage is the presence of properties that can use to control the behavior of the variable. Another great advantage is that since the [Value](#) of the variable is also an Object property, it can be linked to a Virtual Circuit and accessed by other Objects as well as other computers via the I²C network.

In OOPic Basic syntax a variable is declared (or an instance of it is created) with the [Dim](#) statement as follows:

```
Dim variable As New type 'Syntax of the Dim statement
```

You should notice that this is the same syntax used to create the instances of any other type of Object.

Each variable must have a unique name specified by the programmer. The Object's name is case insensitive and is the identifier that is used by the application to refer to that variable. The same rules apply to naming variables as are applied to all other identifiers in the OOPic languages.

Subprocedures

The programmer can simplify programming tasks by breaking programs into smaller logical components called [subprocedures](#). A subprocedure is a section of a program used to perform a particular function. It is used when a particular function must be executed several times from different points within the program. Generally, a subprocedure can do anything that can be done in the main code of the program with the addition of returning to the point in the program that called it. The use of subprocedures saves program memory, program-entering time and makes programs easier to read and debug.

The [Call](#) command instructs the program to branch to the subprocedure. The last line of the subprocedure must be [End Sub](#). The [End Sub](#) statement causes the program flow to go back to the next statement following the [Call](#) statement.

Each subprocedure must have a unique name specified by the programmer. The subprocedure's name is case insensitive and is the identifier that is used by the application to refer to that subprocedure. The only required subprocedure name is "MAIN" which is the subprocedure that is executed when the OOPic is powered up or reset.

All variables in OOPic Basic syntax are global. This means that any subprocedure can modify any variable that appears anywhere in the entire program.

Functions

The OOPic languages include system-provided or intrinsic functions, like [Sin](#). A function is a procedure that can take an argument, perform a calculation and return the value of its result. To use a function, the procedure name and its argument in an expression form are typed into the code. For example, you could use the [Sin](#) function to get the Sine of an angle

```
newvalue = Sin(oldvalue)
```

Control Structures

Control structures allow you to control the flow of your program's execution. If left unchecked by control-flow statements, a program's flow will "fall" through all of the statement until reaching the bottom of program. While some very simple programs can be written with this unidirectional flow, most of the utility of any programming language comes from its ability to change the statements execution order with structures and loops.

Flow Control

Flow Control statements allow the program to change the point of execution.

The Flow Control structures that OOPic Basic syntax supports are:

- [Goto](#)
- [Call](#)

Decision Structures

Decision Structures are statements that can test conditions and then, depending on the result of that test, perform different operations. The condition is usually a comparison, but it can be any expression that evaluates to a

numeric value. The evaluation interprets this value as True or False; a value of 0 is False and a value of not 0 is True. The constants cvTrue and cvFalse can be used in your code in place of 0 or 1.

The decision structures that OOPic Basic syntax supports are:

- [If...Then...Else](#)
- [Select Case](#)

Loop Structures

Loop structures allow you to execute one or more lines of code repetitively.

The loop structures that OOPic Basic syntax supports are:

- [For...Next](#)
- [Do...Loop](#)

Adding Comments to Your Code

As you write code, you may often wish to include some commentary to the code about the insight you had when you wrote the program. The statement [Rem](#) tells an application written in OOPic Basic syntax to do nothing with the words following the [Rem](#) statement. Comments can follow a statement, or can occupy an entire line.

Copyright(c) 1999,2000 by Savage Innovations All rights reserved

•



OOPic Programmer's Guide

Chapter 6. - Introduction to OOP

Contents:

- [What is OOP?](#)
- [Why Objects?](#)
- [What Can You Do With Objects?](#)
- [Where Do Objects Come From?](#)
- [Creating Objects](#)
- [Working With Objects](#)
- [Controlling Objects with Their Properties](#)
- [Performing Actions with Methods](#)
- [Responding to Events](#)
- [Linking Objects Together](#)
- [Dynamic Data Exchange](#)

What is OOP?

OOP is an acronym for Object-Oriented Programming. It is a term used to identify languages that are oriented

towards the use of Objects.

Object-Oriented Programming, in reality, is just a concept. It is an effort to make the task of writing computer programs easier by allowing the programmer to write applications that interact with what appears to be physical objects.

Technically speaking, the Objects in an OOP language are combinations of code and data that are treated as a single unit. Each Object has a unique name and all references to that object are done by that name.

In OOPic, all the hardware I/O and the Virtual Circuits are done by using Objects, even variables are Objects.

Why Objects?

An Object-Oriented Programming language is advantageous because humans instinctively know how to interact with physical objects. By arranging a programming language so that a programmer can interact with cyber objects in the same way, the learning curve is reduced significantly.

In most computer languages, the processing and program functionality is found in the form of several dozen or so unrelated commands and keywords that need to be memorized. Several of these commands and keywords are proprietary and are not found in any other vendor's languages. This creates a situation where it is difficult to keep up with all the different "flavors" of a language.

In OOP languages, all of the processing and functionality has been moved to the Objects. The language itself can then be reduced to just the commands and statements that are necessary to handle the program flow and Object manipulation. This results in a reduced instruction set which in turn results in less to memorize.

In older programming languages, the processing was sequential throughout the application. With the processing moved to the Objects, the application becomes multi-tasking with each Object handling its own processing leaving the application to simply managing the Objects.

In OOPic, Objects are the fundamental way in which an application interacts with its environment. When an application is being created using OOPic, the programmer only needs to remember a small set of flow-control commands and how to interact with the Objects that his application uses.

What Can You Do With Objects?

Because the Objects in the OOPic environment encapsulate the functionality of the hardware found in the PIC16C74 IC, an application can interact with the hardware quickly. This is in contrast to the alternative, which would be working with a long list of memory locations that represent scattered bits and pieces of the same hardware.

An Object also provides multitasking code that interacts with the application program, hardware, and other Objects. This essentially provides the programmer with pre-written code libraries. For example, one could use an **oA2D** Object to read the voltage present at one of the I/O Lines on the OOPic. By simply creating an **oA2D** Object and setting its properties, the application would have a value available to it that represents a numeric scale of a voltage. There is no need for the programmer to be worried about how to actually do the analog to digital conversion, because the Object's multi-tasking code takes care of that.

Because Objects are multi-tasking, an OOPic application can interact with several pieces of hardware at the same time. This means that your applications don't have to stop doing one thing just to go do something else.

By linking Objects together, an application can create a Virtual Circuit. A Virtual Circuit is a circuit in an OOPic that appears to be a discrete, physical electronic circuit but is actually the Objects within the OOPic

emulating the functionality of the circuit. Virtual Circuits are created by pragmatically linking together a set of Objects in the same methodology as one would physically link together a set of electronic components. Once created, each individual part of the virtual circuit can be manipulated or evaluated by the program providing total computerized control of the entire circuit.

The following table describes the types of Objects you can use in OOPic languages.

<u>Type</u>	<u>Description</u>
<u>Hardware</u>	An Object that represents and/or encapsulates a physically implemented piece of hardware in the OOPic. In an application, multiple instances of these Objects can be declared, but only one per piece of hardware present in the OOPic can be operational at any given time.
<u>Processing</u>	An Object that retrieves values from other Objects, performs a specified calculation and then stores the resulting value in another Object. These Objects are the building blocks of Virtual Circuits. A processing Object can be declared for use in as many multiple instances as memory can hold.
<u>Variable</u>	An Object that stores a value and provides evaluation properties about that value. A Variable Object can be declared for use in as many multiple instances as memory can hold.
<u>System</u>	An Object that controls one of several system functions. A system Object is intrinsic and is present at the time the OOPic powers up. You cannot declare new instances of these Objects.

Where Do Objects Come From?

A single statement in the application's code creates an Object . In OOPic Basic syntax, the programmer would use the **Dim** statement in his application to do this. The **Dim** statement expects the programmer to specify a unique name for the new Object and what kind of Object it should be. Then, when the application is run and it encounters this statement in code, an instance of that type of Object will be created.

The following example demonstrates the statement used to create an instance of an Object where the Object's name would be "XYZ" and the Object's type would be **oByte**.

```
Dim XYZ As New oByte
```

OOPic has several different types of Objects that an application can create. Each one of these different types has what is called an Object class. An Object class defines how each Object works. To understand the relationship between an Object and its class, think of an automobile and it's blueprints. The blueprints would be the class, because it defines the characteristics of each car - for instance, size, shape and how the car works. This blueprint is then used to create one or more automobiles and the automobiles that are created, would be the Objects. Each Object that is created from a class is considered to be an instance of that class.

When an Object is created, it is an identical copy of its class. Once it exists as an individual Object, it can be customized by changing its properties. For example, if you create 3 Objects using the same class, each one of the 3 new Objects is an identical instance of that class. Each of the 3 new Objects shares a common set of characteristics and capabilities (properties, methods, and events), defined by the class. However, each one must be created with a different name and once created, each Object's properties and methods can be separately controlled.

All of the **Dim** statements are required to be at the beginning of the application's program. When the OOPic is

initially powered up, or after the OOPic is reset, the **Dim** statements will be the first items encountered, thus all of the application's Objects will be created and their values initialized before any other code is executed.

Creating Objects

In the OOPic Basic syntax programming language, an application must declare its intent to create an instance of an Object by using the **Dim** command. This statement allocates memory out of the free RAM area to hold the instance of the Object. The programmer gives the new Object a name that the application program will use to identify it with. Any attempt made in the application program to refer to an Object that has not yet been created will result in a compile time error and the compilation will not complete.

The following example demonstrates the statement used to create an instance of an Object where the Object's name would be "TheNumberThatWeWant" and the Object's type would be **oByte**.

```
Dim TheNumberThatWeWant As New oByte
```

Object Names

Each Object within an application program must have a unique name so that the application program can refer to it. Object Names can be anything so long as they follow standard naming conventions. The following lists the rules of the standard naming convention:

1. Object Names **must** begin with a letter
2. Object Names **cannot** contain a period.
3. Object Names **must not** exceed 32 characters.
4. Object Names **must** be unique within the application. (Object names are case insensitive)

The name of the Object is not stored in RAM and therefore can be any length (up to 32 characters) without affecting the amount of RAM that gets allocated for the Object's instance.

Object Names are case insensitive. I.E. the names: "ThingOne" and "thingone" are recognized as the same Object.

Creating Arrays of Objects

You can declare and use arrays of Objects by specifying a subscript value in the **Dim** statement. In the following example two arrays of Objects are being created, the first is an array of 3 **oByte** Objects and the second is an array of 6 **oDio1** Objects.

```
Dim BA(3) As New oByte
Dim PA(6) As New oDio1
```

These Objects are now accessed by including the subscript number. In the following example, All the values in BA(1) through BA(6) are added together in Z.

```
For X = 1 to 6
  Z = Z + BA(X).Value
Next X
```

Creating Objects from Classes with a variable size

The class definition of the three Objects [oBuffer](#), [oGate](#) and [oFanout](#) have variable sizes.

In the case of the [oBuffer](#) Object, the class size specifies how many bytes long the buffer will be. In the following example, an instance of the [oBuffer](#) class is created with 8 bytes of storage space.

```
Dim bunchobytes As New oBuffer(8)
```

In the case of the [oGate](#) Object, the class size specifies how many inputs the gate function will have. In the following example, an instance of the [oGate](#) class is created with 2 inputs.

```
Dim thegate As New oGate(2)
```

In the case of the [oFanout](#) Object, the class size specifies how many outputs the fan-out function will have. In the following example, an instance of the [oFanout](#) class is created with 2 outputs.

```
Dim somewires As New oFanOut(2)
```

Refer to the OOPic technical guide for a [list of Object classes](#).

Working With Objects

OOPic Objects supports properties, methods, and events. You can change an Object's characteristics by changing its properties. In addition to Properties, Objects have methods. Methods are a part of Objects just as properties are. Generally, methods are actions that are perform on the Object, while properties are the attributes you set or retrieve. Objects also have events. Events are triggered when some aspect of the Object is changed and program code is executed in response to this change.

Once an instance of an Object has been created, all references to that instance are done by specifying its name followed by a period and then the property or method that you want to reference.

Controlling Objects with Their Properties

An Object's properties are the values that it holds. These properties can be numeric values for storage or specifying what to do or even for indicating that something was done.

Setting Property Values

You would set the value of a property when you want to change the behavior of an Object. For example, you change the [Direction](#) property of a [oDio1](#) Object to specify whether you want the I/O Line to be an input or an output.

To set the value of an Object's property, specify the Object's name followed by a period and then the property to set.

The Following shows the syntax of setting an Object's property to a value:

```
Object.property = expression
```

The following example demonstrates a property being set:

```
Dim ThingA As New oDio1
Sub Main()
  ThingA.IOLine = 1
  ThingA.Direction = cvInput
End Sub
```

In this example, an Object named ThingA is told that its **IOLine** property is to be set to 1 and its **Direction** property is to be set to 1, (**cvInput**). **cvInput** is a constant that equals 1. Therefore, after this example is executed, the **Direction** property of the Object with the name "ThingA" is equal to 1. When the **Direction** property of an **oDio1** Object is 1, the **Value** property of that Object is updated with the electrical state of the I/O Line 1.

Some Objects have properties that are read only. These Objects generally have done some calculations to derive at the value set in the property and writing to them would only erase these values.

Getting Property Values

Reading the value of a property is done when the application program needs to find the state of an Object. For example, a program can determine whether or not a voltage is present at an I/O Line by reading the **Value** property of a **oDio1** Object which points to that I/O Line.

To get the value of an Object's property, specify the Object's name followed by a period and then the property to get.

The Following shows the syntax of getting an Object's property value:

```
Variable = Object.property
```

In the following example the value of one **oDio1** is read, and a second **oDio1** is set to its value.

```
Dim ThingA As New oDio1
Dim ThingB As New oDio1
Sub Main()
  ThingA.IOLine = 1
  ThingA.Direction = cvInput
  ThingB.IOLine = 2
  ThingB.Direction = cvOutput
  Do
    ThingB.Value = ThingA.Value
  Loop
End Sub
```

A property value can also be used in a complex expressions.

In the following code example, MyNumber is set to either 0 or 5 depending on whether or not there was a voltage present on the I/O Line at the time the statement is executed.

```

Dim ThingA As New oDio1
Dim MyNumber As New oByte
Sub Main()
  ThingA.IOLine = 1
  ThingA.Direction = cvInput
  MyNumber.Value = ThingA.Value * 5
End Sub

```

Using Default Properties

Many Objects have default properties. Default properties can be used to simplify the application's code, because the code does not need to refer explicitly to the default property when setting or retrieving the value. The default property can be set when the Object is dimensioned.

For an Object where [Value](#) is the default property, the following 2 code fragments are equivalent:

```
Object = 20
```

And

```
Object.Value = 20
```

Variable and Hardware type Objects will always use [Value](#) as the default property. Not all Objects have a default property, in these cases, specifying the properties will always be required.

In the following 2 code fragments, the value of the variable Q is incremented by one.

```
Q = Q + 1
```

And

```
Q.Value = Q.Value + 1
```

Objects that are used in a Virtual Circuit and are connected to other Objects will use the default properties of the Objects they are connected to, except in the case of explicitly pointing to an Object's [flag](#) properties.

Refer to the OOPic technical guide for a [list of Object properties](#).

Performing Actions with Methods

A method is an action that an Object can perform. When invoked, they instruct the Object to do a specified function.

Using Methods in Code

You would use a method when you want the Object to perform a specified function. For example, an Object will increase its [Value](#) property by 1 when the **Inc** method is invoked.

To invoke an Object's method, specify the Object's name followed by a period and then the method to invoke.

The Following shows the syntax of invoking an Object's method.

```
Object.method
```

The following example demonstrates a method being invoked.:

```
Dim ThingA As New oByte
Sub Main()
  ThingA = 1
  ThingA.Clear
End Sub
```

In this example, an Object named ThingA is told that to set its Value property to 1. The [Clear](#) method is then invoked which sets the Object's Value property to 0.

Refer to the OOPic technical guide for a [list of Object methods](#).

Responding to Events

An Event is any action that has happened within an Object. Often, an application program will need to perform a procedure when a particular Event has occurred.

The oEvent Object

The [oEvent](#) Object is a special OOPic Object that will execute a specified sub procedure when it's [Operate](#) property is set to 1. The [oEvent](#) Object's **Operate** property can be linked to a Virtual Circuit allowing any event in the OOPic to trigger events.

Linking Objects Together

Connections between OOPic Objects give the programmer the ability to build a Virtual Circuit within the OOPic.

Several of the Objects defined in the OOPic languages are capable of manipulating the properties of other Objects. This gives OOPic applications the power of a Virtual Circuit. Virtual Circuits can exchange data and do calculations in the background while the application program is focussing on what to do with, and how to react to the state of the Virtual Circuit. For example, in the application note "Driving a stepper motor" a Virtual Circuit is defined that will position a stepper motor at an absolute location. A single variable contains the value of that location and when changed, the stepper motor will turn to the new location. The only code that the application program has to do to turn the stepper motor to a particular location, is assign one variable the value of the new location. The following example demonstrates an assignment like this:

```
MotorLocation = 5000
```

The building blocks of Virtual Circuits are the Processing Objects. These Objects retrieve their input values and store their output values in the properties of the Objects that they are linked to. Objects are linked to other Object through pointers. Pointers are a data type that simply tells one Object where to find the other Object.

Two types of pointers exist;

1. A Pointer to a target Object.
2. A Pointer to one of the target Object's Flag properties.

A Pointer to a target Object tells the linked Object where the target Object is. A pointer of this type expects to be set to the Object itself. An Object link using this type of pointer will always set or return the target Object's [Value](#) property.

A Pointer to a target Object's Flag property tells the linked Object where the Flag property is. A pointer of this type expects to be set to an Object's Flag property. An Object link using this type of pointer will always set or return the target Object's Flag property that it was pointed to.

A Link is made between Objects when the Processing Object's pointers are linked to the target Object or its [Flag properties](#).

In the following example, two inputs of a [oGate](#) Object are connected to two [oDio1](#) Objects.

```
Dim aGate As New oGate(2)
Dim DioA As New oDio1
Dim DioB As New oDio1
Sub Main()
  aGate.Input1.Link(DioA.Value)
  aGate.Input2.Link(DioB.Value)
End Sub
```

Pointer types and the Object's property types must always match. In the previous example, the pointer types are matched. The input properties of an [oGate](#) Object must always be connected to [Flag properties](#) and the [Value](#) property of a [oDio1](#) Object is a Flag type property.

Dynamic Data Exchange

The OOPic programming language has the ability to communicate and exchange data with the Objects within an entirely different OOPic. This is helpful when you dedicate one OOPic to do a particular function and want to control its activities from another OOPic or other computer. The OOPic uses the Philips I2C two wire network to transfer information about the values within the Objects to other computers hooked into the I2C network. Up to 127 different components can be collectively connected to this network by attaching 2 wires and a ground reference. No specialized cables or components are required to use the I2C network.

For further information on the Philips I2C network, see the link to the Philips web sight at <http://www.oopic.com>

Copyright(c) 1999,2000 by Savage Innovations All rights reserved

•



OOPic Programmer's Guide

Chapter 7. - Variable and Constants

Contents:

- [What are Variables?](#)
- [oBit, oNibble, oByte and oWord Variables](#)
- [oBuffer Variable](#)
- [oEEProm Variable](#)
- [oRam Variable](#)
- [Constants](#)

What are Variables?

As discussed in [Chapter 5, "Programming Fundamentals"](#), Variables are used to store values. In most languages, variables are just a storage area that holds a single value. In the OOPic languages, however, variables are Objects that store a value. This is done so that each variable may have more properties than just its value. This gives OOPic applications some advantages. The most important advantage is the presence of properties and methods that can use to control the behavior of the variable. Another great advantage is that since the Value of the variable is a property, other Objects as well as other computers via the I2C network can access it. (See [Chapter 6, "Introduction to OOP"](#) for more information on how to create Variable Objects and use their properties and methods.)

OOPic provides you with several variables of different sizes and types of Variables to accommodate the needs of your program.

- [oBit](#), [oNibble](#), [oByte](#), and [oWord](#)
- [oBuffer](#)
- [oEEProm](#)
- [oRam](#)

[oBit](#), [oNibble](#), [oByte](#) and [oWord](#) Variables

The [oBit](#), [oNibble](#), [oByte](#) and [oWord](#) Variables are used to store values. Each of the 4 different Variables types work approximately the same, but the magnitude of the Value property is of different sizes.

Type	Bit-Size	Magnitude	Memory Size
oBit	1	1	1
oNibble	4	15	1
oByte	8	255	2
oWord	16	65535	3

Each Variable that is declared in an application will occupy an amount of memory within the OOPic, and it is always best to use the least amount of memory possible in any application. Therefore, to select the type of Variable to use when needing to store a value, determine the highest value that will be needed and select the Variable type with the lowest Bit-size that will still be able to hold that highest value within its magnitude.

These Variables all have a Flag-type [NonZero](#) property that indicates when the variable is not zero in value.

The two Variables, [oByte](#) and [oWord](#) have a Flag-type [MSB](#) property that indicates the state of the Most Significant Bit of the [Value](#) Property.

The [oBit](#) Variable has the unique feature of having its Defalut property also be a Flag-type property.

The Flag-type properties can be linked in Virtual Circuits with Flag-Pointers. ([See Chapter 9, "Virtual Circuits"](#) For more information on linking Flag-type properties to Virtual Circuits)

oBuffer Variable

The [oBuffer](#) Variable is used to store a contiguous group of bytes. It can be defined with as little as 1 byte or as many as 32 bytes of storage.

The [Value](#) property of the [oBuffer](#) Variable is a 1-byte value that access one of the elements in the contiguous group of bytes. The element that the [Value](#) property accesses is specified by the [Location](#) property.

oEEProm Variable

The [oEEProm](#) Variable is used to store or retrieve data from non-volatile memory located within the EEPROM memory chips plugged into the E0 and E1 sockets on the OOPic.

Care must be taken when using this Object, as it has the power to access and change the currently running application program.

oRam Variable

The [oRam](#) Variable is used to access any one of the 256 locations within the OOPic processor chip.

Extreme care must be taken when using this Object, as it has the power to access and change critical OS operations.

Constants

A Constant is a value that is given a name. Like a Variable, a constant is referred to in code to retrieve its value. However, unlike a Variable, its value can only be defined once within the code and cannot be changed during program execution.

Copyright(c) 1999,2000 by Savage Innovations All rights reserved

•



OOPic Programmer's Guide

Chapter 8. - Interfacing with Hardware

Contents:

- [OOPic's Hardware Interface](#)
- [Using Hardware Objects](#)
- [1-Bit Digital I/O](#)
- [4-Bit Digital I/O](#)
- [8-Bit Digital I/O](#)
- [16-Bit Digital I/O](#)
- [16-Bit Digital I/O Expansion](#)
- [Analog to Digital Converter](#)
- [Pulse Width Modulator](#)
- [Serial Transmitter / Receiver](#)
- [High Speed Timer](#)
- [Keypad Input](#)
- [Servo Controller](#)

OOPic's Hardware Interface

The OOPic is designed to be used in embedded applications. these are applications in which a stand-alone processor is required to control some form of hardware. To accommodate this, the OOPic has thirty-one I/O lines that can be used in various ways. Each of the thirty-one physical I/O lines can both sink and source 25mA. I/O Lines 8 – 15 have an internal pull-up resistor that can be activated with the [Pullup](#) property of the [OOPic](#) Object. and I/O Lines 16 – 31 are Schmidt Trigger type inputs when in the input mode. These I/O lines are controlled by a set of special Objects referred to as Hardware Objects.

The Following table shows the electrical characteristics for all the OOPic's I/O lines.

Output Voltage when I/O line is set to output 0.	0 Volts
Max current supplied when I/O line is set to output 0.	25 Ma
Output Voltage when I/O line is set to output 1.	+5 Volts
Max current supplied when I/O line is set to output 1.	25 Ma
Max current supplied on all I/O lines combined	200 Ma

The voltage regulator on the OOPic is a TO-92 case 7805 +5 Volt regulator and is rated at 100 Ma. If more power is needed, a +5 voltage source can be supplied on the +5 lines of the [40 Pin I/O connector](#).

[Hardware Objects](#) are Objects that encapsulate the functionality of the physical hardware circuits within the OOPic. They interface with the physical hardware circuits connected to the OOPic and provide an application program with a hardware interface that has the capability of controlling and respond to that hardware.

Hardware Objects are the one and only method the OOPic uses to provide hardware I/O.

Using Hardware Objects

Like all other Objects, Hardware Objects must be created before they can be used. In Basic syntax, this is done with the Dim statement. The Dim statement expects the programmer to specify a unique name for the Object and a Class specification to tell what kind of Object to create. For more information on using the Dim statement, refer to Chapter 3.

There are 11 Classes of Hardware Objects built into the OOPic. The following list shows the 11 different types of Hardware Objects available.

1. 1-Bit Digital Input / Output
2. 4-Bit Digital Input / Output
3. 8-Bit Digital Input / Output
4. 16-Bit Digital Input / Output
5. 16-Bit multiplexed Digital Input / Output
6. Analog to Digital Converter
7. Pulse Width Modulator
8. Serial Transmitter / Receiver
9. High Speed Timer
10. Matrix Keypad Decoder
11. RC Servo Controller

Although more than one instance of any hardware Object is allowed, Only one instance of a Hardware Object will properly interact with any one specific hardware circuit at any given time.

1-Bit Digital I/O

The **oDio1** Object class defines an Objects that uses one of the OOPic's thirty-one Input/Output Lines.

Each dimensioned instance of the **oDio1** Object has an **IOLine** property that specifies which one of the OOPic's thirty-one physical I/O lines it is to use. This property must be set prior to any other property being set. Setting the **IOLine** property to 0 results in the **oDio1** Object entering an dormant state.

The following table shows the possible values for the **IOLine** property.

IOLine	Description
0	The oDio1 Object is disconnected from any I/O line.
1 – 31	The oDio1 Object is connected to the specified I/O line.

The **Value** property of the **oDio1** Object continuously reflects the electrical state of the I/O Line specified by the **IOLine** property. If the **Direction** property is set to **cvInput**, then the **Value** property reflects the physical I/O Line and if the **Direction** property is set to **cvOutput**, then the physical I/O Line reflects the **Value** property.

The following table shows the possible values for the **Value** property.

Value	Electrical State
0	The I/O line is at 0 Volts
1	The I/O line is at +5 Volts

For more information on the **oDio1** Object, See the [oDio1 Technical Information](#).

4-Bit Digital I/O

The **oDio4** Object class defines an Objects that uses a group of 4 consecutive lines of the OOPic's thirty-one Input/Output Lines.

Each dimensioned instance of the **oDio4** Object has an **IOGroup** and a **Nibble** property that specifies which one of the six possible I/O line configurations it is to use. These property must be set prior to any other property being set. Setting the **IOGroup** property to 0 results in the **oDio4** Object entering an dormant state.

The following table shows the possible values for the **IOLine** property.

IOGroup	Nibble	Description
0	0	the oDio4 Object is disconnected from all I/O Lines.
0	1	the oDio4 Object is disconnected from all I/O Lines.
1	0	the oDio4 Object is connected to I/O lines 8 – 11.
1	1	the oDio4 Object is connected to I/O lines 12 – 15.
2	0	the oDio4 Object is connected to I/O lines 16 – 19.
2	1	the oDio4 Object is connected to I/O lines 20 – 23.
3	0	the oDio4 Object is connected to I/O lines 24 – 27.
3	1	the oDio4 Object is connected to I/O lines 28 – 31.

The **Value** property of the **oDio4** Object reflects the electrical state of the I/O Lines specified by the **IOGroup** and **Nibble** properties. If the **Direction** property is set to **cvInput**, then the **Value** property reflects the physical I/O Lines and if the **Direction** property is set to **cvOutput**, then the physical I/O Lines reflect the **Value**.

The following table shows the possible values for the **Value** property.

Value	Electrical State
0	All 4 I/O lines are at 0 Volts
1-14	The 4 I/O lines are binary equivalent combination of 0 Volts and +5 Volts
15	All 4 I/O lines are at +5 Volts

For more information on the **oDio4** Object, See the [oDio4 Technical Information](#).

8-Bit Digital I/O

The **oDio8** Object class defines an Objects that uses a group of 8 consecutive lines of the OOPic's thirty-one Input/Output Lines.

Each dimensioned instance of the **oDio8** Object has an **IOGroup** property that specifies which one of the three possible I/O line configurations it is to use. This property must be set prior to any other property being set. Setting the **IOGroup** property to 0 results in the **oDio8** Object entering an dormant state.

The following table shows the possible values for the **IOLine** property.

IOGroup	Description
0	The oDio8 Object is disconnected from all I/O Lines.
1	The oDio8 Object is connected to I/O lines 8 – 15.
2	The oDio8 Object is connected to I/O lines 16 – 23.
3	The oDio8 Object is connected to I/O lines 24 – 31.

The **Value** property of the **oDio8** Object reflects the electrical state of the I/O Lines specified by the **IOGroup** property. If the **Direction** property is set to **cvInput**, then the **Value** property reflects the physical I/O Lines and if the **Direction** property is set to **cvOutput**, then the physical I/O Lines reflect the **Value**.

The following table shows the possible values for the **Value** property.

Value	Electrical State
0	All 8 I/O lines are at 0 Volts
1 - 254	The 8 I/O lines are binary equivalent combination of 0 Volts and +5 Volts
255	All 8 I/O lines are at +5 Volts

For more information on the **oDio8** Object, See the [oDio8 Technical Information](#).

16-Bit Digital I/O

The **oDio16** Object class defines an Objects that uses a group of 16 lines of the OOPic's thirty-one Input/Output Lines.

Each dimensioned instance of the **oDio16** Object has an **IOGroup** property that specifies if it is using its predetermined I/O line configuration. This property must be set prior to any other property being set. Setting the **IOGroup** property to 0 results in the **oDio16** Object entering an dormant state.

The following table shows the possible values for the **IOLine** property.

IOGroup	Description
0	The oDio16 Object is disconnected from all I/O Lines.
1	The oDio16 Object is connected to I/O lines 8 – 15 and I/O lines 24-31.

The **Value** property of the **oDio16** Object reflects the electrical state of the I/O Lines specified by the **IOGroup** property. If the **Direction** property is set to **cvInput**, then the **Value** property reflects the physical I/O Lines and if the **Direction** property is set to **cvOutput**, then the physical I/O Lines reflect the **Value**.

The following table shows the possible values for the **Value** property.

Value	Electrical State
0	All 16 I/O lines are at 0 Volts
1 - 65534	The 16 I/O lines are binary equivalent combination of 0 Volts and +5 Volts
65535	All 16 I/O lines are at +5 Volts

For more information on the [oDio16](#) Object, See the [oDio16 Technical Information](#).

16-Bit Digital I/O Expansion

The [oDio16x](#) Object class defines an Objects that uses a group of 8 and a group of 3 lines of the OOPic's thirty-one Input/Output Lines but provides 16 Inputs and 16 outputs through multiplexing.

For more information on the [oDio16x](#) Object, See the [oDio16x Technical Information](#).

Analog to Digital Converter

The [oA2D](#) Object class defines an Objects that uses 1 Input line of the OOPic's thirty-one Input/Output Lines and encapsulates the OOPic's analog-to-digital converter module.

The analog-to-digital converter (A/D) module allows conversion of an analog input signal to a corresponding 8-bit digital number. The A/D module has four analog inputs available on the OOPic's I/O lines 1 – 4, which are multiplexed into a single sample and hold register. The sample and hold register is the input into the converter, which generates the result via successive approximation. The **ExtVRef** property of the **OOPic** Object determines the analog reference voltage, which is selectable to either 5 volts or the voltage level supplied on the OOPic's I/O line 4.

For more information on the [oA2D](#) Object, See the [oA2D Technical Information](#).

Pulse Width Modulator

The [oPWM](#) Object class defines an Objects that uses 1 Output line of the OOPic's thirty-one Input/Output Lines and encapsulates the OOPic's Pulse-Width-Modulator module.

The Pulse-Width-Modulator (PWM) module provides an output that rapidly switches between on and off. The ratio between the On time and the Off time is controllable. The larger the ratio is, the more current will flow from the output and the smaller the ration is, the less current will flow from the output. This creates an output who's output current is controllable.

For more information on the [oPWM](#) Object, See the [oPWM Technical Information](#).

Serial Transmitter / Receiver

The [oSerial](#) Object class defines an Objects that uses 1 Input line and 1 Output line of the OOPic's thirty-one Input/Output Lines and encapsulates the OOPic's serial communication (SC) module.

The serial communication (SC) module provides a high-speed serial communication port capable of communicating at 1200, 2400, 9600 and 31250 BPS. It can be configured as full duplex asynchronous or synchronous and can communicate with serial devices such as CRT terminals and personal computers. It uses the standard non-return-to-zero (NRZ) format asynchronous mode, (one start bit, eight data bits and one stop bit). The SC module transmits and receives the lowest significant bit first. The SC module's transmitters and receiver are functionally independent but use the same data format and baud rate.

The outputs of the SC module are TTL levels. To convert the TTL serial signals of 0 - 5 Volts to the RS232 Serial signals of +/- 12 Volts, any TTL to RS232 chip such as the SN75188 or the MAX203 can be used.

For more information on the [oSerial](#) Object, See the [oSerial Technical Information](#).

High Speed Timer

The [oTimer](#) Object class defines an Objects that uses 1 Input line of the OOPic's thirty-one Input/Output Lines and encapsulates the OOPic's Timer module.

The Timer module provides a 16 bit hardware counter who's input can be configured as an input line on the OOPic. The input line can be used a standard TTL level input or can be configured to drive a crystal.

For more information on the [oTimer](#) Object, See the [oTimer Technical Information](#).

Keypad Input

The [oKeypad](#) Object class defines an Objects that uses 8 lines of the OOPic's thirty-one Input/Output Lines and encapsulates the OOPic's Keypad matrix controller module.

The Keypad Matrix Controller module provides the matrix scanning pattern needed to read a 4 x 4 matrix keypad.

For more information on the [oKeypad](#) Object, See the [oKeypad Technical Information](#).

Servo Controller

The [oServo](#) Object class defines an Objects that uses 1 Output line of the OOPic's thirty-one Input/Output Lines and encapsulates the OOPic's Servo Motor (SM) control module.

The Servo Motor (SM) control module provides the PWM control pulse that is required to electrically refresh a servo motor's position. The SM control module can output the PWM control pulse on any of the OOPic's 31 I/O Lines specified by the **IOLine** property.

For more information on the [oServo](#) Object, See the [oServo Technical Information](#).

Copyright(c) 1999,2000 by Savage Innovations All rights reserved

•



OOPic Programmer's Guide

Chapter 9. - Virtual Circuits

Contents:

- [What is a Virtual Circuit?](#)
- [How Are Virtual Circuits Made?](#)
- [Pointer Properties](#)
- [Object Pointers](#)
- [Flag Pointers](#)

What is a Virtual Circuit?

A Virtual Circuit is a circuit in an OOPic that appears to be a physical discrete electronic circuit, but is actually the OOPic operating system emulating the functionality of the circuit.

A Virtual Circuit can be comprised of any number of any kind of Objects so long as at least one [Processing](#) Object is included. Each individual part of the Virtual Circuit can be manipulated or evaluated by the program providing total computerized control of the entire circuit.

How are Virtual Circuit made?

Virtual Circuits are created by pragmatically linking together a set of Objects in the same methodology as one would physically link together a set of electronic components to assemble an electronic circuit. The application's program can assemble, disassemble, or reconfigure the structure of a Virtual Circuit at any time during the execution of the program.

The Objects that are used to link together the Virtual Circuits retrieve their input values and store their output values in the properties of other Objects. These Objects are referred to as [Processing](#) Objects. These Objects encapsulate various mathematical, logical and other data manipulation functions. When they are linked to other Objects, their encapsulated function is performed on the "Linked" Objects. The Link that specifies what Objects are used, is done with a special Object Property called a Pointer.

Linking Objects

To pragmatically link together the Objects of a Virtual Circuit, The [Link](#) method is used in conjunction with the Objects and properties that are to be connected. This method creates a link between a two Objects. Once a link is created, the linking Object will use the linked property's value.

While other methods are used to instruct Objects to perform operations, the [Link](#) method is used to instruct a Pointer Property to point to another Object's property. The syntax for the [Link](#) method is as follows;

```
baseobject.pointerproperty.Link(linkobject.propertytolink)
```

The [Link](#) method is only available on properties that are designated as a Pointer Property.

Pointer Properties

A Pointer Property is an Object's Property that tells that Object where to find information in another Object. It does not store the information itself, instead each time the information is needed, the Pointer Property is used to identify where to retrieve the information from.

The Object's property that a Pointer Property is linked to can be changed at any time during the execution of the applications program.

Two types of Pointer Properties exist;

1. A Pointer to the target Object's Default property. referred to as an Object-Pointer
2. A Pointer to one of the target Object's Flag properties. referred to as a Flag-Pointer

When Linking Pointer Properties to Objects properties, the Pointer Property's type and the Object property's type must always match.

Object Pointers

An Object-Pointer is used to link a Processing Object to the Default Property of another Object. When linking a Pointer Property of this type, The argument that is expected inside a set of parentheses after the link method is the Target Object's Name plus a period and the Target Object's Default Property or just the Target Object's Name

The Default Property of an Object is the property that will be used if none is specified when referring to that Object. To find out which Object's properties are Default Properties, refer to the [Object's Technical Information](#) where each property's Data-Type is specified

The following example shows the Object-Pointers of the oMath Object being linked to the default properties of two other Objects.

```
Dim a As New oMath      ' Create an oMath Processing Object
Dim b As New oByte      ' Create a 1 byte Variable Object
Dim c As New oDio8      ' Create a 8-bit I/O port
Sub Main()
  a.Input1.Link(b.Value) ' Link the 1st oMath input to the oByte Object
  a.Input2.Link(c.Value) ' Link the 2nd oMath input to the I/O port
End Sub
```

Flag Pointers

A Flag-Pointer is used to link a Processing Object to a Flag-type Property of another Object. When linking a Pointer Property of this type, The argument that is expected inside a set of parentheses after the link method is the Target Object's Name plus a period and the desired Target Object's Flag-type Property.

A Flag-type Property is a property that consists of 1-bit and is configured in such a way to be able to be linked to. To find out which Object's properties are Flag-type Properties, refer to the [Object's Technical Information](#) where each property's Data-Type is specified

The following example shows the Flag-Pointers of the oGate Object being linked to the Flag-type properties of two other Objects.

```

Dim a As New oGate      ' Create an oGate Processing Object
Dim b As New oByte      ' Create a 1 byte Variable Object
Dim c As New oDio1      ' Create a 1-bit I/O port
Sub Main()
  a.Input1.Link(b.NonZero)  ' Link the 1st oGate input to 1st I/O Port
  a.Output.Link(c.Value)    ' Link the oGate's Output to the 2nd I/O port
End Sub
    
```

Copyright(c) 1999,2000 by Savage Innovations All rights reserved



OOPic Programmer's Guide

Chapter 10. - The OOPic System Object

Contents:

- [What is the OOPic Object?](#)
- [Controlling the OOPic Operating System.](#)
- [Reading OOPic Operating System Status.](#)
- [Controlling Miscellaneous OOPic Functions.](#)

What is the OOPic Object.

The **OOPic** Object is an intrinsic Object that controls the OOPic's operating system. Several properties are provided that either control operating system functions or report operating system status. The OOPic Object is the only Object that is intrinsic.

Since the **OOPic** Object is intrinsic, it is always present in every OOPic application and the application programmer does not need to dimension it.

Controlling the OOPic Operating System.

Four properties are provided to control the OOPic's Operating System:

- **Node**
- **Operate**
- **Pause**
- **Reset**

The **Node** property is a value that is used to specify the OOPics network node when two or more OOPics are connected via the I2C network.

The **Operate** property is a value that controls the power mode of the OOPic Chip.

The **Pause** property is a value that suspends the program flow.

The [Reset](#) property is a value that resets the OOPic when set.

Reading OOPic Operating System Status.

Three properties are provided that report system status.

- **Hz1**
- **Hz60**
- **StartStat**

The [Hz1](#) property is a value that cycles every 1hz.

The [Hz60](#) property is a value that cycles every 60hz.

The [StartStat](#) property is a value that indicates the cause of the last OOPic reset.

Controlling Miscellaneous OOPic Functions

Two miscellaneous properties are provided that control miscellaneous hardware functions.

- **ExtVRef**
- **PullUp**

The [ExtVRef](#) property is a value that specifies the source of the voltage reference for the OOPics analog to digital module.

The [PullUp](#) property is a value that specifies the state of the internal pull-up resistors on I/O lines 8 - 15.

Copyright(c) 1999,2000 by Savage Innovations All rights reserved

-



OOPic Programmer's Guide

Chapter 11. - OOPic Mathematics

Contents:

- [OOPic Mathematics](#)
- [Expressions](#)
- [Operators](#)
- [Precedence of Operators](#)
- [Arithmetic Operators](#)
- [Logical Operators](#)
- [Relational Operators](#)
- [Functions](#)

OOPic Mathematics

Any mathematical formula that needs to be solved while an application's program is running, requires that it be written out as an Expression in the program's code.

Expressions

An Expression is a written mathematical formula that details the values, operations and functions as well as the order that the operations and functions are performed in order for the formula to be solved. The resulting value of the mathematical formula is the final value derived after all of the formulas operations and functions have been executed.

An Expression can be a single value or variable, or it can be comprised of several values and variables separated by Operators and functions.

Expressions can be used in the following situations:

- In an assignment statement that assigns the resulting value to a Variable
- In a flow control statement that conditionally changes the flow of the program depending on the resulting value.

If an Expression is comprised of more than one value or variable, then it is evaluated from left to right, performing, in the order of precedence, the operators and functions that it encounters.

Operators

Operators are used to specify that a mathematical operation needs to be performed between two values. They are always performed by applying the value on the right side to the value on the left side in the manner specified by the operator.

The following is a list of Operators.

- Arithmetic: +, -, *, /, mod
- Logical: And, Or, Xor, Not
- Relational: =, <>, <, >, <=, >=

In the following example, the variable "A" is assigned the value of the Variable "B" plus the value of the variable "C" minus the value of the variable "D"

$$A = B + C - D$$

Precedence of Operators

The precedence of operators determines the order in which mathematical operations are executed. OOPic languages follow the operator precedence rules of algebraic expressions. These rules dictate that the expression is scanned from left to right and that no operation is performed until it encounters an operator of lower or equal precedence.

The following list defines OOPic's order of precedence:

1. Operators in parenthesis
2. Negation (-)
3. Multiplication (*), Division (/) and Modulus (Mod)
4. Addition (+) and Subtraction (-)
5. Relational expressions (=, <>, <, >, <=, >=)
6. Logical AND (And)
7. Logical OR (Or)
8. Logical XOR (XOr)

Parenthesized expressions have the highest precedence, so their use is a good way for you to reduce ambiguity and make your programs more readable.

In the following example, the variable "A" is assigned the value of the Variable "C" divided by the value of the variable "D" plus the value of the variable "B"

$$A = B + C / D$$

In the following example, the variable "A" is assigned the value of the Variable "B" plus the value of the variable "C" divided by the value of the variable "D"

$$A = (B + C) / D$$

Arithmetic Operators

The Arithmetic Operators perform basic arithmetic functions between two values.

Syntax	Resulting value
{expr1} + {expr2}	The sum of the two expressions.
{expr1} - {expr2}	The difference of the two expressions.
{expr1} * {expr2}	The product of the two expressions.
{expr1} / {expr2}	The quotient of the two expressions.
{expr1} Mod {expr2}	The remainder of the two expressions.

Logical Operators

The Logical Operators perform the bitwise logic functions between two values

Syntax	Resulting value
{ expr1 } And { expr2 }	The result of Logically ANDing the bits of the two expressions.
{ expr1 } Or { expr2 }	The result of Logically ORing the bits of the two expressions.
{ expr1 } Xor { expr2 }	The result of Logically Exclusivly ORing the bits of the two expressions.
Not { expr1 }	The result of Logically Inverting the bits of { expr1 }

Relational Operators

The Relational Operators perform comparisons between two values.

Syntax	Resulting value
{ expr1 } = { expr2 }	1 if the two expressions are equal, 0 if not.
{ expr1 } <> { expr2 }	1 if the two expressions are not equal, 0 if not.
{ expr1 } < { expr2 }	1 if { expr1 } is less than { expr2 }, 0 if not.
{ expr1 } > { expr2 }	1 if { expr1 } is more than { expr2 }, 0 if not.
{ expr1 } <= { expr2 }	1 if { expr1 } is less than or equal to { expr2 }, 0 if not.
{ expr1 } >= { expr2 }	1 if { expr1 } is more than or equal to { expr2 }, 0 if not.

Functions

Functions are used when one value needs to be converted from one form to another. They are performed by applying the value within the functions parenthesis to the conversion process specified by the function.

The following is a list of Function.

Syntax	Resulting Value
Sin({ expr })	The binary formatted trigonometric sine of { expr }

Copyright(c) 1999,2000 by Savage Innovations All rights reserved

-



OOPic Programmer's Guide

Chapter 12. -Dynamic Data Exchange (DDE)

Contents:

- [What is Dynamic Data Exchange?](#)
- [The oDDELink Object](#)
- [The DDE Virtual Circuit](#)
- [Network Nodes](#)
- [DDE Conversations](#)
- [DDE Masters](#)
- [DDE Slaves](#)
- [I2C Network Connections](#)

What is Dynamic Data Exchange?

Dynamic Data Exchange is a function that allows two or more OOPics to exchange data. This exchange of data is accomplished by a Virtual Circuit that utilizes [oDDELink](#) Objects to transfer data over an I2C Network.

While other processing Objects are used to create Virtual Circuits that manipulate and exchange data with Objects within the same OOPic, the [oDDELink](#) Object is designed to utilize the I2C network to exchange data with other OOPics.

The oDDELink Object

The [oDDELink](#) Object is a [Processing](#) Object that dynamically exchanges data between two OOPics by transmitting data over an I2C network of which both OOPics are connected. It can be configured to both send and receive data.

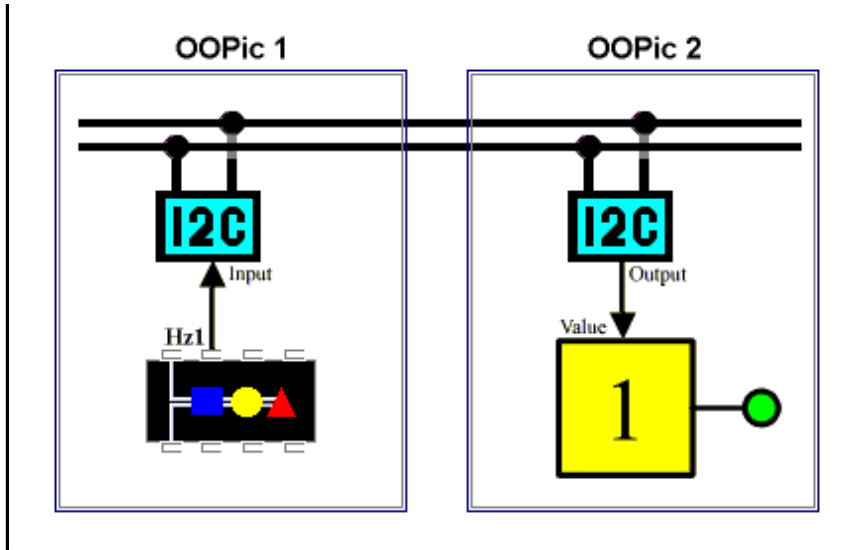
Any properly configured [oDDELink](#) Object can initiate a data transfer by setting the [Sync](#) property to 1 ([cvTrue](#)), thereby causing the data to be either sent to or retrieved from other [oDDELink](#) Objects in the network depending on the state of its [Direction](#) property.

The DDE Virtual Circuit

A working knowledge of Virtual Circuits is necessary to understand how the [oDDELink](#) Object functions. If you have not done so already, read [Chapter 3 - Your First Virtual Circuit](#) and [Chapter 9 - Virtual Circuits](#).

In order for a Virtual Circuit to be capable of transferring data over the I2C network, an [oDDELink](#) Object must be included and linked to other Objects within the OOPic. Each Occurrence of the [oDDELink](#) Object has two [pointer properties](#) which are used to link to the Objects which contain the data to be transferred.

The [oDDELink](#) Object retrieves the data to be transferred from the Object linked to by the [Input](#) property and the [Output](#) property links to the Object that the transferred data will be stored into.



The [oDDELink](#) Object's [Sync](#) property, which initiates the data transfer, is a [Flag type](#) property and can be controlled by the Virtual Circuit by linking it to other Flag-pointer properties within the Virtual Circuit.

Network Nodes.

Each OOPic connected together via an I2C network must be differentiated by a unique Node number. The [OOPic](#) Object's [Node](#) property is used to specify this number. This is the identifier that the I2C network uses to properly route the data when an [DDELink](#) transfer is made.

The [OOPic.Node](#) property must be set to a number greater than 0 to turn on the OOPic's network functions. Once the Node number is set, the I2C network is enabled and starts to listen for incoming data packets. If the [OOPic.Node](#) property is set back to 0 ([cvOff](#)), the I2C network is disabled and any incoming data packets are ignored.

It is important to remember to enable the I2C network in both the Master application and the Slave application before the applications attempt to transfer data.

The [oDDELink](#) Object also has a [Node](#) property. This property specifies which OOPic the [oDDELink](#) Object is going to be communicating with.

DDE Conversations.

When two OOPics exchange information, they do so by engaging in a DDE conversation. The application that initiates the conversation is called the "Master-Application", or just the "Master"; and the application responding to the Master is the "Slave-Application", or just the "Slave". An application can be configured to engage in several conversations, acting as the Master in some and the Slave in others.

A single [oDDELink](#) Object can operate as both the Master and the Slave in a [DDELink](#) conversation. In addition, it can act as the Master in a [DDELink](#) conversation with one OOPic, and the Slave in an entirely different [DDELink](#) conversation with a different OOPic.

To initiate a [DDELink](#) conversation, the [oDDELink](#) Object that is to be used as the Master must be properly configured, pointing to the Node and Location of the Slave [oDDELink](#) Object with the [Operate](#) property set to 1. Once it is configured, setting the Master's [Sync](#) property to 1, initiates the [DDELink](#) conversation which synchronizes its data with the Slave.

DDE Masters.

The Master [oDDELink](#) Object always controls the DDE conversation. Whether it is sending data to, or receiving data from the Slave, the Master always initiates the conversation.

In order for the Master to activate a DDE conversation, it must be configured to specify which OOPic it intends to communicate with and the location of the Slave [oDDELink](#) Object within that OOPic.

To specify the OOPic that it intends to communicate with, it must set the Master [oDDELink](#) Object's [Node](#) property to the same number that the Slave application's [OOPic.Node](#) property has been set to.

To specify the Slave [oDDELink](#) Object's location, it must set the Master [oDDELink](#) Object's [Location](#) property to the same number that the Slave [oDDELink](#) Object's [Address](#) property is set to. Note: The [Address](#) property of all Objects is set when the application is compiled and cannot be changed by the application's program.

In order for the Master to send data, its [Input](#) property must be linked to the Object that contains the data to send, and in order for it to receive data, its [Output](#) property must be linked to the Object to store the received data into.

How the Master oDDELink Object operates:

When an [oDDELink](#) Object's [Sync](#) property is set to 1 ([cvTrue](#)), it first checks to see if its [Operate](#) property is also 1 ([cvTrue](#)). If it is not, the [oDDELink](#) Object remains dormant. If, on the other hand, the [Operate](#) property is 1, then the [Node](#) and [Location](#) properties are checked. If either property is 0. then the operation is canceled. If, on the other hand, they are set to values of more than 0, then the DDELink conversation is initiated.

Once a DDELink conversation begins, the Master checks to see how the [Direction](#) property is set. If it is set to 0 ([cvSend](#)), then the Master sends data to the Slave, and if it is set 1 ([cvReceive](#)), then the Master receives information from the Slave.

When the Master sends data to the Slave, it retrieves the data to send from the Object that its [Input](#) property was linked to. It then sends this data to the Slave which then stores the data into the Object that its [Output](#) property is linked to..

When the Master receives data from the Slave, it first sends a request to the slave, which instructs the slave to retrieve the data out of the Object that its [Input](#) property is linked to. The Master then draws this data out of the Slave and stores it in the Object that its [Output](#) property is linked to.

The following is an example of an OOPic application that uses a Master [oDDELink](#) Object in a Simple Virtual Circuit.

```

Dim Master As New oDDELink
Sub Main()
    OOPic.Node = 1
    Master.Input.Link(OOPic.Hz1)
    Master.Node = 2
    Master.Location = 43
    Master.Direction = cvSend
    Master.Operate = cvTrue
    Do
        If Master.Tranmitting = cvFalse then
            Master.Sync = 1
        End If
    Loop
End Sub
    
```

```

    End If
  Loop
End Sub

```

DDE Slaves.

The Slave [oDDELink](#) Object continuously monitors the I2C network for DDELink communications from a Master [oDDELink](#) Object. Until it detects a DDELink conversation request from the Master, it remains dormant.

In order for the Slave to engage in an DDELink conversation, it only needs to have its [Input](#) and [Output](#) properties configured to operate. The Slave's [Operate](#) property does not need to be set to 1 ([cvTrue](#)) in order to respond to the masters instructions.

How the Slave oDDELink operates:

When the Slave [oDDELink](#) Object is engaged in a DDELink conversation by the Master, it will respond by either sending data to the Master or by receiving data from the Master.

If the Master is requesting data from the Slave, then the Slave responds by retrieved data from the Object pointed to by its [Input](#) property, which is then sends to the Master.

If the Master is sending it data to the Slave, then the Slave will wait until the data packet has been received, Once it has the full packet of data, the Slave DDELink Object will copy that data into the Object pointed to be the [Output](#) property.

The following is an example of an OOPic application that uses a Slave [oDDELink](#) Object in a Simple Virtual Circuit.

```

Dim SLAVE As New oDDELink
Dim LED As New oDio1
Sub Main()
  OOPic.Node = 2
  LED.IOLine = 31
  LED.Direction = cvOutput
  Slave.Output.Link(LED.Value)
  Slave.Operate = cvTrue
End Sub

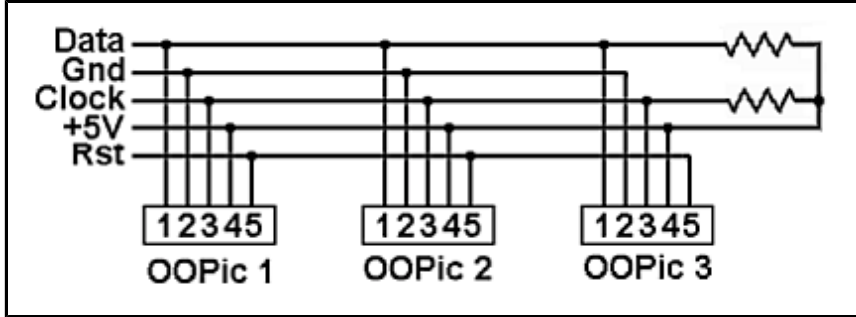
```

I2C Network Connections.

The I2C network is built in to the hardware of the OOPic, and runs at 19,200bps. It has 2 [5-Pin Network Connectors](#) available on the OOPic's circuit board which consists of the 5 lines; I2C Serial Data (**Data**), I2C Serial Clock (**Clock**), Ground (**Gnd**), +5 Volts (**+5V**) and Reset (**Rst**). The pin out of the 2 5-Pin Network Connectors are identical and can be used to daisy chain the networked OOPics together.

When connecting OOPics together with the I2C network, the 3 lines; **Data**, **Clock** and **Gnd** must be connected to each OOPic on the network, the 2 lines; **+5V** and **Rst** can be optionally connected depending on the applications need's.

A 4.7k pull up resistor is required on both the **Data** & **Clock** lines for the network to operate properly.



If the **Rst** line is connected to each OOPic on the network, then when any OOPic's Reset button is pressed, all of the connected OOPics will also reset. Note: Setting the [OOPic.Reset](#) property to 1 (**cvTrue**) in any of the network OOPics does not pull the **Rst** line low, and therefore will not reset all the OOPics on the network.

Copyright(c) 1999,2000 by Savage Innovations All rights reserved