

Copyright Notice

Copyright © Simon Brattel, 1996. All rights are reserved.

Trademarks

All trademarks that may be contained within this publication are registered with their respective companies.

TABLE OF CONTENTS

CHAPTER 1 : INTRODUCTION	5
1.1. INSTALLATION.....	5
<i>Getting Help</i>	5
1.2. INTRODUCTION.....	5
1.3. OVERVIEW	6
<i>Connecting to a Multi-IO Module</i>	6
CHAPTER 2 : CONFIGURING THE ACTIVEX CONTROL	7
2.1. CONFIGURING.....	7
2.2. SETTING UP THE ACTIVEX CONTROL (VISUAL BASIC).....	7
CHAPTER 3 : CONFIGURING THE MULTI-IO MODULES	18
3.1. INTRODUCTION.....	18
3.2. DEFINING A SESSION.....	19
3.3. DEFINING THE HARDWARE.	19
3.4. DEFINING THE PROCESSES.	22
3.5. DEFINING VARIABLES	23
3.6. SAVING THE SESSION	25
3.7. OPENING A SESSION.....	26
3.8. EDITING A SESSION	26
3.9. REPORT FILES.	27
3.10. THE SETUP WIZARD.	28
CHAPTER 4 : MIO ANALOGUE/DIGITAL IO BOARD.....	29
4.1. OVERVIEW	29
4.2. RESOURCE NAMES	30
4.2.1. <i>Analogue Inputs</i>	30
4.2.2. <i>Analogue Outputs</i>	30
4.2.3. <i>Digital Inputs</i>	31
4.2.4. <i>Digital Outputs</i>	31
4.2.5. <i>Module Variables</i>	32
4.3. BOARD CONFIGURATION.....	33
4.3.1. <i>Watchdog mode</i>	34
4.3.2. <i>Slot Info</i>	35
4.4. STEPPING MOTOR (4-PHASE).....	36

4.4.1. Setup	36
4.4.2. Usage	38
4.5. STEPPING MOTOR (PULSED)	39
4.6. PULSE WIDTH MODULATOR (PWM)	40
4.6.1. Usage	40
4.7. FREQUENCY COUNTER	40
4.7.1. Setup	41
4.7.2. Usage	41
4.8. CLOCK GENERATOR	42
4.8.1 Setup	42
4.9. TIMER	43
4.9.1 Setup	43
4.9.2. Usage	43
CHAPTER 5 : A MORE IN-DEPTH DISCUSSION OF THE ACTIVEX CONTROL	45
5.1. SETTING UP THE ACTIVEX CONTROL (MANUALLY)	50
CHAPTER 6 : DATA TYPES	52
6.1. OVERVIEW	52
6.2. COMMON DATA TYPES	52
<i>Enumerated Data Types</i>	53
CHAPTER 7 : PROPERTIES	56
7.1. PROPERTIES	56
7.2. ENABLED PROPERTY	57
7.3. INTERVAL PROPERTY	57
7.4. PROCESS PROPERTY	58
7.5. SESSIONNAME PROPERTY	58
7.6. STATUS PROPERTY	58
CHAPTER 8 : METHODS	60
8.1. METHODS	60
8.2. ACTIVATEPROCESS	60
8.3. CLOSEMIOSESSION	61
8.4. ISPROCESSACTIVE	61
8.5. GETPROCESSES	62
8.6. GETVARIABLE	63
8.7. GETVARIABLES	64
8.8. OPENMIOSESSION	65

8.9. OPENSESSIONFILE.....	66
8.10. ADDITIONAL METHODS.....	67
8.10.1. <i>Quick Note</i>	67
8.10.2. <i>GetEnabled</i>	67
8.10.3. <i>GetInterval</i>	67
8.10.4. <i>GetProcess</i>	67
8.10.5. <i>GetSessionName</i>	67
8.10.6. <i>GetStatus</i>	68
8.10.7. <i>SetEnabled</i>	68
8.10.8. <i>SetInterval</i>	68
8.10.9. <i>SetProcess</i>	68
8.10.10. <i>SetSessionName</i>	69
8.10.11. <i>SetStatus</i>	69
CHAPTER 9 : EVENTS.....	70
9.1. OVERVIEW.....	70
9.2. ONMIOERROR.....	70
9.3. ONMIOTIMER.....	72
CHAPTER 10 : CONTROLLING STEPPER MOTORS.....	73
10.1. OVERVIEW.....	73
10.2. CONTROLLING A STEPPER MOTOR.....	73
CHAPTER 11 : NETWORKING THE MIO MODULES.....	73
CHAPTER 12 : EXAMPLE PROGRAMS.....	75
12.1. EXAMPLES.....	75
12.2. EXAMPLE #1.....	75
12.3. EXAMPLE #2.....	75
12.4. EXAMPLE #3.....	76
12.5. EXAMPLE #4.....	76
12.6. EXAMPLE #5.....	76
12.7. EXAMPLE #6.....	76
12.8. EXAMPLE #7.....	76
12.9. EXAMPLE #8.....	77
12.10. EXAMPLE #9.....	77
12.11. EXAMPLE #10.....	77

Chapter 1 : Introduction

1.1. Installation

To install the Multi-IO development software insert disk #1 into your floppy drive. You can then either double click on the 'setup.exe' contained on the disk to install the software or you can click on the install button on the "Add\Remove programs" applet contained in control panel to install the software.

The installation program will install all the tools needed to setup and communicate with the MIO modules. Also included in the installation are example programs files for the most common development environments and also example files for Integrated Development Environment (IDE).

Getting Help...

Technical support can be reached at:-

Tel: +44 1978 660 145

Fax: +44 1978 664 201

e-mail: mio@maelor-displays.co.uk

homepage: www.maelor-displays.demon.co.uk

1.2. Introduction

Welcome to the Multi-IO ActiveX control and configuration utilities. This suite of utility programs has been designed to allow you to easily setup and access the Multi-IO modules. All development environments that support the ActiveX control interface will be able to allow you to communicate with the I/O module.

Visual Basic and similar development environments have ActiveX as a central design feature, enabling you to quickly and easily set up controls through a common user interface.

ActiveX itself is an interface standard that allows applications to communicate with each other through a standard set of routines. This allows design environments like Visual Basic to access many different types of controls through one common user interface.

There is no restriction on what an ActiveX control can do or where it is located. ActiveX encapsulates many complex technologies which go beyond the scope of this document.

The current version of the ActiveX driver has not been designed to be used by an application across a network. If you do need to communicate across a network you should write a 'local' application which is capable of communicating across a network to control the Multi-IO modules. This will shield the Multi-IO from receiving possibly erroneous information.

It should be noted that ActiveX is an evolving interface standard that could change with time and so we reserve the right to change the software without prior notice.

1.3. Overview

The Multi-IO (MIO) is a very compact yet powerful I/O module capable of supporting 16 analogue inputs, 16 analogue outputs, 16 digital inputs and 16 digital outputs. Further modules can be daisy chained in parallel upto a maximum of 32 modules.

Each module can be setup independently and each can exist anywhere within the chain. Configuration for the modules is done through the standalone configuration program 'MIOConfig.Exe'. This utility will generate a file containing all the data needed to 'talk' to the MIO boards.

This configuration file holds all the data needed to initialise the MIO modules.

Connecting to a Multi-IO Module

Please refer to the Technical manual for connection details.

Chapter 2 : Configuring The ActiveX control

2.1. Configuring

The Multi-IO modules are configured in 2 stages. The 1st stage involves using the 'MIOConfig.Exe' configuration utility program to identify and configure the Multi-IO boards connected to your PC. The configuration program saves this information in a setup file which will later be needed by the ActiveX control to initialise the physical Multi-IO modules.

The 2nd stage involves the MIO ActiveX control module. If you are using Visual Basic you can select a configuration file and do all necessary setting up, including setting up default values, quickly and easily through the "property" pages.

The ActiveX control sits in between your application and the physical MIO modules. Its sole purpose is to handle information coming from and going to the MIO modules.

The control handles all communication with the MIO modules and collates all the necessary data. Your application simply 'talks' to the ActiveX control and the ActiveX control 'talks' to the MIO modules.

2.2. Setting up the ActiveX control (Visual Basic)

The easiest way to describe the setting up procedure is to walk through a simple example. This example will show you how to import the ActiveX control, configure it and then communicate with it from your program. A MIO module will have to be connected to your machine.

For this example our aim is to produce a simple application which will show the current voltage for analogue input 00 (please refer to the chapter on Resource Names for more information). The current value of the input, shown on the PC, will be updated each time the user clicks on it.

This example will assume that you have used Visual Basic (VB) before. If you are unsure about any of the topics discussed below please refer to your Visual Basic manual.

Step 1. Start Visual Basic.

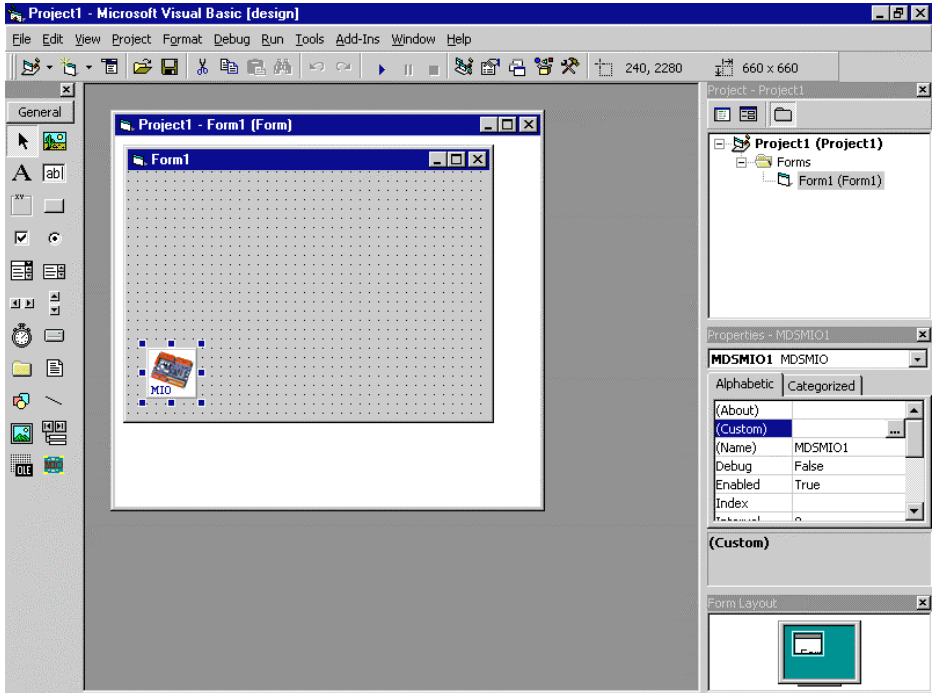
Step 2. Create a new “Standard EXE” project. This will create a new form page when you click the “Open” button.

Step 3. At this stage we have a blank form page. We now need to tell VB that we want to use the Multi-IO ActiveX in our project. To do this you will need to select the “Components...” menu item contained in the “Project” menu. Visual Basic will present you with a dialog box containing a list of ActiveX controls available to your system.

Select the “MDS Multi-IO control module” from the list by clicking in the small box to the left of the control name. Selecting the control will place a small tick within the box. Click the “OK” button to import the control into your project. A small Multi-IO icon will appear in your controls window, normally situated along the left hand edge of your screen.

Step 4. To place a Multi-IO control on your page click on the Multi-IO icon. Then move the mouse to the form then select an area. To do this press the left mouse down, then move your mouse which should draw a rectangular outline, finally release the mouse button. If you have made an error just repeat this step. Your page should now look like the one below.

Visual Basic will automatically assign a name by to the Multi-IO control, “MDSMIO1” in this case. This is the name by which you will refer to the ActiveX control throughout your program. Your page should now look like the page below.

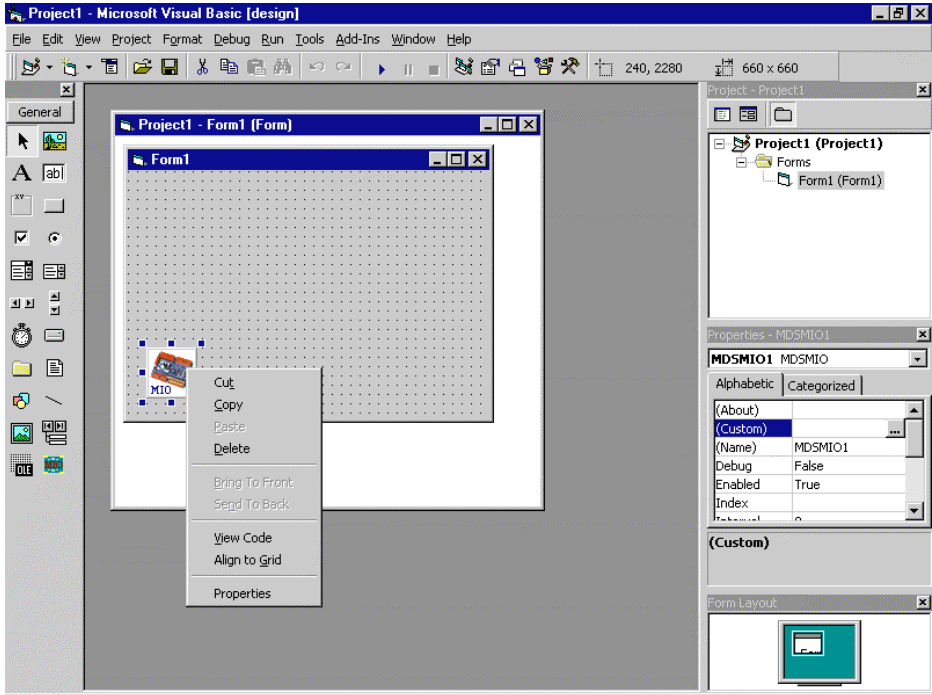


At this point we have created a new project, created a new form page, told Visual Basic which controls we want to use with our project and finally created a Multi-IO ActiveX control on our page.

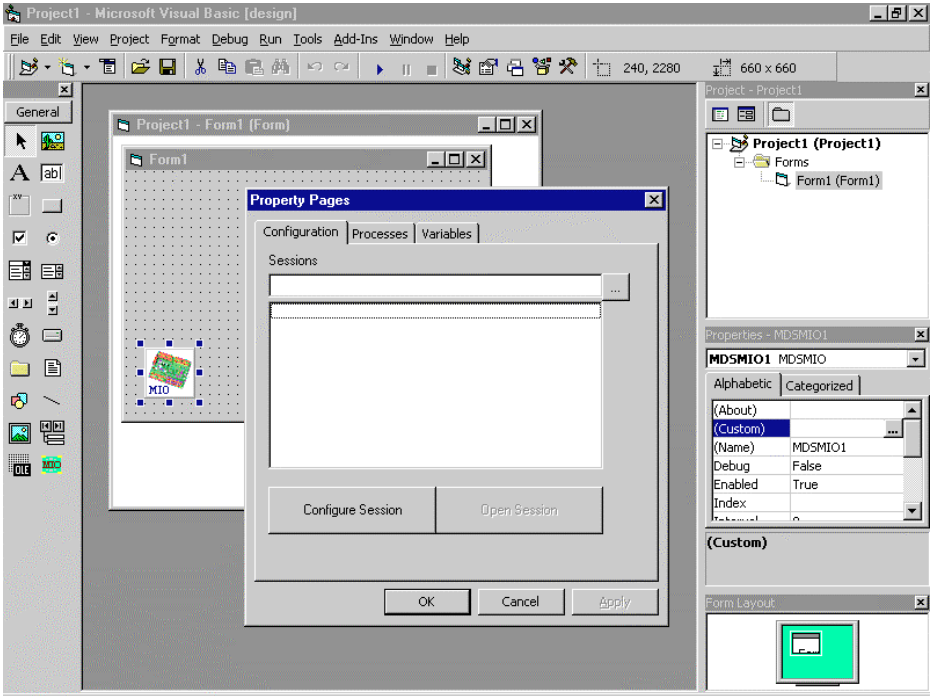
Step 5. The Multi-IO control doesn't at this point know about the physical Multi-IO modules connected to your PC. This information is held in a MIO configuration file created by the configuration utility (see configuring the MIO modules for more information). The configuration file holds information about the MIO boards connected to your PC, how each board is individually configured and which serial to use when communicating with the boards.

To tell the ActiveX control about the connected MIO modules we have to give it a configuration file to use. To do this right click on the Multi-IO icon on the form page, a small menu like the one below will appear.

Chapter 2: Configuring The ActiveX control

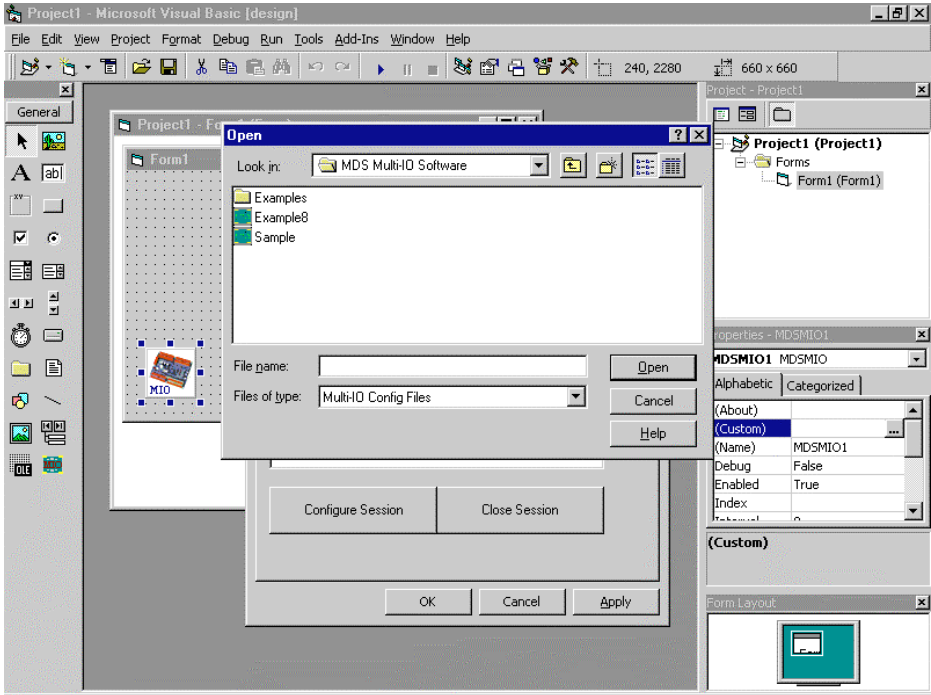


Click on the “properties” menu item, this will take you to the property pages shown below.



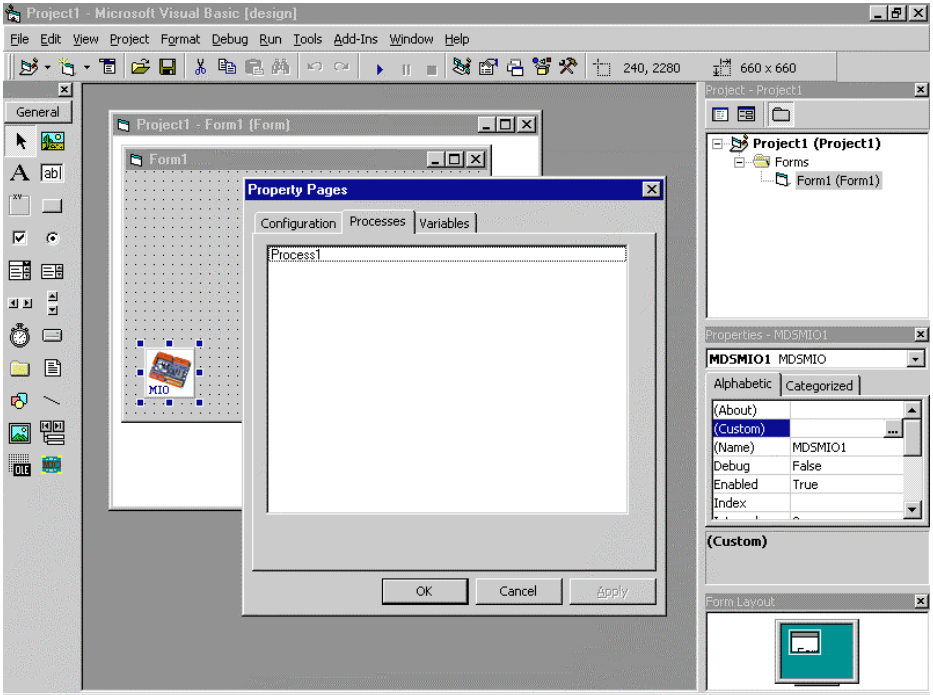
Clicking on the “...” button will allow you to locate a Multi-IO configuration file contained on disc. For this example the file we are looking for is called “sample” and the directory in which this file can be found is “Program Files\MDS Multi-IO Software” or in the directory into which you installed the MIO software.

Chapter 2: Configuring The ActiveX control



Opening the configuration file automatically initialises the ActiveX control. The control will do all the necessary setting up needed in preparation for communication with the connected MIO module.

Step 6. The final step in configuring the ActiveX control is to tell it which “process” it should be using. A “process” (discussed later in this manual) simply tells the control which data to access in the physical MIO modules. Selecting a “process” is done by first clicking on the “Processes” tab which will show the following page.



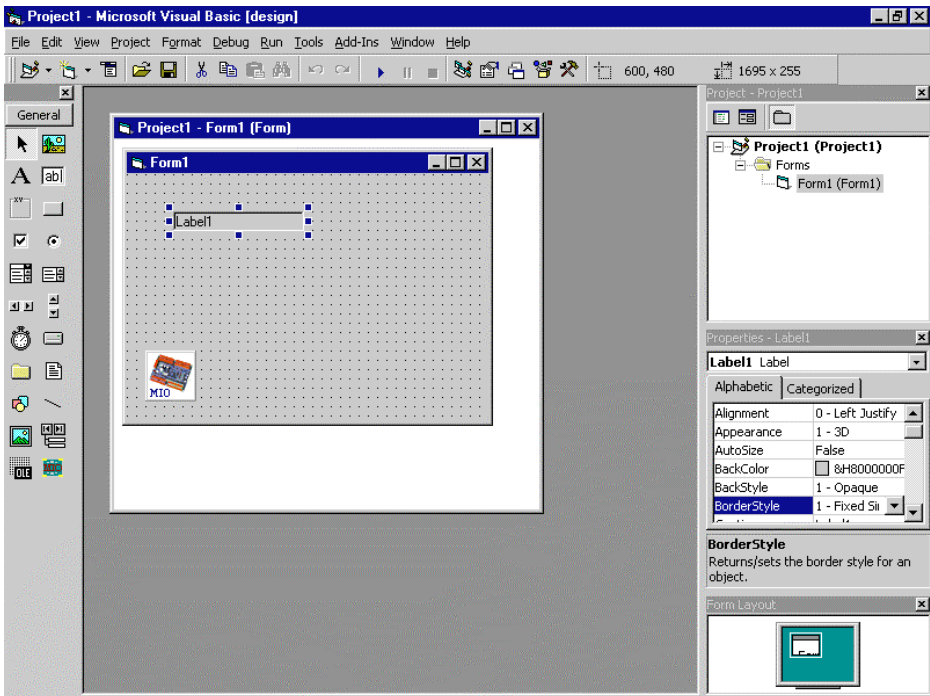
A list of available processes will appear in the list box. For this application we just have the one process called “process1”. Select it by clicking on it.

Step 7. All the configuring has now been done. We have told the Multi-IO control about the MIO modules connected to your PC by selecting a configuration file. It now knows where, how and what data to collect from MIO modules. The control will, at this point, be trying to communicate with the MIO modules even though you are still developing your project.

You can check that the ActiveX control is communicating with the connected MIO module. To do this click on the “Variables” tab. The data you will see will be ‘live’ data coming in from the MIO modules. If you run your fingers along the edge of the analogue inputs you should be able to see the values changing. Finally click on the “OK” button to close the property dialogue box.

Chapter 2: Configuring The ActiveX control

Step 8. We finally need one other control in our project. What we need is a label control. This will be used to display the current value of our analogue input. The label control can be found in the components window, it is the control which has an “A” inside of it. Click on it to select it. Then select an area in a similar way to how you placed the ActiveX control. Your page should now look something like the screen below.

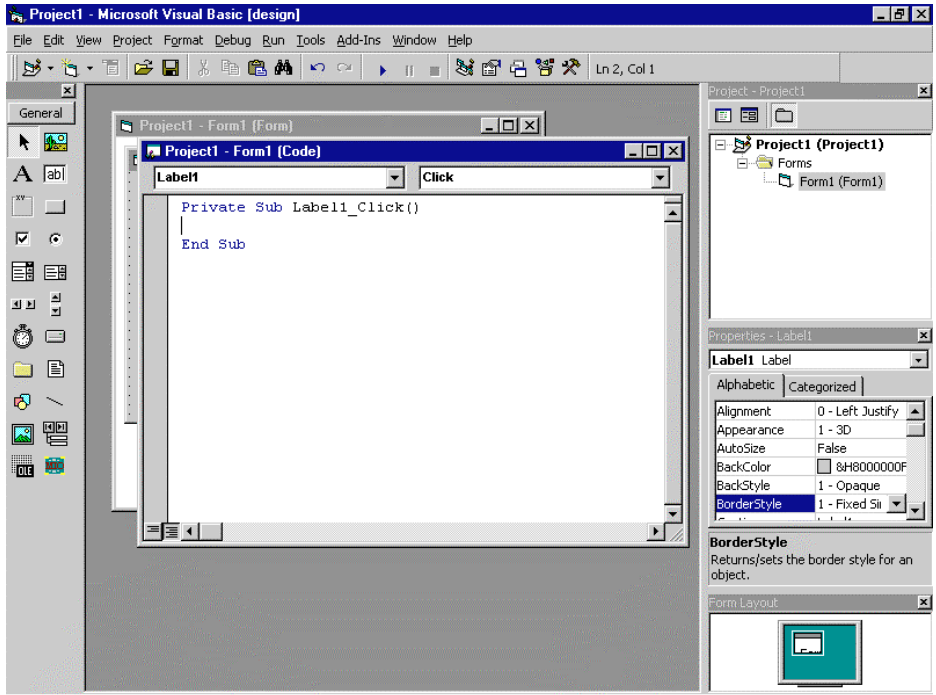


Step 9. As mentioned at the end of step 7 the Multi-IO ActiveX will already be gathering information from the MIO modules, what we now need to do is to make use of this information within our application.

Lets recap on what we are trying to achieve. We want to able to display the current value of an analogue input. When the user clicks on the label control. So what we

need to be able to do now is to tell VB that when the label is clicked we want it to pass control to us so that we can do something useful.

The simplest way to do this is to double click on the label control. VB will automatically assume that you want to be notified when the control is clicked and will generate an editing containing some lines of code (see the diagram below).



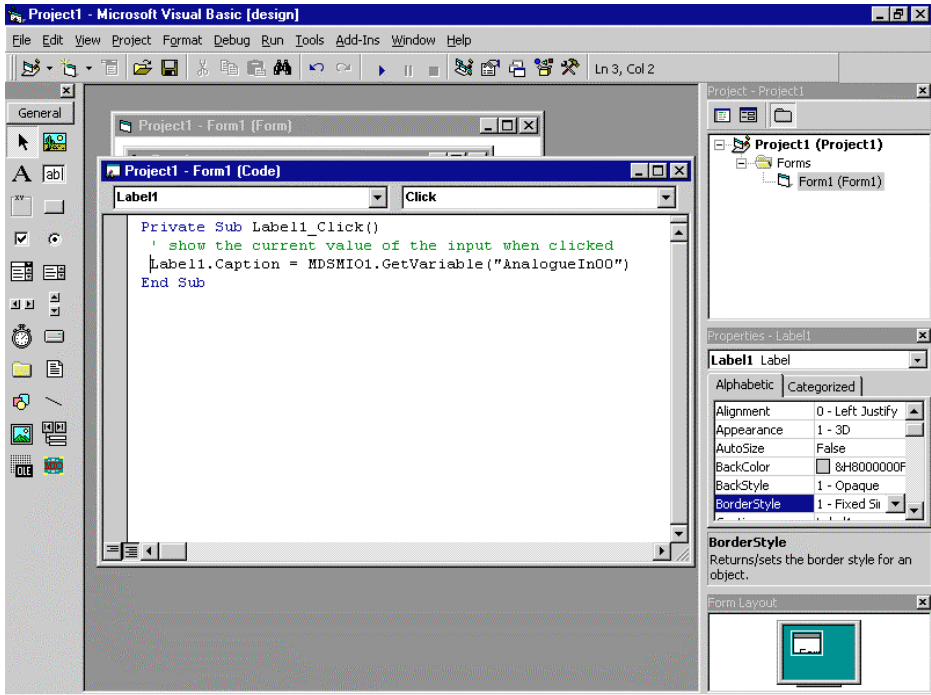
VB expects you to insert your code in between the 2 lines of code it has automatically created. For further information please refer to your Visual Basic manual.

For our example we only need to insert a very small amount of code. The line we need to insert is:-

```
Label1.Caption = MDSMIO1.GetVariable("AnalogueIn00")
```

Chapter 2: Configuring The ActiveX control

When you type this in, your screen will look like the one below.



This may look a little daunting at first and perhaps needs a little explaining. Lets break this statement done into 2 logical parts, the bit before the equals and the bit after the equal sign. Lets take the bit after the equal sign first.

`MDSMIO1.GetVariable("AnalogueIn00")`

This statement is simply saying get information about "AnalogueIn00" from the Multi-IO ActiveX control. The variable name "AnalogueIn00" is actually created during the configuration process – more about this later. The information retrieved by the above statement is the passed on to the 2nd bit of the statement

`Label1.Caption =`

This statement tells VB to display the given information in the label control as text.

So this whole statement gathers information from the ActiveX control and then displays this information in a human readable form on the PC.

You are now ready to try your application pressing F5 will run your application. When your application starts clicking on the label control, the current value of analogue input 00 being read from the MIO module will be displayed.

This example in some ways is over simplified. We haven't actually discussed how to set up a configuration file. This will be discussed later.

Using The Multi-IO Configuration Software

Chapter 3 : Configuring The Multi-IO modules

3.1. Introduction

The configuration program allows you to set up one or more MIO modules. All the information needed to communicate with the MIO boards is provided by the configuration data. This data is then stored in a configuration file or **session** file, which can later be used by the ActiveX control to talk to the boards connected to your PC.

The whole purpose of configuring the boards is solely for the purpose of the ActiveX control. The ActiveX control needs to be told what, how and where to get information from the MIO modules.

Configuring

Configuring allows you to select which resources will be used on each board. By **resource** (see Resource Names) we mean the physical attributes of a MIO module for instance an analogue input or perhaps a digital output. For each resource that you wish to access you will give a “name” to. These “names” are ultimately how you will access or control a resource from your application.

This information is then collated and stored in a configuration file called a “session”. The ActiveX driver uses this file to initialise the MIO hardware and to allow it to communicate with the controlling application program.

Resources which you wish to access are grouped together to form a **process**. The purpose of a “process” is really to restrict access a particular resource. This becomes more obvious when you consider that a single board could in theory be accessed by a number of ActiveX controls. It would be undesirable to allow all the different controls free access to all the resources as problems would soon arise.

Access rights

The process wholly owns resources allocated to it. This means that once an ActiveX control opens a process no other control can open the same process or indeed access any of the resources allocated to it.

This behaviour in certain circumstances could be restrictive. So fortunately it is possible to ‘allow’ other applications or processes access to certain resources by

changing the resource access flags. More often than not you will not to do this but if you do the facility is available.

The resource access flags determines whether an application, which doesn't own the resource, can access and/or control the resource. If you allow a resource read access then the application will only be allowed to read from the resource. If you allow a resource write access the application will only be allowed to write to the resource. A resource which has read/write access enabled will allow full access to any application.

Details of each process, which resources are to be utilised and how each resource can be accessed will be stored in the session configuration file.

MIO Slots

There are four 'slots' on the MIO, each of which can be set to anyone of the special modules supported by board. These slots do not have a real, physical, existence, but it helps to think of them as being four possible slots into which any one of the available modules can be plugged.

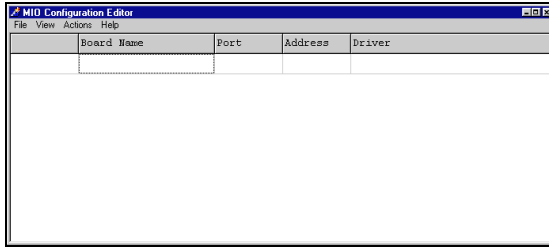
3.2. Defining a session.

On starting the configuration program, no configuration file will be loaded and the program window will appear as below.

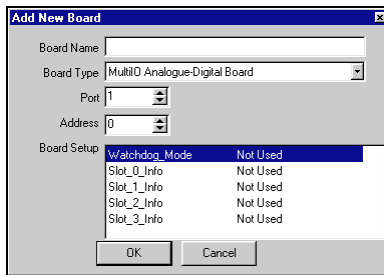


3.3. Defining the hardware.

The first step in generating a new configuration (session) file is to define what physical boards are connected to the PC. To do this we first show the board list by selecting View|Boards from the main menu.



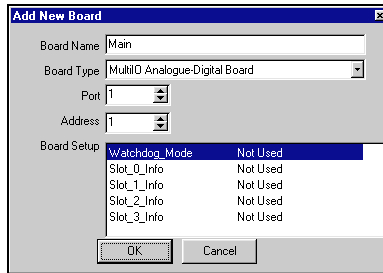
Initially empty we need to add a new board. To do this we call up the ‘Add New Board’ menu by selecting Actions|Add from the main menu.



From the dialogue box you can see that a number of items need to be set up. These items actually tell the ActiveX how to initialise the MIO modules.

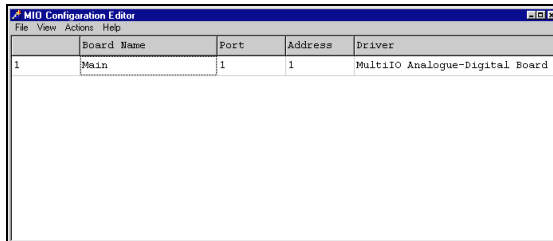
You must first insert a board name into the field “Board Name”. The board name is used throughout the configuration to reference one particular physical MIO module. At the time of writing all boards are of the same type.

“Port” sets which serial port the MIO module is connected. “Address” is extremely important, the number entered here **MUST** match the corresponding address on the MIO module. Each MIO module can have an address between 0 and 31 inclusive. The DIP switches on each MIO board set its address Id (see the Technical Manual for further information).



The Board Setup indicates how the **slots** will be used (for more information. On slots refer to the Technical manual). These “slots” can be configured independently of each other. Slots allow optional features to be used for example it could be possible to set up a slot to drive a stepper motor or to implement a pulse width modulator (PWM). The optional features depend upon the board type, at the time of writing only 1 board type is supported.

To set up an optional feature double click on a slot entry and a set up dialog for that item will appear. Refer to the chapter relating to each specific board type for a fuller explanation of the optional items on each board. Click OK when the board details are correct and they will appear in the board list.

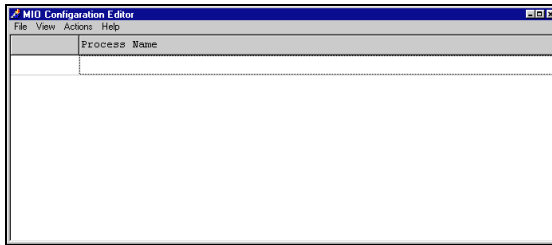


Repeat this action until all connected boards have been added to the board list. If you make a mistake or wish to amend any of the entries in the board list, highlight the entry and bring up the ‘Edit Board Menu’ by selecting Actions|Edit from the main menu or double click its entry in the board list.

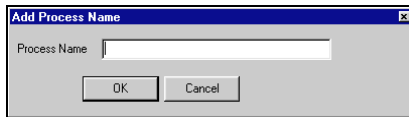
If you wish to delete a board from the board list, highlight the entry and select Actions|Delete from the main menu.

3.4. Defining the processes.

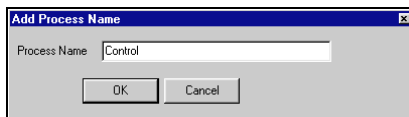
The next step is to decide how many applications will have access to the MIO resources. Each application needs to be given a “process” name which it will use to gain access to MIO resources. To do this we first show the process list by selecting View|Processes from the main menu.



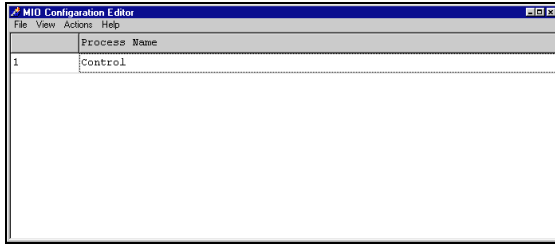
To add a new process we call up the ‘Add Process Name’ menu by selecting Actions|Add from the main menu.



Type in a name for the process and click OK. The name you choose doesn't really matter but something meaningful is preferable. I tend to use names like process1, process2, etc.



The process name will now be added to the process list.



Repeat this action until all processes have been added to the process list. If you make a mistake or wish to amend any of the entries in the process list, highlight the entry and bring up the 'Edit Process Name' by selecting Actions|Edit from the main menu or double click its entry in the process list.

If you wish to delete a process from the process list, highlight the entry and select Actions|Delete from the main menu.

3.5. Defining variables

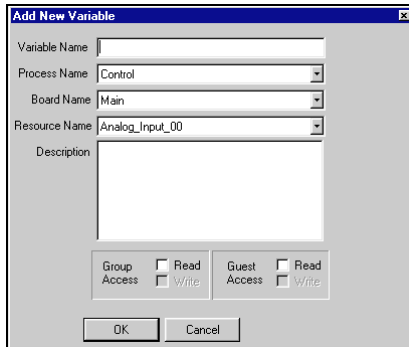
The final step in generating a session is to decide which board resources are going to be used and, in the case of a multi-application system, which applications have access to which resources.

Each resource to be used has to be allocated a variable name with which it will be accessed. To allocate a resource we first show the variable list by selecting View|Variables|All from the main menu.

The screenshot shows the 'MID Configuration Editor' window with the 'View|Variables|All' menu option selected. The window displays a table with the following data:

	Board	Object	Variable Name	Owner
1	Main	Slot_0_status		
2	Main	Slot_1_status		
3	Main	Slot_2_status		
4	Main	Slot_3_status		
5	Main	Analog_Input_00		
6	Main	Analog_Input_01		
7	Main	Analog_Input_02		
8	Main	Analog_Input_03		
9	Main	Analog_Input_04		

This will display a list of resources that are currently accessible across all the defined boards. To define a variable we call up the 'Add New Variable' menu by highlighting the resource and selecting Actions|Add from the main menu or double clicking its entry in the variable list.



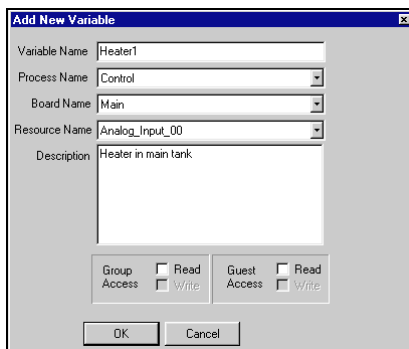
The screenshot shows a dialog box titled "Add New Variable". It contains the following fields and controls:

- Variable Name: [Empty text box]
- Process Name: [Control] (dropdown menu)
- Board Name: [Main] (dropdown menu)
- Resource Name: [Analog_Input_00] (dropdown menu)
- Description: [Empty text area]
- Group Access: Read, Write
- Guest Access: Read, Write (greyed-out)
- Buttons: [OK], [Cancel]

A name for the variable should be entered into the box provided and the process to which it belongs set in the 'Process Name' Box. A description of the variable can be typed into the description area and control over whether other processes have access to the variable can be set in the access Boxes. This description is only for your use.

There are two types of access rights, 'Group Access' and 'Guest Access'. By checking the group access boxes, any other named process will be allowed read and/or write access to the variable.

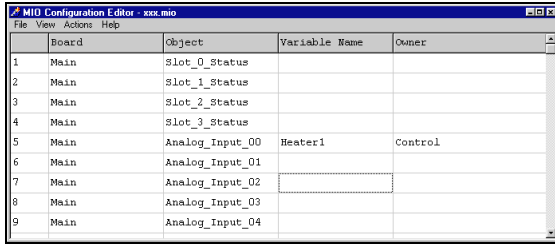
The guest access boxes are reserved for use by future MIO software products and should be left unchecked at present. If the resource that is being allocated to the variable is read-only (as in the example) the write access checkboxes will appear greyed-out.



The screenshot shows the same dialog box as above, but with the following data entered:

- Variable Name: [Heater1]
- Process Name: [Control] (dropdown menu)
- Board Name: [Main] (dropdown menu)
- Resource Name: [Analog_Input_00] (dropdown menu)
- Description: [Heater in main tank]
- Group Access: Read, Write
- Guest Access: Read, Write (greyed-out)
- Buttons: [OK], [Cancel]

When all data for the variable has been entered click OK and the variable details will be added to the variable list.

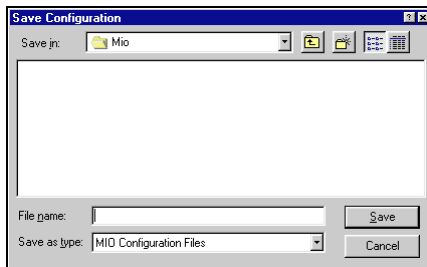


	Board	Object	Variable Name	Owner
1	Main	Slot_0_status		
2	Main	Slot_1_status		
3	Main	Slot_2_status		
4	Main	Slot_3_status		
5	Main	Analog_Input_00	Heater1	Control
6	Main	Analog_Input_01		
7	Main	Analog_Input_02		
8	Main	Analog_Input_03		
9	Main	Analog_Input_04		

Repeat this action until all variables have been added to the variables list. If you make a mistake or wish to amend any of the entries in the variable list, highlight the entry and bring up the 'Edit Variable' by selecting Actions|Edit from the main menu or double click its entry in the variable list. If you wish to delete a variable from the variable list, highlight the entry and select Actions|Delete from the main menu.

3.6. Saving the session

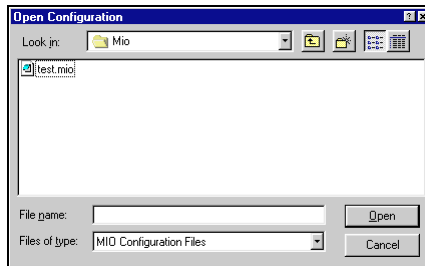
When all boards, processes and variables have been defined they must be saved to disk in order for the MIO communication driver to be able to access the data. To do this bring up the 'Save Configuration' menu by selecting File|Save from the main menu.



Type in a session name in the 'File_name' box and press Save to write the session to disk.

3.7. Opening a session

To review the contents of a previously save session file, it must firstly be loaded into the configuration program by bringing up the ‘Open Configuration’ menu by selecting File|Open from the main menu.



Select the session to load and press Open to read the session from disk. The configuration program will now display the session’s board list. To view the process and variable list select View|Processes and View|Variables|Defined repectively.

3.8. Editing a session

At any time it is possible to edit any of the configuration items that have been entered and the other item lists will be adjusted accordingly.

Processes can be added to the process list at any time and from that point on it is possible to re-assign variables to that process. If a process is renamed, all references to the old process name are also updated to the new name. If a process is deleted and variables have been assigned to that process, a warning will be shown that in order to delete the process, the attached variables will also be deleted.

In a similar way boards can be added, renamed, edited or deleted from the session and the variable lists will be amended accordingly.

Once the edited session file is saved to disk, the next time an MIO application opens that session file all changes will take effect.

Whenever an application program reads or writes a variable it does so by sending a request to the MIO communication driver. The MIO communication driver uses

the information in the configuration file to translate these requests into the information sent to specific MIO boards via a PC serial port.

This allows some system reconfiguration to be made without having to recompile the application programs. For example, boards can be moved onto different serial ports/station numbers or variables can be moved to different IO pins or even onto different MIO boards.

3.9. Report Files.

Whenever a session file is saved, the configuration program will write a report file that will list information on the session. This information will include

- The type of each board
- A list of the defined processes
- A list of the variables on each board
- A list of the variables accessible by each process
- The connector assignment of each board

This file will have the same filename as the session file but with a .txt extension.

3.10. The Setup Wizard.

The setup wizard will guide you through the first few steps needed to create a configuration file. Follow the instructions and enter the data required. When complete the setup wizard will take you directly to the variable definition page discussed earlier.

Decide which modules and resources you wish to use, assign names to them (so you can reference them from the ActiveX control), set access rights if need be, then save the file. Configuration done.

Chapter 4 : MIO Analogue/Digital IO Board

4.1. Overview

This board provides 16 digital inputs, 16 analogue inputs, 16 digital outputs and 16 analogue outputs. Each of these **resources** is identified in the variable list by a unique resource name.

In addition, additional resources are defined that allow reading and writing of groups of inputs and outputs simultaneously.

The board is also capable of driving outputs under its own control to perform some tasks that would be difficult or, at best, slow to handle remotely, such as driving stepping motors or generating pulse width modulated (PWM) outputs. The board does this by making use of software modules that reside internally.

These modules, more commonly known as **slots**, are completely independent of each other, and up to 4 modules can be used simultaneously in any combination. It is possible, for example, to be moving 3 stepping motors and generating a PWM output at the same time.

Whenever a module is setup, extra resources will be made available in the variable list. You can then use these extra resources in your application.

Some modules directly control some of the digital outputs. Whenever this occurs, the digital outputs will not be available for direct control from your application. So their resource names will be removed from the variable list.

If these digital outputs had already been allocated to variables, a warning will be generated before these variables are deleted.

Whenever a module takes direct control of digital outputs, it allocates four adjacent outputs whose position is dependent on the slot number involved.

<u>Slot No</u>	<u>Outputs used</u>
Slot 0	Digital outputs 0 to 3
Slot 1	Digital outputs 4 to 7
Slot 2	Digital outputs 8 to 11
Slot 3	Digital outputs 12 to 15

4.2. Resource Names

These are the names given by the MIO configuration software to describe each of the resources present on the board that the application program can use. This could be a physical input or a output but could equally be a resource that control or reports the state of some internal process.

If the application program wishes to use a particular resource, you should use the configuration program to assign a meaningful variable name to this resource. This achieves three objectives, it gives the application programmer an easy-to-remember handle with which to access the resource.

Secondly it removes any direct links between the application program and the resource which allows reconfiguration in the field.

Lastly it allows the MIO communication driver to generate a list of used resources that allows it to minimise the amount of information interchanged with the hardware to allow the MIO system to operate at its optimum performance.

4.2.1. Analogue Inputs

There are sixteen analogue input resources which are referred to as follows

<u>Internal Name</u>	<u>Description</u>
Analog_Input_00	Analogue input 0
Analog_Input_01	Analogue input 1
..	
Analog_Input_15	Analogue input 15

4.2.2. Analogue Outputs

There are sixteen analogue output resources which are referred to as follows

<u>Internal Name</u>	<u>Description</u>
Analog_Output_00	Analogue output 0
Analog_Output_01	Analogue output 1
..	
Analog_Output_15	Analogue output 15

4.2.3. Digital Inputs

There are sixteen digital input resources which are referred to as follows

<u>Internal Name</u>	<u>Description</u>
Digital_Input_00	Digital input 0
Digital_Input_01	Digital input 1
..	
Digital_Input_15	Digital input 15

In addition there are two resources that allow multiple inputs to be read simultaneously.

Inputs. This resource returns the current state of all sixteen digital inputs combined into a single value. Bit 0 represents the state of digital input 0, bit 1 represents the state of digital input 1, etc.

Raw Inputs. This resource returns the current state of all sixteen digital inputs prior to being filtered by the MIO logic. All states are combined into a single value. Bit 0 represents the state of digital input 0, bit 1 represents the state of digital input 1, etc.

4.2.4. Digital Outputs

There are sixteen digital output resources which are referred to as follows

<u>Internal Name</u>	<u>Description</u>
Digital_Output_00	Digital output 0
Digital_Output_01	Digital output 1
..	
Digital_Output_15	Digital output 15

In addition there are other resources that allow multiple outputs to be written to simultaneously.

<u>Internal Name</u>	<u>Description</u>
Outputs_Bank_0	Writes Outputs 0 to 3
Outputs_Bank_1	Writes Outputs 4 to 7
Outputs_Bank_2	Writes Outputs 8 to 11
Outputs_Bank_3	Writes Outputs 12 to 15
<u>Internal Name</u>	<u>Description</u>

Outputs_Lower	Writes Outputs 0 to 7
Outputs_Upper	Writes Outputs 8 to 5
Outputs	Writes Outputs 0 to 15

4.2.5. Module Variables

The optional modules each define a number of resources that can be used by the application to control its operation. In the examples below, the variable name corresponds to the name defined if the module were used in Slot 0. Replace the ‘_0_’ with ‘_1_’, ‘_2_’ or ‘_3_’ for use in the other slot positions. See later in this chapter for more information on the use of these resources.

Stepper.

Slot_0_Step_Cmd
Slot_0_Step_Count
Slot_0_Step_Position
Slot_0_Step_Mode
Slot_0_Acceleration
Slot_0_Min_Delay
Slot_0_Max_Delay
Slot_0_Step_PowerDown
Slot_0_Limit_Neg
Slot_0_Limit_Plus

Pulse Width Modulator(PWM).

Slot_0_PWM_Period
Slot_0_PWM_OnTime

Frequency Counter.

Slot_0_Count_Mask
Slot_0_Count_Period
Slot_0_Count_Total
Slot_0_Count_Freq

Clock Generator.

Slot_0_Clock_Mask

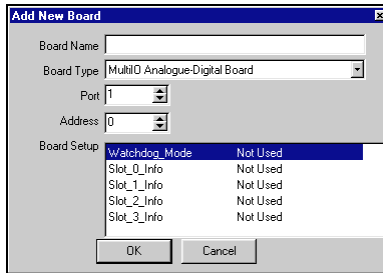
Timer.

Slot_0_TMR_0_State
Slot_0_TMR_0_Time
Slot_0_TMR_0_Start_Polarity
Slot_0_TMR_0_Start_Mask
Slot_0_TMR_0_Run_Polarity
Slot_0_TMR_0_Run_Mask
Slot_0_TMR_1_State
Slot_0_TMR_1_Time
Slot_0_TMR_1_Start_Polarity
Slot_0_TMR_1_Start_Mask
Slot_0_TMR_1_Run_Polarity
Slot_0_TMR_1_Run_Mask
Slot_0_TMR_2_State
Slot_0_TMR_2_Time
Slot_0_TMR_2_Start_Polarity
Slot_0_TMR_2_Start_Mask
Slot_0_TMR_2_Run_Polarity
Slot_0_TMR_2_Run_Mask

NOTE. It should be noted that defining a slot as a stepper, PWM or Clock Generator will reassign the digital outputs associated with that slot. The frequency counter modules do not use any digital outputs so they will remain available for use as general purpose outputs.

4.3. Board configuration

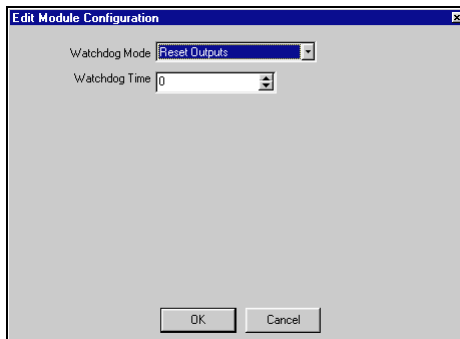
The board has a number of optional features that are setup when the board is added to the board list. The 'Add New Board' menu will show these options in the board setup box.



The options include a Watchdog Mode and the setup for the four modules that reside within the MIO board. To setup each item, double click on its entry in the list and a setup menu for that item will be displayed.

4.3.1. Watchdog mode

The watchdog mode is used to set the board's behaviour whenever it loses communication with the control application, eg if the application crashes or a fault develops in the interconnecting communication cabling.



This variable is by default 'Not Used' but can be set to

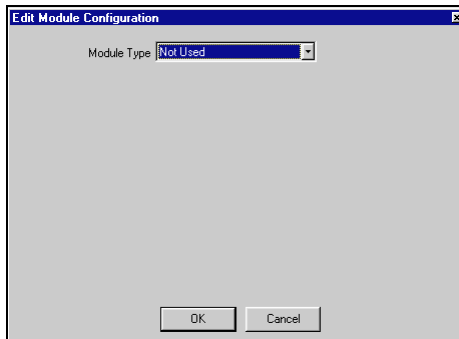
- Reset Outputs. This will zero all board outputs after Time x 10 mS.
- Reset Board. This will reset the board after Time x 10 mS.

The watchdog settings for each board are stored in the session file and are used to initialise the board when the application starts up.

4.3.2. Slot Info

There are four 'slots' on the MIO, each of which can be set to one of the modules supported by board. These slots do not have a real, physical, existence, but it helps to think of them as being four possible slots into which any of the specialist modules can be plugged.

Setting up one of the slots will bring up the 'Edit Module Configuration' menu.



A slot module type can be determined from the info box. Changing the module type may cause other associated information to be set up and to be displayed within the menu.

The module type of each slot together with any other set up information for that module is stored in the session file and is automatically used to initialise the MIO boards when the application starts up.

4.4. Stepping Motor (4-phase)

This module drives small stepping motors using the digital outputs to drive the coils directly. It supports motors which require less than 1.5A, less than 40V and which have unipolar drive. It does not support any other number of phases, or higher current or voltage. In order to do this it takes control over 4 of the digital outputs on the board.

This module can drive stepping motors as slowly as 210 half-steps/second up to as fast as 2100 steps/second, and it supports a limited form of velocity profiling in that it is possible to set a start-rate, a minimum step delay and an acceleration factor.

This module supports both half and full stepping, it defaults to half-steps.

Each motor movement consists of three sections, an initial acceleration period where the step rate increases from an initial rate to a set maximum, a period of fixed speed movement and finally a deceleration period where the step rate decreases down to zero.

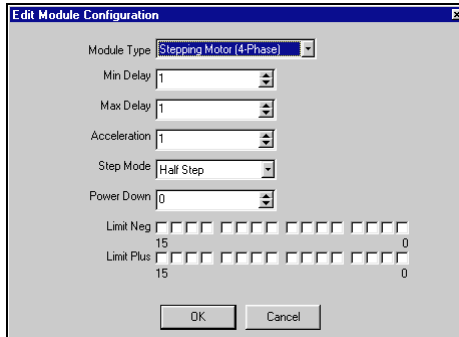
If desired, any set of the digital inputs can be used as end-stop signals, and these will stop the motor if driven. The clockwise and anti-clockwise end-stops need not be the same, and it is possible to support extra digital inputs which will stop the motor at any time (for emergency stops, etc).

This module holds the current motor position to 32-bits, in other words as a number in the range 0 to 4,294,967,296. The motor move commands can either take the form of move to a given (32-bit) position, or move in a given direction by a given (32-bit) distance.

It is possible to set a motor time-out period so that after every motor-move the MIO will wait for the time-out period to elapse before ceasing to drive any phases and thus ‘floating’ the motor. This can be used to save power in applications that do not require any holding torque.

4.4.1. Setup

The Stepper Motor (4-Phase) setup menu is as follows



The stepping motor module also defines the following additional setup information.

- **Min Delay**. This setting is the minimum value of the time interval between adjacent steps. The smaller this value the higher the step rate when the motor has finished accelerating.
- **Max Delay**. This setting is the maximum value of the time interval between adjacent steps. This value specifies the initial step rate.
- **Acceleration**. This setting determines the rate at which the step rate changes from the Max Delay to the Min Delay value and back again during the step cycle.
- **Step Mode**. This setting determines whether the motor will operate in half or full stepping modes. Half stepping results in a halving of the step rates but will allow greater positioning accuracy of the motor as it will double the number of steps for each revolution of the motor.
- **Power Down**. This setting specifies a timeout period after which the MIO will power down the motor. The timeout period is the value of this setting multiplied by 33 microseconds.
- **Limit Neg**. These check boxes allow one or more of the digital inputs to cause an immediate termination of stepper movement when the motor is moving in a negative direction.
- **Limit Plus**. These check boxes allow one or more of the digital inputs to cause an immediate termination of stepper movement when the motor is moving in a positive direction.

4.4.2. Usage

The module defines a number of extra resources that may be assigned to variables to allow the user program control over the stepping motor.

These resources initiate movement in the stepper motor or return feedback of the current position of the motor. Resources are also available to set the velocity profile and limit switches.

The stepping motor module defines the following variables

- **Step Command**. This is used to start or stop any stepper movement when used in conjunction with the Step Count value. The following commands are supported.
- **Stop Stepping**. This command causes any stepper motion to be aborted
- **Stepper Move**. This command causes the stepper to move the number of steps specified by the Step Count value. The Step Mode value determines the direction of stepping and whether half or full stepping is employed.
- **Stepper Relative Move**. This command causes the stepper to move the number of steps specified by the Step Count value. The Step Mode value determines whether half or full stepping is employed. The sign of the Step Count determines the step direction.
- **Stepper Absolute Move**. This command causes the stepper to move to a specified absolute position. The step distance and direction of movement is determined automatically from the difference between the current motor position and the target position. The Step Mode value determines whether half or full stepping is employed.
- **Reset Position**. This command zeroes the current motor position making the current position the origin or start position. All step movements are done relative to the start position.
- **Step Count**. This is used to specify a relative distance or absolute step position for the step commands.
- **Step Mode**. This determines the method of stepping employed by the step commands
 - 2 Full Step Reverse (Step command 2 only)
 - 1 Half Step Reverse (Step command 2 only)
 - 1 Half Step Forward
 - 2 Full Step Forward
- **Step Position**. This returns the current absolute position of the stepper motor.
- **Power Down**. This sets the period for the stepper power down timer.

- **Min Delay, Max Delay, Acceleration.** These setup the velocity profile of any movement of the stepper motor. They can be accessed if control over the ramp profile is required from within the application program.
- **Limit Neg, Limit Plus.** These allow the application program to set which of the digital inputs control the limits of stepper travel. Bit 0 selects digital input 0, bit 1 selects digital input 1, etc. More than one bit can be set in each mask which will cause the stepper to halt if any of the selected inputs are activated.
- **Slot Status.** The slot status is set by the stepper module to reflect its current operating status. It defines four flags that appear in the bottom bits of the status as follows
 - 0 Motor is in motion.
 - 1 Motor motion completed.
 - 2 Motor motion was aborted by limit switch.
 - 3 Motor motion was aborted by command.

4.5. Stepping Motor (Pulsed)

This module generates direction and step pulses which can be used to drive an external stepping-motor driver module to drive stepping motors which are too large to be driven directly from the MIO outputs, or which do not have 4 phases.

In order to do this it takes control over 4 of the digital outputs on the board.

This module can drive stepping motors as slowly as 210 steps/second up to as fast as 2100 steps/second, and it supports a limited form of velocity profiling in that it is possible to set a start-rate, a minimum step delay and an acceleration factor.

Each motor movement consists of three sections, an initial acceleration period where the step rate increases from an initial rate to a set maximum, a period of fixed speed movement and finally a deceleration period where the step rate decreases down to zero.

If desired, any set of the digital inputs can be used as end-stop signals, and these will stop the motor if driven. The clockwise and anti-clockwise end-stops need not be the same, and it is possible to support extra digital inputs which will stop the motor at any time (for emergency stops, etc).

This module holds the current motor position to 32-bits, in other words as a number in the range 0 to 4,294,967,296. The motor move commands can either take the form of move to a given (32-bit) position, or move in a given direction by a given (32-bit) distance.

It is possible to set a motor time-out period so that after every motor-move the MIO will wait for the time-out period to elapse before ceasing to drive the motor enabled output, which can be used to tell the external driver module to ‘power-down’.

This can be used to save power in applications not requiring any holding torque. Alternatively this signal can be used to drive a ‘high-power’ enable signal to the module, this could then be used to drive the motor while the motor was being pulsed, then removed after it had ceased to move.

See Stepping Motor (4-Phase) for setup and usage information.

4.6. Pulse Width Modulator (PWM)

This module generates an output pulse at a user-determined frequency with a user-determined mark-space ratio. All 4 outputs are driven with the same pulse at exactly the same time, this allows them to be connected in parallel in order to sink higher currents than a single output.

There are no additional setup items for the PWM module.

4.6.1. Usage

The PWM module defines two resources that are used by the application program to control the pulse width modulated outputs.

- **PWM Period** This sets the overall period of the waveform. Its value can range from 2 up to 32767. The lower the number, the greater the frequency of the output.
- **PWM Ontime** This sets the amount of time the output will be on out of the PWM period. The value can range from zero (output always off) to the value of the PWM period (output always on). Values in between will result in a pulsed waveform whose mark space ratio is determined by the ratio of the PWM ontime to the PWM Period settings.

4.7. Frequency Counter

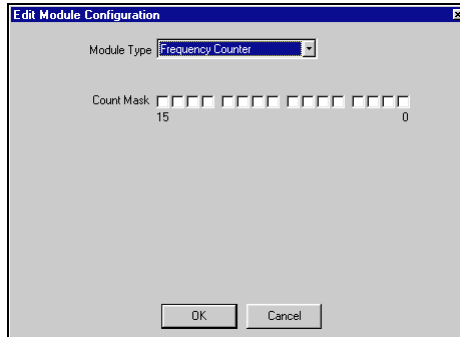
This module counts (to 16-bits) the number of changes detected by a user-specified set of digital inputs. In addition, it will also report the number of changes that occurred during a predefined period.

It is possible and even occasionally useful to select more than one input in this mask, the result of this is that they are all logically OR'ed together before being considered by this frequency counter module.

Since the count value is 16-bit, it can only hold values up to 65535, the next pulse after it has reached this maximum value will reset it to 0. Since the MIO cannot count much faster than 2000 pulses per second, it will take the MIO at least 30 seconds before this value overflows. It is the responsibility of the user's application to notice that the count has overflowed (usually by seeing the top-bit return to 0) and to take appropriate action.

4.7.1. Setup

The Frequency Counter module setup menu is as follows



The Count Mask check boxes determine which of the digital inputs are used to trigger the module.

4.7.2. Usage

The frequency counter module defines extra resources that allow application programs to setup and to read status from the module.

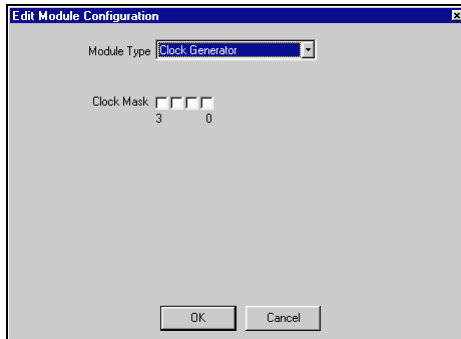
- **Count Total** This returns the total number of valid input transitions that have been detected by the module.
- **Count Period** This sets the time interval over which frequency measurements take place in steps of 200 microseconds. The default value of 5000 will allow the number of transitions per second to be measured by smaller or larger sampling periods can be selected as required.
- **Count Freq** This returns the number of input transitions detected over the duration of the last Count Period. This value is updated every count period.
- **Count Mask** This sets which combination of digital inputs will trigger the count. Bit 0 selects digital input 0, bit 1 selects digital input 1, etc.

4.8. Clock Generator

The clock generator allows the generation of up to 4 predefined high frequency clocks of 15 KHz, 7.5 KHz, 3.75 KHz and 1.875 KHz on the four outputs associated to the module.

4.8.1 Setup

This module setup menu is as follows



- Check Box 0 enables the 15 KHz output.
- Check Box 1 enables the 7.5 KHz output
- Check Box 2 enables the 3.75 KHz output
- Check Box 3 enables the 1.875 KHz output

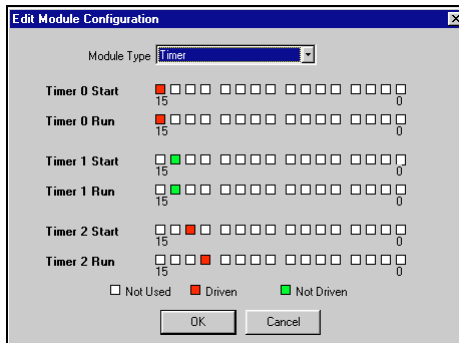
The module also defines a clock mask resource that allows enabling and disabling of each of the frequency outputs.

4.9. Timer

This module provides three independent, identical timers that can be used to measure the time that elapses after a specified start event and whilst a specified condition exists.

4.9.1 Setup

This module setup menu is as follows



Each timer defines a start condition and a run condition. Once the timer is ‘Armed’ by writing to the State register, then on detection of the ‘start’ condition and providing the ‘run’ condition is also satisfied, an internal 16-bit counter will be incremented every millisecond. When the ‘run’ condition is no longer ‘true’, the timer halts and the counter holds the elapsed time between the two events.

Each condition specifies which of the digital inputs are used and whether or not the true or complement of the input is used. In the example shown, Timer 0 will start when Digital input 15 is driven and will run until Digital input 15 is no longer driven. Timer 1 will start when Digital input 14 is not driven and will count until it is driven. Timer 2 will start when Digital input 13 is driven and runs until Digital input 12 is driven.

4.9.2. Usage

The timer module defines extra resources that allow applications programs to setup and to read status from the module.

For each of the timers it defines

- **TMR ? State.** This is used to ‘Arm’ the timer as well as returning its current operational status. Its valid values are
 - 0 Timer Stopped
 - 1 Timer Armed
 - 2 Timer Running
- **TMR ? Time.** Once the timer has been triggered, this is used to read the time between the ‘start’ event and the end of the ‘run’ event.
- **TMR ? Start Polarity.** This is used to determine the active polarities of each of the digital inputs for the ‘start’ condition. Bit 0 is used to control Input 0, etc. A ‘0’ in the control bit will cause the active state of that input to be when it is driven, a ‘1’ will set the active state to be when it is not driven.
- **TMR ? Start Mask.** This is used to determine which of the digital inputs are used to generate the ‘start’ event. A ‘1’ in the control bit will enable the input to control the ‘start’ event, a ‘0’ will cause the input to be ignored.
- **TMR ? Run Polarity.** This is used to determine the input polarities for the ‘run’ condition.
- **TMR ? Run Mask.** This is used to determine which of the digital inputs control the ‘run’ condition.

Before a timer can be triggered, it must first be armed by writing the value ‘1’ to its State. As soon as a ‘start’ event occurs, the value read back will change from a ‘1’ to a ‘2’ to indicate that it is now counting. When the ‘run’ condition ends, the State changes to ‘0’. The Time value can now be read which will indicate the elapsed time in milliseconds between the ‘start’ event and the end of the ‘run’ event.

For more information on the timer module, please consult the hardware manual.

Chapter 5 : A More In-depth Discussion Of The ActiveX Control

Earlier on in this manual we demonstrated how to use the Multi-IO ActiveX control in a simple application. Now would be an appropriate time to discuss some of features in greater depth. In its simplest form the ActiveX control provides the interface between your application and the physical MIO modules.

Your application communicates with the ActiveX control and the ActiveX control communicates with the MIO modules.

Property Pages

The MIO ActiveX control like most other controls requires some specialised setting up if it is to perform in your application. The way you set these special settings is through **property** pages. These property pages are part of the ActiveX control and they allow you the developer to setup data in a user friendly way. The use of property pages isn't the only way the control can be setup but it is the easiest.

In most development environments property pages can be accessed by right clicking on the control and selecting the "properties" menu item.

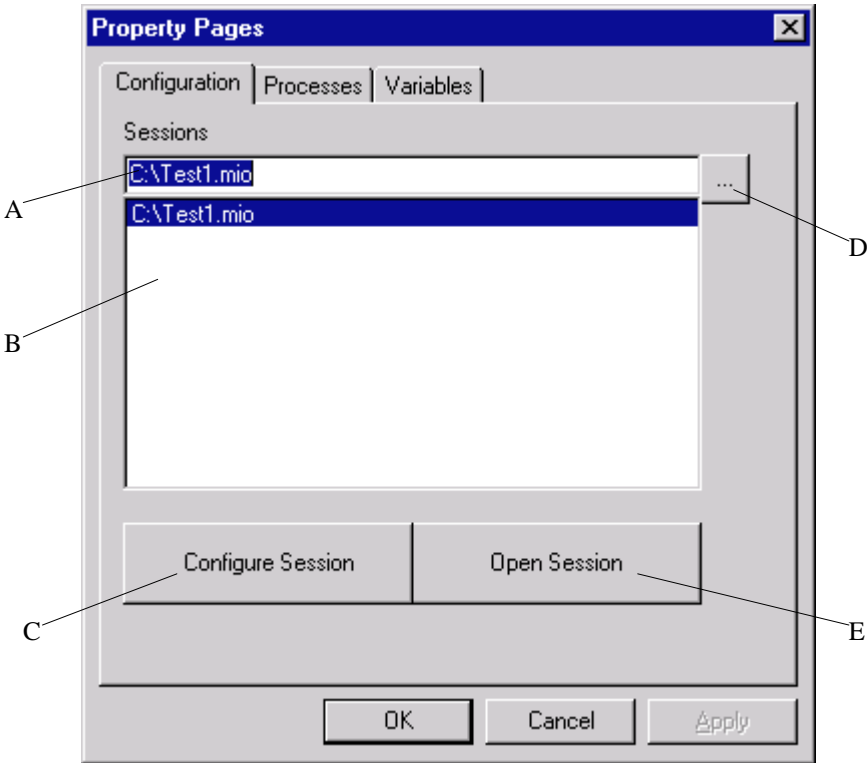
Setting up the ActiveX control

For the moment we will assume you have created a MIO ActiveX control in your project. Exactly how you do this depends on the development environment you are using.

At this stage the control knows nothing about the physical MIO modules which you have connected to your PC. This information is created by the MIO configuration utility and stored in a file which typically has a '.mio' file extension.

So the first thing we need to do is point the ActiveX control at this configuration file. To do this we need to be able to access the property pages, as this where this setting up has to be done. As mentioned earlier right clicking on the control usually displays a popup menu which contains the menu item "properties". Selecting the

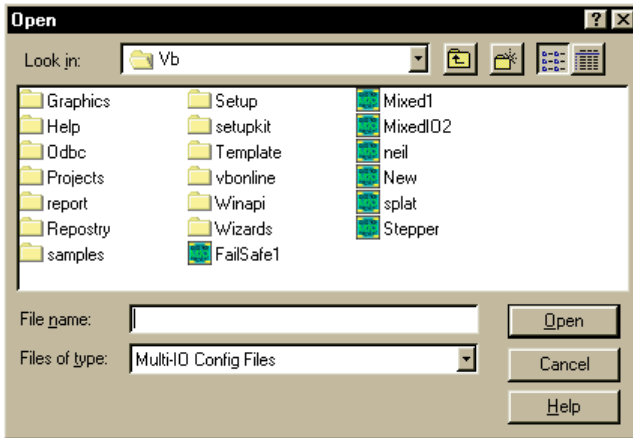
option invokes the property pages. Typically you see the page shown in the diagram below.



As you can see from the diagram there are 5 main areas of interest.

- Area A shows the configuration file currently in use.
- Area B lists the most recently used configuration files.
- Area C will allow you to configure/reconfigure or create a configuration file.
- Area D will allow you to select a new configuration file.
- Area E will allow you to tell the ActiveX control either to “open” the configuration file, which initialises and ‘connects’ to the MIO modules, or “close” the configuration file, which disconnects from the MIO modules.

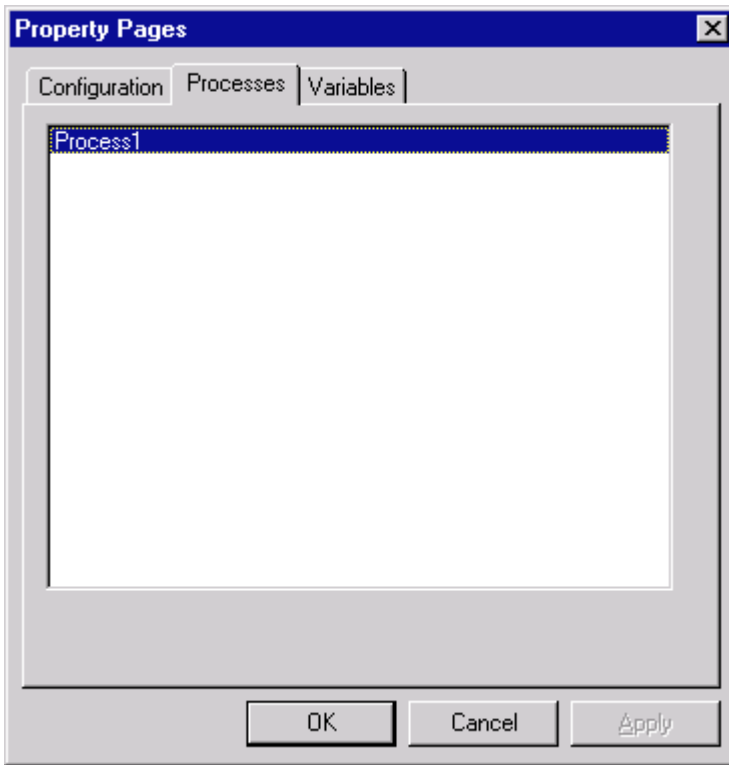
Clicking the “...” button shows the file selection dialogue box.



Select a configuration file then click on the “open” button to open the file.

When you open a configuration file the ActiveX control automatically parses the file, initialising its internal data structures, initialises the serial port and finally initialises the MIO modules. If a previous configuration file was open this will be closed before the new configuration is opened.

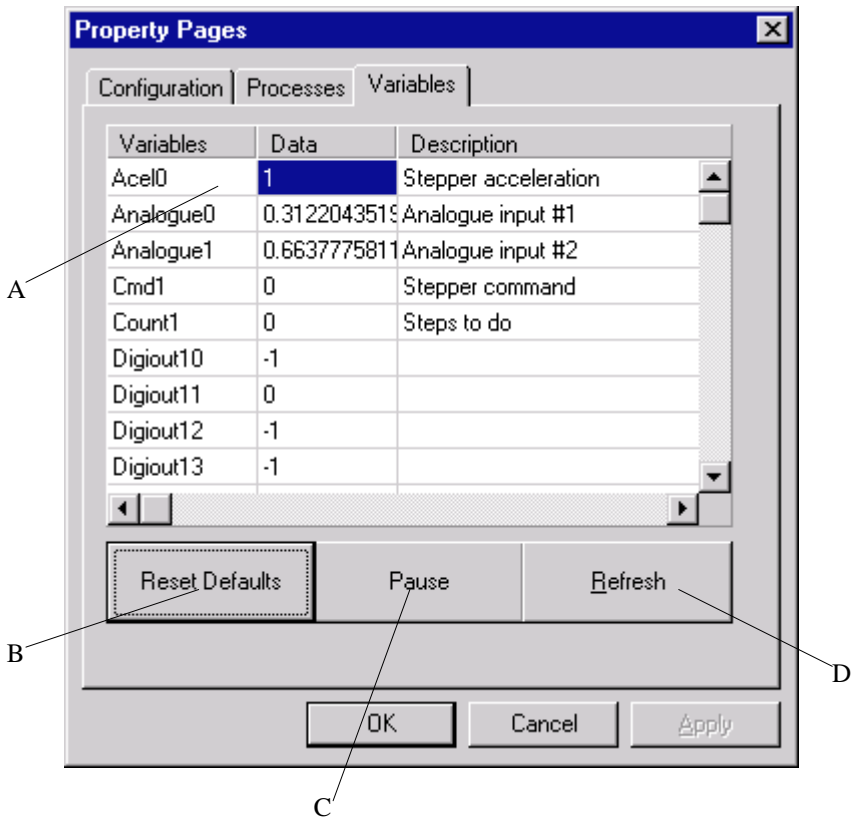
Clicking the ‘processes’ tab lists the available “**processes**” associated with the currently open configuration file. Clicking a process opens it, closing the previous process if required. If you click the same process again it will be closed and de-selected. Opening a process allows you access to the associated variables.



The 'variables' tab lists the currently active variables and shows their current values. If the configuration file is valid and the MIO has been configured correctly the values shown will be of 'live' data. These values will be updated on a 2 second basis.

Entering a value for a variable item will set the appropriate variable in the MIO module. This value will, from now on, become the default value for the variable. Each time you run your application the default variable values are sent to MIO module. To clear a previously assigned value enter 'def' into the data field.

NOTE: If you select a new process or new configuration file the default values will become invalid.



Reset Defaults - Clears any previously defined default values.

Pause - The data shown on this property page is 'live' data. These values are updated at 2 second intervals. To stop the data from being continuously updated click this button. Click again to re-enable updating.

Refresh - If you have stopped the data from being updated by clicking the 'Pause' button. The result of changing values will not be seen. Clicking this button reads data from the MIO module and shows the current values.

The ActiveX control is now configured all the data needed to initialise it has been setup. You will now be able to communicate with the MIO very easily from your application. Retrieving and sending data to the MIO units is discussed later on in this manual.

5.1. Setting up the ActiveX control (manually)

If you are not using Visual Basic setting up the ActiveX control is slightly different. Using Delphi, for instance, you're development cycle will be different.

Delphi and similar development environments keep the original design page active while running the application. This will cause problems with the ActiveX control because when you run your application you actually create a new page and a new ActiveX control that shares the same attributes as the control on the design page.

As mentioned earlier in this manual once a control opens a process no other control can later open the same process. To get around this problem you will need to do 2 things.

The first step is to make sure that the configuration file you have selected is not left open when you want to run your application. So, from the property pages, close the configuration file.

The second step involves a small amount of code to be entered into your program to initialise the MIO ActiveX control manually.

You will be required to tell the MIO ActiveX control which configuration file to use and which 'process' you want to attach to this control. You should do this during the initialisation of your application.

Unfortunately the downside of initialising the control using this method is that default values set from the property pages will no longer be valid. So if you need to set initial values you will required to write the code to do so.

The following code fragment (in Delphi's Pascal) gives an example. The code extract below introduces a number of new items that will be discussed in greater detail later on in this manual, but you should be able to work out what the code is doing without too much more explanation.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  { the lines below simply try to open
  a MIO configuration (session) file. If the session could
  be opened a 'process' is then made
  'active'. The ActiveX control then owns
  the resources associated with the process. Successful completion
  indicates the MIO can be 'talked' to. }
```

```
if MDSMIO1.OpenSessionFile('c:\test.mio') then
  MDSMIO1.ActivateProcess('Process1', true)
else
  ShowMessage('MIO couldn't be opened');
end;
```

This initialises the MIO ActiveX control, tells it which configuration file to use and activates a 'process'. You could actually expand the above code to test if the process could in fact be opened. This is the only setting up required.

If all goes well you will be able to communicate with the MIO module. If you do not initialise the MIO control correctly you will not be able access any data. Check that you have used a valid and a valid process name.

You can still test your configuration files from the property pages as described earlier in this manual.

Examples for development environments other than Visual Basic will be installed on to your hard disk along with the other development tools.

Chapter 6 : Data Types

6.1. Overview

This section briefly describes the data types you will encounter when using the ActiveX driver. Most of the data types discussed in this section are not specific to the MIO ActiveX control but are in fact part of the ActiveX interface standard. All ActiveX controls support these data types.

The need for standard data types is essential. For example, in the past, different programming languages have interpreted integer data in different ways. This is of no use when you are trying to create an application that can be used on many computer systems in many locations. The ActiveX standard is meant to be independent of computer type, locale and even the operating system.

6.2. Common Data Types

The following topics introduce various data types that you may or may not be familiar with. These data types are consistent across different versions of Windows and thus provide an amount of standardisation. Variables accessed through the MIO ActiveX driver all use these standard data types.

WordBool: describes a variable that is 16 bits long and simply denotes a true or false value. A 'true' value is any value other than 0. A 'false' value equals 0.

Integer: describes a signed 32 bit variable.

OleVariant: describes a variable which can be of any data type ie it could represent an integer or a string or an array, etc. An 'OleVariant' is a very convenient place holder for different types of data in a single package. This makes writing some routines easier due to the fact you do not have write routines to be able to cope with each possible data type.

WideString: a widestring by definition uses 2 bytes to represent each character instead of the usual 1. Primarily used to represent character sets with more than 256 characters. Widestrings have found use in ActiveX controls and OLE controls because of their need for locale independence.

The version of 'WideString' described here in some quarters is called an 'OleStr'. This differs from a simple 'wide' string in that the system allocates and controls the memory used by the string. This means that even if the provider of the string 'goes' away the calling routine will still be able to access the data without causing a memory violation. You do not need to worry about the memory allocation as this will automatically be catered for by the development environment you are using.

Long: a 32 bit signed integer value.

Enumerated Data Types

An enumeration data type is used to provide mnemonic identifiers for a set of constant integer values. In your application you can use these identifiers instead of having to remember particular numbers. This makes your application easier to read and therefore easier to follow and therefore easier to debug.

Enumeration types are simply provided for the convenience of the programmer as an aid to writing software. You don't have to use them (you can use the actual values instead) but they make life harder than it need be.

The MIO ActiveX driver defines a number of enumeration types for your use. Enumeration types defined here are grouped under a general name that indicates their use. The first item in the group has a value starting at 0, incrementing thereafter for each additional item. For example

```
MyErrorType {
    firstItem,      // has the value 0
    secondItem,    // has the value 1
    thirdItem      // has the value 2
}
```

in your application you could have the statement

```
if GetData(firstItem) = 0 then ...
```

The following enumerations are defined by the MIO ActiveX control.

TxStatus - indicates general status information

```
TxStatus {
    tsOffline,      // MIO module is currently offline
```

```
tsOnline,      // MIO module is online and communicating
tsError       // MIO configuration could not be opened
}
```

TxError - indicates the current error status

```
TxError {
    teOk,          // no error occurred
    tePortFail,   // The communication port couldn't be opened
    teTimeout,    // The MIO module failed to respond in time. Many
                // errors of this type indicate problematic
                // communications with the MIO module
    teReset,      // The MIO module has been reset
    teIllegalAccess, // You tried to access a variable not 'owned' by your
                // process
    teBadComms,   // The MIO module couldn't acknowledge the last
                // message. Many errors of this type indicate
                // problematic communications with MIO module.
    teCreateFail, // The configuration file couldn't be opened due to a
                // system error
    teIllegalVar  // You tried to access a variable which does not exist
}
```

TxMotorCmds - stepper motor control commands

```
TxMotorCmds {
    tmStop,      // stop the stepper motor
    tmMove,      // move the stepper motor a number of steps in the
                // current direction
    tmRelMove,   // move the stepper motor a number of steps relative to
                // the current position
    tmAbsMove,   // move the stepper motor to this absolute position
    tmResetPos,  // reset stepper position counters. The current stepper
                // position will become the origin from which all
                // subsequent steps will be calculated.
    tmSetPhase   // move the stepper to an absolute phase position
}
```

TxMotorState - indicates the current status of a stepper motor

```
TxMotorState {  
    txRunning,      // the stepper is executing the current step command  
    txCompleted,   // the stepper has completed the last step command  
    txLimitReached, // the stepper has hit a limit switch  
    txStopped      // the stepper has been stopped  
}
```

Chapter 7 : Properties

7.1. Properties

A property has an assigned data type for example string, or perhaps a 32 bit integer. It is normally used to set an object attribute, for instance an object's colour is often set through a 'property'. Applications like Visual Basic make extensive use of properties to make configuring objects much easier.

Fundamentally properties invoke 2 different methods calls - a GET method to retrieve data and a SET method to set data. These methods calls are normally hidden from the developer by the user interface that implements the GET\SET methods automatically.

For example, the Enabled property, detailed below, returns\expects a 'Boolean' data type. If in you write the statement:-

```
if MDSMIO1.Enabled then ...
```

This statement invokes the GET method to retrieve the data. Similarly, writing the statement:-

```
MDSMIO1.Enabled:= true;
```

will invoke the SET method. This example outlines another use of properties in that the same command can be used to access data. Your development environment automatically sorts out which method should be called depending on the context.

Not all development environments support 'properties' but even for these environments there are function calls available which will allow you to do the same tasks.

7.2. Enabled Property

Description:

Enables/disables the control. In fact this merely enables or prevents the reporting of timer events if the internal timer has been set-up.

Data Type: Boolean

Sample Code:

Delphi: MDSMIO1.Enabled:= true;

CBuilder: MDSMIO1->Enabled = true;

Visual Basic: MDSMIO1.Enabled = true

Remarks:

None

7.3. Interval Property

Description:

Sets the timer interval after which a 'timer event' will be generated and 'interval' times after.

Data Type: Long

Sample Code:

Delphi: MDSMIO1.Interval:= 1000;

CBuilder: MDSMIO1->Interval = 1000;

Visual Basic: MDSMIO1.Interval = 1000

Remarks:

Setting the interval to 0 disables the event. If the control is disabled then no timer event is generated but the timer interval is still valid.

7.4. Process Property

Description:

Retrieves the 'process' name currently associated with this control. A process is owned by the control that opens it. Only the control that **owns** the process has access to its variables. Setting the process property automatically tries to open the process. If it has already been opened by another control nothing will happen.

Data Type: WideString

Sample Code:

Delphi: MDSMIO1.Process:= 'Process1';

CBuilder: MDSMIO1->Process = WideString("Process1");

Visual Basic: MDSMIO1.Process = "Process1"

Remarks:

If the control already has an open process the current process is closed before the new one is opened.

7.5. SessionName Property

Description:

Sets the pathname of the session configuration file to be used. The session has to be later opened with a call to 'OpenMIOSession'.

Data Type: WideString

Sample Code:

Delphi: MDSMIO1.SessionName:= 'c:\config1.mio';

CBuilder: MDSMIO1->SessionName = WideString("c:\\config1.mio");

Visual Basic: MDSMIO1.SessionName = "c:\\config1.mio"

Remarks:

The full pathname must be entered and it must be valid.

7.6. Status Property

Description:

Returns the current status of a MIO session. Data Type: TxStatus

Sample Code:

Delphi: if MDSMIO1.Status = tsOnline then ...

CBuilder: if (MDSMIO1->Status == tsOnline) ...

Visual Basic: if MDSMIO1.Status = tsOnline then ...

Remarks:

If you try to go online by setting the 'status' property to 'tsOnline' the MIO control will show a message if you specify an invalid session name or if a session file is already open.

Chapter 8 : Methods

8.1. Methods

Method (or function) calls are routines that can have a number of parameters passed to them and do not share the restrictions of properties. They may or may not return a result. For environments that do not support ‘properties’ there are additional method calls available.

Due to the way ActiveX controls are imported into certain applications we cannot guarantee that method declarations will be exactly as stated here.

The reason for this is to do with how ActiveX controls are imported. All development environments use the ‘type library’ to generate custom code which your application calls.

For example if you make a call to the routine ‘OpenSessionFile’ you do not call this routine directly instead you call an interface routine generated by your development application which makes the call on your behalf.

A ‘type library’ holds information about the available functions and properties within the ActiveX control.

8.2. ActivateProcess

Declaration: ActivateProcess(ProcessName: WideString: Activate: WordBool)

Returns: WordBool

‘true’ if the process could be opened, ‘false’ if not.

Description: Activate\deactivate a process.

Sample Code:

Delphi: MDSMIO1.ActivateProcess(‘Process1’, true);

CBuilder: MDSMIO1->ActivateProcess(WideString(“Process1”), true);

Visual Basic: MDSMIO1.ActivateProcess “Process1”, true

Remarks:

A process once opened is then owned by the control. No other control can have access to the process.

8.3. CloseMIOSession

Declaration: CloseMIOSession

Returns: WordBool

‘true’ if the session could be closed without error, ‘false’ otherwise

Description:

This function closes the currently open configuration file. All communication with the Multi-IO module is broken.

Sample Code:

Delphi: MDSMIO1.CloseMIOSession;

CBuilder: MDSMIO1->CloseMIOSession();

Visual Basic: MDSMIO1.CloseMIOSession

Remarks:

After closing the session all further communication is invalid. All variables are inaccessible.

8.4. IsProcessActive

Declaration: IsProcessActive(ProcessName: WideString)

Returns: WordBool

‘true’ if process is active. Returns ‘false’ otherwise.

Description:

This routine returns the status of the given process.

Sample Code:

Delphi: if MDSMIO1.IsProcessActive(‘Process1’) then ...

CBuilder: if (MDSMIO1.IsProcessActive(WideString(“Process1”))) ...

Visual Basic: if MDSMIO1.IsProcessActive(“Process1”) then ...

Remarks:

None.

8.5. GetProcesses

Declaration: GetProcesses

Returns: OleVariant

Description:

This routine returns the names of all the processes defined in the current session file. The names are returned as an OleVariant array.

Sample Code:

Delphi:

```
procedure ListProcesses;
var
  Data: OleVariant;
  i: Integer;
begin
  Listbox1.Clear;

  { get a list of all the processes in this session }
  Data:= MDSMIO1.GetProcesses;
  { add all the processes to the listbox }
  for i:= VarArrayLowBound(Data) to VarArrayHighBound(Data) do
    Listbox1.Add(Data(i));
end;
```

CBuilder:

```
void ListProcesses(void)
{
Listbox1->Clear;
// get a list of all the processes in this session
Variant Data = MDSMIO1->GetProcesses();
// add all the processes to the list box
for (int i = VarArrayLowBound(Data, 1); i <=
    VarArrayHighBound(Data, 1); i++)
{
Listbox1->Add(Data.GetElement(i));
}
}
```

Visual Basic:

```
Private Sub ListProcesses()
List1.Clear

'list all the processes in this session
Data = MDSMIO1.GetProcesses
'add all the items to the list box
For i = LBound(Data) to UBound(Data)
List1.AddItem(Data(i))
Next
End Sub
```

Remarks:

This routine returns a variant which contains an array of strings. If no processes are available this routine returns an 'empty' variant.

8.6. GetVariable

Declaration: GetVariable(VarName: WideString)

Returns: OleVariant

Description:

This routine returns the data of the given variable name.

Sample Code:

Delphi: Data:= MDSMIO1.GetVariable('Analogue1');

CBuilder: Data = MDSMIO1->GetVariable(WideString("Analogue1"));

Visual Basic: Data = MDSMIO1.GetVariable("Analogue1")

Remarks:

The type of the data returned depends on the configuration of the Multi-IO module. The data could be an 'integer', a 'float', a 'boolean', etc...

8.7. GetVariables

Declaration: GetVariables

Returns: OleVariant

Description:

This routine returns all the names of the variables accessible from the current process.

Sample Code:

Delphi:

```
procedure ListVariables;
var
  Data: OleVariant;
  i: Integer;
begin
  Listbox1.Clear;

  { get a list of all available variables }
  Data:= MDSMIO1.GetVariables;

  { add all the processes to the listbox }
  for i:= VarArrayLowBound(Data) to VarArrayHighBound(Data) do
    Listbox1.Add(Data(i));
end;
```

CBuilder:

```
void ListVariables(void)
{
Listbox1->Clear;
// get list of all available variables
Variant Data = MDSMIO1->GetVariables();
// add all the processes to the list box
for (int i = VarArrayLowBound(Data, 1); i <=
    VarArrayHighBound(Data, 1); i++)
{
Listbox1->Add(Data.GetElement(i));
}
}
```

Visual Basic:

```
Private Sub ListVariables()
List1.Clear

' get list off all variables
Data = MDSMIO1.GetVariables

'add all the items to the list box
For i = LBound(Data) to UBound(Data)
List1.AddItem(Data(i))
Next
End Sub
```

Remarks:

The returned data is a variant array of strings. If no variables are accessible the returned value is an 'empty' variant.

8.8. OpenMIOSession

Declaration: OpenMIOSession

Returns: WordBool

Description:

This routine will try to open the current configuration session file (set by the property 'SessionName' discussed earlier).

Sample Code:

Delphi: MDSMIO1.OpenMIOSession;

CBuilder: MDSMIO1->OpenMIOSession();

Visual Basic: MDSMIO1.OpenMIOSession

Remarks:

The configuration session file needs to have been set via the 'SessionName' property before this routine is called.

8.9. *OpenSessionFile*

Declaration: OpenSessionFile(ConfigFile: WideString)

Returns: WordBool

'true' if the configuration file could be opened, 'false' if not.

Description:

Open the given session configuration file.

Sample Code:

Delphi: MDSMIO1.OpenSessionFile('c:\config.mio');

CBuilder: MDSMIO1->OpenSessionFile(WideString("c:\\config.mio"));

Visual Basic: MDSMIO1.OpenSessionFile("c:\config.mio")

Remarks:

The pathname name provided must be a valid pathname.

8.10. Additional Methods

8.10.1. Quick Note

These additional methods are provided for use by development environments that do not support object properties. You should only use these methods if this is the case.

8.10.2. GetEnabled

Declaration: GetEnabled()

Returns: WordBool

‘true’ if enabled, ‘false’ if disabled

Description:

See if the control is enabled.

8.10.3. GetInterval

Declaration: GetInterval()

Returns: Long

Description:

Get the current timer interval value.

8.10.4. GetProcess

Declaration: GetProcess()

Returns: WideString

Description:

Get the active process attached to the current control.

8.10.5. GetSessionName

Declaration: GetSessionName()

Returns: WideString

Description:

Get the path name of the configuration file used by the current control.

8.10.6. GetStatus

Declaration: GetStatus()

Returns: TxStatus

The result indicates the current module status.

0 = module offline - MIO module connected

1 = module online - MIO module not connected

2 = error, module could not go online

Description:

Get the status of the connected MIO module.

8.10.7. SetEnabled

Declaration: SetEnabled(enable: WordBool)

Returns: None

Description:

Enables\disables the control depending on the value passed. '0' disables the control any other value enables the control.

8.10.8. SetInterval

Declaration: SetInterval(newValue: Long)

Returns: None

Description:

Sets the control's timer interval. Each control has its own timer and can be set independently of any other timers you may have setup. Your timer routine will be called every 'newValue' milliseconds.

8.10.9. SetProcess

Declaration: SetProcess(newProcess: WideString)

Returns: None

Description:

This function sets the “process” associated with the current control. If the control already has a process associated with it, that process is closed before the new one is opened.

8.10.10. SetSessionName

Declaration: SetSessionName(newSession: WideString)

Returns: None

Description:

This function sets the configuration file to be used by this control. Only 1 configuration file can be opened at a time but it is possible for the same configuration file to be opened by more than 1 application.

8.10.11. SetStatus

Declaration: SetStatus(newStatus: TxStatus)

Returns: None

Description:

Call this return to try to connect to a MIO module. You must have set a valid session configuration filename before calling this routine. The valid values which can be passed are:-

- 0 = disconnect from MIO module
- 1 = try to connect to MIO module

any other values are ignored.

Chapter 9 : Events

9.1. Overview

An event is a mechanism that allows you to link some code to some occurrence. Windows informs you of events through its use of messaging. You can either react to the event or choose to ignore it. The MIO ActiveX control can only generate 2 types of events - a timer event and an error event. How you deal with these events is left entirely up to you.

9.2. OnMIOError

Declaration: OnMIOError(BoardAddr: Integer; ErrorFlag: TxError;
Msg: WideString)

Description:

This event is called by the Multi-IO ActiveX control when an error occurs. The 'BoardAddr' refers to the actual number assigned to the Multi-IO board. The 'ErrorFlag' indicates the type of error and the 'Msg' is an optional error string. 'ErrorFlag' can be anyone of the following:-

- teOk - no error.
- tePortFail - the port specified in the configuration file couldn't be opened.
- teTimeout - the reply to the last message timed out.
- teReset - the Multi-IO module has reset.
- teIllegalAccess - you have tried to access a variable or process to which you have no rights.
- teBadComms - communications to the Multi-IO module have become inconsistent.
- teCreateFail - The session couldn't be opened because of a system error.
- teIllegalVar - you have tried to access a variable which does not exist.

Sample Code:

Delphi:

```
procedure TMDSMIO1OnMIOError(BoardAddress: Long;
```

```
ErrorFlag: TXError; Msg: WideString)
begin
case ErrorFlag of
teTimeout:
    inc(Timeouts);
teReset:
    inc(Resets);
end;
end;
```

CBuilder:

```
void TMDSMIO1::OnMIOError(long BoadrAddress, TxError ErrorFlag,
    WideString Msg)
{
switch (ErrorFlag)
{
case teTimeout:
    ++Timeouts;
    break;

case teReset:
    ++Resets;
    break;
}
}
```

Visual Basic:

```
Private Sub MDSMIO1_OnMIOError(ByVal BoardAddress As Long,
    ByVal ErrorFlag As MDSMIOControl.TxError, ByVal Msg As String)
Select Case ErrorFlag
Case teTimeout
    Timeouts = Timeouts + 1
Case teResets
    Resets = Resets + 1
End Select
End Sub
```

Remarks:

You shouldn't take too long to process an error message in case a reset occurs. The Multi-IO modules send a reset message immediately after power up, this message is sent once only.

9.3. OnMIOTimer

Declaration: OnMIOTimer

Description:

This routine is signalled by the Multi-IO ActiveX control when the timer interval has expired.

Sample Code:

Delphi:

```
procedure TForm1.MDSMIO1MIOTimer(Sender: TObject);
begin
  { show the current value of this variable }
  Label1.Caption:= MDSMIO1.GetVariable("analogue1");
end;
```

CBuilder:

```
void __fastcall TForm1::MDSMIO1MIOTimer(TObject *Sender)
{
  // show the current value of this variable
  Label1->Caption =
  Variant(MDSMIO1>GetVariable(WideString("analogue1")));
}
```

Visual Basic:

```
Private Sub MDSMIO1_OnMIOTimer()
'show the current value of the variable
Label.Caption = MDSMIO1.GetVariable("analogue1")
End Sub
```

Remarks:

Taking excessive amounts of time processing this event could have an impact on the overall performance of the communications with Multi-IO module.

Chapter 10 : Controlling Stepper Motors

10.1. Overview

Controlling a stepper motor is an important and very useful function of the MIO module. You have to setup stepper control from the 'MIOConfig' utility (please see the 'MIOConfig' manual for more information). Having setup the MIO unit to control a stepper motor, controlling the motor from your application is simply a case of setting up some variables.

10.2. Controlling a Stepper Motor

Assuming you have correctly setup the MIO module to control a stepper motor and connected a stepper motor to DO0 – DO3 (Slot 0). Controlling the motor from within your application becomes a simple matter of setting variables.

The sample code below (in Pascal for clarity) assumes 'Slot 0' has been configured for stepper control. The variables 'StepCommand' and 'StepCount' have been assigned to 'Slot_0_Step_Count' and 'Slot_0_Step_Cmd' respectively from the 'MIOConfig' utility program.

```
MDSMIO1.SetVariable("StepCount", 1000);  
MDSMIO1.SetVariable("StepCommand", tmRelMove);
```

When you run your application and you execute the above commands the stepper motor will move a 1000 steps from its current position.

It should be noted that the step count should be set before the step command is issued. The stepper command variable will be set to 0 when the command has been executed.

Chapter 11 : Networking the MIO Modules

From a programming point of view the MIO modules should be viewed as a shared system resource. You should consider the communication with the MIO modules as a network where you have a single server and a number of clients. Only the server actually does the communicating with modules, the clients merely access locations within a shared memory space made available to them by the server.

The server polls the MIO modules on a regular basis and updates any data which needs to be refreshed.

An application becomes a server simply by being executed first, applications loaded later become clients. You do not have to specify a server directly. The server itself will not run slower because it is the server. So there is no need for you to create a specific 'server' application.

As stated earlier, the server is the only application capable of communicating with the MIO modules directly, and so it follows that the server is the only application capable of handling communication errors. This is true but all applications will be notified of the errors. So when writing multiple applications you should allow them all to deal with all possible errors and simply filter out the errors from MIO devices that a particular application doesn't have access to.

Due to the 'shared' design structure you should be careful when reading and writing variables from multiple applications. No provision is made for access synchronisation to variables. So be aware that writing to the same variables from different applications could invalidate the data and introduce errors unless you make some sort of allowance for this possibility.

Chapter 12 : Example Programs

12.1. Examples

Various examples are included with this package. They are available for the most popular development environments and give an idea of how to communicate with the ActiveX control and a Multi-IO module.

All the examples given here are for demonstration use only and assume you have a working knowledge of the development environment you are using. If you need any information on connection details or require further information please refer to the Technical manual.

To use the examples you will need to run\compile the version for the supported environments and setup the MIO hardware as necessary.

These examples have been designed and tested using Visual Basic 5, Borland Delphi 3 and Borland CBuilder 3.

12.2. Example #1

Example #1 demonstrates reading the analogue inputs. No additional equipment is needed to use this example. Simply running your fingers along the edge of pins AI0-AI7 should be enough to provide varying data that you will see on your screen.

12.3. Example #2

Example #2 demonstrates using the digital outputs. You will need some LEDs or at least something which can show an output being driven. Connect the LEDs to digital outputs DO0 - DO7.

12.4. Example #3

Example #3 demonstrates using the analogue outputs AO0-AO7 and the analogue inputs AI0-AI7. You will be required to connect the outputs to the inputs. As you vary the voltages of the outputs the voltages at the inputs should vary accordingly.

12.5. Example #4

Example #4 demonstrates using the analogue outputs AO0-AO7 to control the digital inputs DI0-DI7. A digital 'true' state is achieved when the voltage rises above 2.5V. You may find, when running this example, that the state of a digital input will change at lower values than the value given but in practice these 'observed' values should not be relied upon.

12.6. Example #5

Example #5 demonstrates using a 4-phase stepper motor. Connect your stepper motor to DO0-DO3. You should be able to control the motor from this example program. If you do not have a stepper motor available an array of LEDs would be suitable for this demonstration.

12.7. Example #6

Example #6 illustrates trapping and dealing with errors signalled by the MIO ActiveX driver.

12.8. Example #7

Example #7 demonstrates the pulse width modulation capabilities of the MIO modules. This example uses digital outputs DO0-DO7. Connecting LEDs to the outputs will allow you to view the changes as the period and ontime variables are manipulated.

12.9. Example #8

Example #8 illustrates using multiple ActiveX controls in the same application. Each form contains its own control that links to a particular process. Each process has a specific task.

12.10. Example #9

Example #9 demonstrates creating multiple applications which use different processes within a session. The main program executes the sub-applications example9a.exe, example9b.exe, example9c.exe and example9d.exe. These sub-programs must be created before the main application is run.

12.11. Example #10

Example #10 illustrates the use of the timer module. The application waits for an input on one of the digital inputs DI0-DI7. When a digital input has a 'true' status the timer will be triggered. The timer value on the screen will increment in accordance with the length of time the input remains 'true'.

Index

A	
Acceleration	<i>See</i> Stepping Motors
ActivateProcess	41
ActiveX	4
B	
board	
resources	16
Board	
Setup	14
boards	4, 6, 14, 15, 17, 18, 19
C	
clients	4
Clock Frequencies	<i>See</i> Clock Generator
Clock Generator	29
CloseMIOSession	41
Communicating with the Multi-IO Modules	4
communication	4
communication errors	4
configuration file	4
configuration program	6
configuring	6
Connecting to a Multi-IO Module	4
Count Freq	<i>See</i> Frequency Counter
Count Mask	<i>See</i> Frequency Counter
Count Period	<i>See</i> Frequency Counter
D	
data types	36
Digital Inputs	22
E	
Enumerated Data Types	36
errors	4
F	
Frequency Counter	28
Count Freq	29
Count Mask	29
Count Period	29
Count Total	<i>See</i> Frequency Counter
G	
GetEnabled	45
GetInterval	45
GetProcess	45
GetProcesses	42
GetSessionName	45
GetStatus	45
GetVariable	43
GetVariables	43
I	
Integer	36
IsProcessActive	42
L	
Limit Neg	<i>See</i> Stepping Motors
Limit Plus	<i>See</i> Stepping Motots
M	
mark-space ration	<i>See</i> Pulse Width Modulator
Max Delay	<i>See</i> Stepping Motors
Min Delay	<i>See</i> Stepping Motors
MIO	4
MIOConfig.Exe	6
N	
network	4
O	
OleVariant	36
OnMIOError	47
OnMIOTimer	48
OpenMIOSession	44
OpenMIOSessionFile	44
P	
Power Down	<i>See</i> Stepping Motors
process4, 16, 17, 18, 19, 21, 32, 33, 34, 35, 37, 39, 41, 42, 43, 45, 46, 47, 48	
Programming the MIO Modules	4
properties	38
Pulse Width Modulator	28
PWM OnTime	28
PWM Period	28
PWM OnTime	<i>See</i> Pulse Width Modulator
PWM Period	<i>See</i> Pulse Width Modulator
R	
Raw_Inputs	22
Reset Position	<i>See</i> Stepping Motors
S	
server	4
server polls	4
session4, 13, 14, 16, 18, 19, 25, 32, 35, 40, 41, 42, 43, 44, 46, 47	
SetEnabled	46
SetInterval	46
SetProcess	46
SetSessionName	46
SetStatus	46
shared' design	4
Step Mode	<i>See</i> Stepping Motors
stepper motor	50
Stepping Motor s(Pulsed)	27
Stepping Motors	26
Acceleration	27
Limit Neg	27
Limit Plus	27
Max Delay	27
Min Delay	27
Power Down	27
Reset Position	27
Slot Status	27
Step Count	27
Step Mode	27
Step Position	27
Stepper Commands	
Step Command	27

Stepper Absolute Move	27	TMR_?_State.....	<i>See Timers</i>
Stepper Move.....	27	TMR_?_Run_Mask	<i>See Timers</i>
Stepper Relative Move	27	TMR_?_Run_Polarity	<i>See Timers</i>
Stop Stepping.....	27	TMR_?_Start_Mask	<i>See Timers</i>
Stepper Control		TMR_?_Start_Polarity	<i>See Timers</i>
Acceleration.....	26	TMR_?_Time	<i>See Timers</i>
Limit Negative	26	tmRelMove	37
Limit Plus	26	tmResetPos	37
Max. Delay	26	tmSetPhase	37
Min Delay	26	tmStop	37
Power Down	26	tsError.....	37
Step Mode.....	26	tsOffline	37
synchronisation	4	tsOnline.....	37
T		txCompleted	37
teBadComms	37	TxError	37
teCreateFail.....	37	txLimitReached.....	37
tellegalAccess.....	37	TxMotorCmds	37
tellegalVar	37	TxMotorState.....	37
teOk	37	txRunning	37
tePortFail	37	TxStatus.....	37
teReset	37	txStopped.....	37
teTimeout.....	37	V	
Timers.....	30	Variables.....	22
TMR_?_Run_Mask	30	velocity profiling	<i>See Stepping Motors</i>
TMR_?_Run_Polarity	30	W	
TMR_?_Start_Mask	30	Watchdog mode.....	24
TMR_?_Start_Polarity.....	30	WideString.....	36
TMR_?_Time	30	WordBool.....	36
tmAbsMove	37		
tmMove.....	37		
