

# Application Note

**AEROFLEX**  
A passion for performance.

## 5800 Series- Digital I/O using the PUC

Reference: AN504-03



### Introduction

The Aeroflex Integrated Development Environment (AIDE) provides a comprehensive programming platform capable of controlling a vast array of devices in addition to the 5800 Automatic Test Equipment (ATE) hardware.

These notes describe how general purpose Digital Input/Output facilities on the Power and Utility Card (PUC) may be utilised to monitor, test and control devices on a Unit Under Test (UUT) from the AIDE in a .NET environment.

## Functionality

The AIDE provides facilities for carrying out Analog In-Circuit, Functional and Digital Tests within the .NET Framework. Within the AIDE both the high level functionality such as a RESISTOR test and the low level functionality such as a basic Analogue to Digital Conversion (ADC) measurement are made available through a series of .NET Assemblies with the general reference name structure of Aeroflex.ATE.AIDE.assembly. AIDE test programs also have access to particular hardware and software facilities via a number of system Built In Variables (see system documentation for further details).

With the \$Fixture built in variable, the user can access the DataPort, a group of 8 digital output lines on the PUC in order to write to external digital hardware and the SenseLines, a group of 8 digital input lines to read from external digital hardware.

These signals are used to control personality cards in the fixture, such as the CMUX card when CODA testing. When not employed for this purpose the user can make use of these signals in any number of ways to provide additional testing capabilities.

## Program Example – Programming a Serial EEPROM using the PUC

Create a new Program called, for example, AIDEdio

### Configuration Data

Expand the Configuration Data section of the program and if not already present, add the following .NET Assembly References from the .NET Global Assembly Cache (GAC) using the browse facilities provided:-

```
.NET Assembly Reference AssemblyFile:mscorlib.dll
.NET Assembly Reference AssemblyFile:System
```

If not already present, add the following .NET Namespace References:-

```
.NET Namespace Reference System
```

### Global Symbols

A number of decisions have to be made as to how to interface to the device. A typical example of a Serial EEPROM is the 93C86C. This comes in various package types with either 8 or 6 pin connections. Assuming the EEPROM has been configured as 2048 x 8-bit, we require to allocate each signal to either Power, Ground, Digital Input or Digital Output lines from the PUC J8 Connector and the User Fixture Power D-type connector. The PUC has 8 data output lines (DATA0\_H..DATA7\_H) and 8 inverted data input lines (SENSE0\_L..SENSE7\_L) on the J8 connector. The User Fixture Power D-Type connector has 4 fixed power outputs (5V @ 5A, 24V @ 2A, +15V @ 2A and -15V @ 2A). The allocation of pins might be done as follows.

Name	Function	PUC Signal	PUC/ PSU Connection
CS	Chip Select	DATA2_H	J8-B6
CLK	Serial Data Clock	DATA1_H	J8-C6
DI	Serial Data Input	DATA0_H	J8-D6
DO	Serial Data Output	SENSE0_L	J6-D7
VSS	Ground	GND	D-Type Pin A2
PE	Program Enable	Connected to Vcc	
ORG	Memory Configuration	Connected to Vss	
VCC	Power Supply	5V	D-Type Pin A1

The PE and ORG pins are directly wired to force the 8-bit configuration and permanently enable programming of the device.

Having specified the appropriate connections we can now define some Global Constants that may be used to simplify the programming and make the AIDE code more readable to the user.

Create new Global Constants of type Byte as follows.

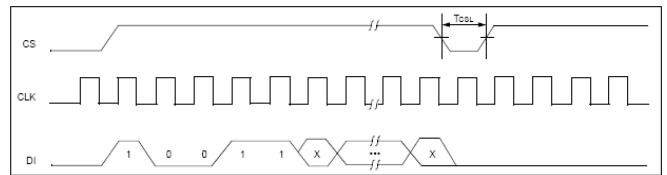
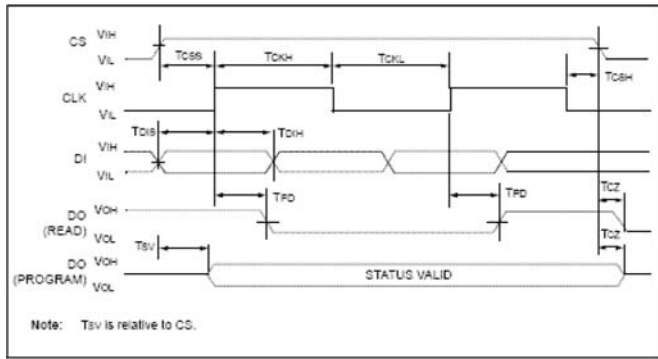
### Global Symbols

```
Constants
Constant [ Byte ] CS_H = 0b100
Constant [ Byte ] CS_L = 0b011
Constant [ Byte ] CLK_H = 0b010
Constant [ Byte ] CLK_L = 0b101
Constant [ Byte ] DI_H = 0b001
Constant [ Byte ] DI_L = 0b110
Constant [ Byte ] DO = 0b001
```

Note that the AIDE allows numeric values to be expressed in BINARY notation of the form 0bn...n where n is either a 0 or a 1.

The following table of the electrical characteristics of the 93C86C and the timing diagram illustrates that the read/write timing is not critical (specifies only a minimum requirement for most parameters).

93C86C Electrical characteristics for Vcc = 5V				
Symbol	Parameter (Vcc=5V)	Min	Max	Units
FCLK	Clock frequency	–	3	MHz
TCKH	Clock high time	200	–	ns
TCKL	Clock low time	100	–	ns
TCSS	Chip Select setup time	50	–	ns
TCSH	Chip Select hold time	0	–	ns
TCSL	Chip Select low time	250	–	ns
TDIS	Data input setup time	50	–	ns
TDIH	Data input hold time	50	–	ns
TPD	Data output delay time	–	100	ns
TCZ	Data output disable time	–	100	ns
TSV	Status valid time	–	200	ns
TWC	Program cycle time	–	2	ms
TEC	Program cycle time	–	6	ms
TWL	Program cycle time	–	15	ms



The above diagram shows the first CLK cycle is generated with CS and DI low, the second CLK cycle with CS and DI high to produce the SB condition. The next two CLK cycles generate the Opcode for EWEN (00). The next eleven CLK cycles generate the Address (A10.A0) where A10 = 1, A9 = 1 and A8.A0 are all "don't care" so the Address may be defined here as 0b1100000000. This gives a total of 14 CLK cycles as shown in the Instruction Set Table above.

Since every INSTRUCTION has the same format we can split the generation of the digital I/O pattern into three basic routines, namely – SendStart, SendOpCode and SendAddress.

### Main Method

Create a new Method called Main and reference it in the Program OnStart Property.

We can now create Evaluate statements that can be used to control writing and reading of an EEPROM.

### SendStart Method

Create a new Method called SendStart that takes no parameters and returns nothing, with the following Method Code :-

```
Method SendStart [] () Comment = "send start sequence to EEPROM"
```

Local Symbols

Method Code

```
Evaluate $Fixture.DataPort.Data = CLK_H
Evaluate $Fixture.DataPort.Data = [Byte] (DI_H & CLK_L)
Evaluate $Fixture.DataPort.Data = [Byte] ((DI_H | CS_H) & CLK_L)
Evaluate $Fixture.DataPort.Data = [Byte] (DI_H | CS_H | CLK_H)
```

This Method will be used to send the Start Bit (SB) condition for each instruction.

Note that a Byte Constant can be assigned directly to a Byte Variable as in \$Fixture.DataPort.Data = CLK\_H but that if any arithmetic operations are involved between Byte values, the resultant value is an Int which must be cast as a Byte when assigned to a Variable of type Byte, hence the [Byte] (DI\_H & CLK\_L) expression following the = in the second Evaluate statement.

### SendOpCode Method

Create a new Method called SendOpCode that takes a Byte code parameter and returns nothing, with the following Method Code :-

```
Method SendOpCode [] ( [Byte] code ) Comment = "send op-code sequence to EEPROM"
```

Local Symbols

Method Parameters

Method Parameter [ Byte ] code

The instruction set for the 93C86C device organised as 2048 x 8-bit is shown in the table2 below.

Instruction	SB	Op code	Address	Data In	Data Out	Req. CLK Cycles
READ	1	10	A10 A9 A8 A7 A6 A5 A4 A3 A2 A1 A0	–	D7–D0	22
EWEN	1	00	1 1 X X X X X X X X X	–	HIGH-Z	14
ERASE	1	11	A10 A9 A8 A7 A6 A5 A4 A3 A2 A1 A0	–	(RDY/BSY)	14
ERAL	1	00	1 0 X X X X X X X X X	–	(RDY/BSY)	14
WRITE	1	01	A10 A9 A8 A7 A6 A5 A4 A3 A2 A1 A0	D7–D0	(RDY/BSY)	22
WRAL	1	00	0 1 X X X X X X X X X	D7–D0	(RDY/BSY)	22
EWDS	1	00	0 0 X X X X X X X X X	–	HIGH-Z	14

The chip powers up in the ERASE/WRITE DISABLE (EWDS) state. All programming modes must be preceded by an ERASE/WRITE ENABLE (EWEN) instruction. Once the EWEN instruction is executed, programming remains enabled until an EWDS instruction is executed or Vcc is removed from the device.

Create new Global Constants of type Byte as follows.

Global Symbols

Constants

```
Constant [ Byte ] READ = 0b10
Constant [ Byte ] EWEN = 0b00
Constant [ Byte ] ERASE = 0b11
Constant [ Byte ] ERAL = 0b00
Constant [ Byte ] WRITE = 0b01
Constant [ Byte ] WRAL = 0b00
Constant [ Byte ] EWDS = 0b00
```

The Start bit (SB) is detected by the device if CS and DI are both high with respect to the positive edge of CLK for the first time. Before a Start condition is detected, CS, CLK, and DI may change in any combination (except to that of a Start condition), without resulting in any device operation (READ, WRITE, ERASE, EWEN, EWDS, ERAL or WRAL). As soon as CS is high, the device is no longer in Standby mode. An instruction following a Start condition will only be executed if the required opcode, address and data bits for any particular instruction are clocked in. The EWEN Timing Diagram is shown below.

## Method Code

```
Evaluate $Fixture.DataPort.Data = [ Byte] (((code
>> 1) & DI_H) | CS_H) Comment = "MSB + CS_H &
CLK_L"
Evaluate $Fixture.DataPort.Data = [ Byte] (((code
>> 1) & DI_H) | CS_H | CLK_H) Comment = "MSB +
CS_H + CLK_H"
Evaluate $Fixture.DataPort.Data = [ Byte] ((code &
DI_H) | CS_H) Comment = "LSB + CS_H & CLK_L"
Evaluate $Fixture.DataPort.Data = [ Byte] ((code &
DI_H) | CS_H | CLK_H) Comment = "LSB + CS_H +
CLK_H"
```

This Method will be used to send the Op-code for each instruction.

## SendAddress Method

Create a new Method called SendAddress that takes an Int addr parameter and returns nothing, with the following Method Code:-

```
Method SendAddress [] ( [ Int] addr )Comment =
"send address bits to EEPROM, MSB first"

Local Symbols

Method Parameters

Method Parameter [ Int ] addr

Variables

Variable [ Byte ] abit Comment "address bit"

Method Code

For ( Int i=10 ; i>=0 ; i=i-1 )
Evaluate abit = [ Byte] ((addr >> i) & DI_H)
Comment = "A[i] into DI bit position"
Evaluate $Fixture.DataPort.Data = [ Byte] (abit
| CS_H) Comment = "DI + CS_H & CLK_L"
Evaluate $Fixture.DataPort.Data = [ Byte] (abit
| CS_H | CLK_H) Comment = "DI + CS_H +
CLK_H"
```

This Method will be used to send the Address for each instruction.

## SendData Method

Create a new Method called SendData that takes a Byte data parameter and returns nothing, with the following Method Code:-

```
Method SendData [] ( [ Byte] data )Comment = "send
data bits to EEPROM, MSB first"

Local Symbols

Method Parameters

Method Parameter [ Byte ] data

Variables

Variable [ Byte ] dbit Comment "data bit"

Method Code

For ( Int i=7 ; i>=0 ; i=i-1 )
Evaluate dbit = [ Byte] ((data >> i) & DI_H)
Comment = "D[i] into DI bit position"
Evaluate $Fixture.DataPort.Data = [ Byte] (dbit
| CS_H) Comment = "DI + CS_H & CLK_L"
Evaluate $Fixture.DataPort.Data = [ Byte] (dbit
| CS_H | CLK_H) Comment = "DI + CS_H +
CLK_H"
```

This Method will be used to send the Data for each instruction that has data to be written to the EEPROM.

## WriteByte Method

Create a new Method called WriteByte that takes two parameters, the target address and the data byte to be written:-

```
Method WriteByte [] ( [ Int ] address,
[ Byte ] data )
```

### Local Symbols

#### Method Parameters

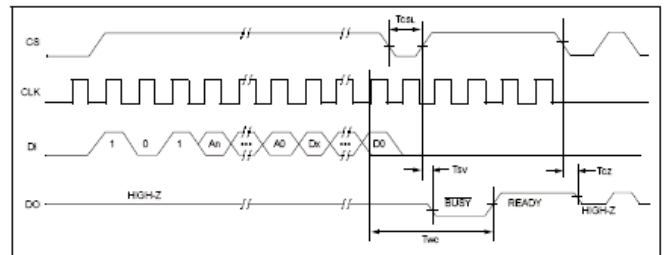
```
Method Parameter [ Int ] address Comment
= "the EPROM Address"
```

```
Method Parameter [ Byte ] data Comment =
"the EPROM data"
```

### Variables

```
Variable [ Int ] status = 0 Comment ="the
EPROM status"
```

This Method will be used to write a single byte at the specified EPROM address by implementing the WRITE TIMING diagram shown below.



We now implement the WriteByte Method by adding the following:-

```
Comment "send write enable"
Evaluate SendStart()
Evaluate SendOpCode(EWEN)
Evaluate SendAddress(0b1100000000)
Evaluate $Fixture.DataPort.Data = CS_H
Evaluate $Fixture.DataPort.Data = [ Byte] (CS_L &
CLK_L & DI_L) Comment = "End of Instruction
CS_L, CLK_L, DI_L"
Comment "do write"
Evaluate SendStart()
Evaluate SendOpCode(WRITE)
Evaluate SendAddress(address)
Evaluate SendData(data)
Evaluate $Fixture.DataPort.Data = CS_H
Evaluate $Fixture.DataPort.Data = [ Byte] (CS_L &
CLK_L & DI_L) Comment = "End of Instruction
CS_L, CLK_L, DI_L"
Evaluate $System.Wait(1us) Comment = "delay
until Tcs1 expires"
Evaluate $Fixture.DataPort.Data = CS_H Comment =
"set CS_H to strobe out status"
Evaluate $System.Wait(1us) Comment = "delay
until Tsv expires and status is valid"
While ( status == 0 ) Comment = "wait until not
busy"
Evaluate status = ($Fixture.SenseLines & DO) ^
DO Comment = "SenseLines are INVERTED so flip
DO, the status bit"
Evaluate $Fixture.DataPort.Data = [ Byte] (CS_L &
CLK_L & DI_L) Comment = "End of Instruction
CS_L, CLK_L, DI_L"
```

When executed, the above code sends write enable, then the target address and the data byte and finally checks the status to see that the write cycle has completed successfully.

## Program Execution

The PUC data output lines remain in a floating HIGH-Z state until an enable command is executed. This action also drives the SWITCH\_L line LOW on J8-E5 which might be used to enable any output drivers deemed necessary to provide signal buffering.

To Execute an EEPROM Write test, add the following to the Main Method:-

```
Evaluate $Fixture.DataPort.Enable = true Comment = "Enable DataPort outputs"
Evaluate $Fixture.FixturePSURelay = true Comment = "Apply fixture power"
Evaluate $System.Wait(100ms) Comment = "delay to allow PSU output to settle"
For ( Int Address = 0 ; Address < 2048 ; Address = Address + 1 )
Evaluate WriteByte(Address,[ Byte] Address)
Comment = "fill each EEPROM location with the bottom byte value of its own address"
Evaluate $Fixture.FixturePSURelay = false
Comment "Remove fixture power"
Evaluate $Fixture.DataPort.Enable = false
Comment = "Disable DataPort (Tri-state)"
```

When executed, the above code enables the output port, applies the fixture power, writes 2048 bytes to the EEPROM, removes power and finally disables the output port. The test pattern used is effectively "count fill" in the range 0.255 (0b00000000..0b11111111). An alternative might be to use an alternating complementary pattern such as 0b01010101 and 0b10101010.

```
For ( Int Address = 0 ; Address < 2048 ; Address = Address + 1 )
If ((Address & 1)==0)
Then
Evaluate WriteByte(Address,[ Byte] 0b01010101)
Else
Evaluate WriteByte(Address,[ Byte] 0b10101010)
```

## EEPROM Verification

In order to verify the contents of the EEPROM a data read capability must be added. As the data is returned one bit at a time in a serial data stream we first implement the ability to receive a single bit of data, then we implement a means of receiving data as bytes or words and finally a method for reading data from a specified EEPROM address.

## RecvBit Method

Create a new Method called RecvBit that takes a single Boolean lastBit parameter and returns a bit in the LSB of an Int, with the following Method Code:-

```
Method RecvBit [ Int] ([ Bool ] lastBit)Comment = "receive a single data bit from EEPROM"
Local Symbols
Method Parameters
```

```
Method Parameter [ Bool ] lastBit Comment = "receive last bit in data stream flag"
Variables
Variable [ Int ] dbit = 0 Comment = "the EEPROM data bit"
Method Code
Evaluate $Fixture.DataPort.Data = CS_H Comment = "CS_H,CLK_L,DI_L"
Evaluate dbit = ($Fixture.SenseLines & DO) ^ DO
Comment = "SenseLines are INVERTED so flip(XOR) DO, the data bit"
If ( lastBit )
Then
Evaluate $Fixture.DataPort.Data = [ Byte] (CS_L & CLK_L & DI_L) Comment = "CS_L,CLK_L,DI_L to terminate RECEIVE cycle"
Else
Evaluate $Fixture.DataPort.Data = [ Byte] ((CS_H | CLK_H) & DI_L) Comment = "CS_H,CLK_H,DI_L to continue RECEIVE cycle"
Return dbit
```

Having created the ability to receive data one bit at a time we now implement the method to receive a byte (8-bits) of data. Note that when the lastBit flag is true the CS and CLK lines are returned to LOW (0V) otherwise they are set to HIGH for the next bit receive cycle.

## RecvData Method

Create a new Method called RecvData that takes no parameters and returns 8-bits of serial data from the device having previously sent an instruction that returns a serial data stream:-

```
Method RecvData [ Byte ] ()
Local Symbols
Variables
Variable [ Byte ] bdata = 0 Comment = "the EEPROM byte data"
```

This Method will be used to receive a single byte from the EEPROM.

We now implement the RecvData Method by adding the following:-

```
Method Code
Comment "clock the leading dummy bit"
Evaluate $Fixture.DataPort.Data = CS_H Comment = "CS_H, CLK_L, DI_L"
Evaluate $Fixture.DataPort.Data = [ Byte] (CS_H | CLK_H) Comment = "CS_H, CLK_H, DI_L to continue RECEIVE cycle"
For ( Int i = 7 ; i >= 0 ; i = i + 1 )
Evaluate bdata = [ Byte] ((RecvBit(i==0) << i) | bdata)
Return bdata
```

When executed, the above code clocks the leading dummy bit, then receives the next 8 data bits. Note the lastBit flag is set to true when bit 0 (the last of the 8 bits) is being received.

### ReadByte Method

Create a new Method called ReadByte that takes one parameter, the target address and returns the data byte received from the specified location :-

```
Method ReadByte [ Byte ] ( [ Int ] address )
    Local Symbols
    Method Parameters
    Method Parameter [ Int ] address Comment = "the EPROM Address"
    Variables
    Variable [ Byte ] bdata = 0 Comment = "the EPROM data"
```

This Method will be used to read a single byte at the specified EPROM address.

We now implement the ReadByte Method by adding the following:-

```
Comment "send read instruction"
Evaluate SendStart()
Evaluate SendOpCode(READ)
Evaluate SendAddress(address)
Comment "read the data"
Evaluate bdata = RecvData()
Return bdata
```

When executed, the above code sends the read instruction, then the target address and finally reads the data returned by the EPROM.

### EEPROM Write/Read/Verify

In order to write/read/verify the contents of the EEPROM the Main Method can be modified as follows.

```
Evaluate $Fixture.DataPort.Enable = true Comment = "Enable DataPort outputs"
Evaluate $Fixture.FixturePSURelay = true Comment = "Apply fixture power"
Evaluate $System.Wait(100ms) Comment = "delay to allow PSU output to settle"
For ( Int Address = 0 ; Address < 2048 ; Address = Address + 1 )
    Evaluate WriteByte(Address,[ Byte] Address)
    Comment = "fill each EEPROM location with the bottom byte value of its own address"
For ( Int Address = 0 ; Address < 2048 ; Address = Address + 1 )
    Evaluate ReadByte(Address) Comment = "read the contents of each EEPROM location"
Evaluate $Fixture.FixturePSURelay = true Comment "Remove fixture power"
Evaluate $Fixture.DataPort.Enable = false
    Comment = "Disable DataPort (Tri-state)"
```

Additional code may be added to carry out a comparison with the expected data. This requires the addition of a Local Variable to the

Main Procedure as shown below.

```
Local Symbols
Variables
```

```
Variable [ Byte ] bdata = 0 Comment = "the EEPROM byte reference data"
```

The contents of the second For loop may be changed to include an Analog Test structure as follows:-

```
For ( Int Address = 0 ; Address < 2048 ; Address = Address + 1 )
```

```
Analog Test EEPROM TestLimits = 0,255 DeviceID = IC1
```

Setup

Section Code

```
Evaluate bdata = [ Byte] (Address & 0x00ff)
Comment = "set reference values for each EEPROM location to the Least Significant Byte(LSB) value of its own address"
Evaluate $Test.ResultLimits.LowLimit = [ SIUnknown] bdata
Evaluate $Test.ResultLimits.LowLimit = [ SIUnknown] bdata
```

Measure

Section Code

```
Evaluate $Test.TestResult = [ SIUnknown] ReadByte(Address) Comment = "read each EEPROM location and assign to test result"
```

Upon execution of the Analog Test structure, the PASS/FAIL result status is automatically generated. By adding a few statements to the Main method we can display the status of each test in the AIDE Results windows as shown.



To achieve this the contents of Main become:-

```
Evaluate $System.Reset()
Evaluate $Program.DisplayChannels.Clear()
Evaluate $Program.PrintChannels.Clear()
Evaluate $Scope.DebugMode = Off
Evaluate $Scope.DisplayMode = DisplayOnTest
Evaluate $Scope.PrintMode = PrintOnTest
Evaluate $Program.DisplayChannels.Add(Display Channel.IDE. Create())
Evaluate $Program.PrintChannels.Add(Print Channel.IDE.Create ("DIO Test Display"))
Evaluate $Fixture.DataPort.Enable = true Comment = "Enable DataPort outputs"
Evaluate $Fixture.FixturePSURelay = true Comment = "Apply fixture power"
Evaluate $System.Wait(100ms) Comment = "delay to allow PSU output to settle"
```

```

For ( Int Address = 0 ; Address < 2048 ; Address
    = Address + 1 )
Evaluate WriteByte(Address,[ Byte] Address)
    Comment = "fill each EEPROM location with the
    bottom byte value of its own address"
For ( Int Address = 0 ; Address < 2048 ; Address
    = Address + 1 )
Analog Test EEPROM TestLimits = 0,255 DeviceID =
    IC1
Setup
    Section Code
        Evaluate bdata = [ Byte] (Address & 0x00ff)
        Comment = "set reference values for each
        EEPROM location to the bottom
        byte value of its own address"
        Evaluate $Test.ResultLimits.LowLimit =
        [ SIUnknown] bdata
        Evaluate $Test.ResultLimits.LowLimit =
        [ SIUnknown] bdata
Measure
    Section Code
        Evaluate $Test.TestResult =
        [ SIUnknown] ReadByte(Address) Comment =
        "read each EEPROM location and assign to
        test result"
        Evaluate $Fixture.FixturePSURelay = true
        Comment "Remove fixture power"
Evaluate $Fixture.DataPort.Enable = false
    Comment = "Disable DataPort (Tri-state)"

```

The additional statements used to produce result output are described in other Application Notes in this series.

### Writing and Reading Mixed Data Types

The purpose of relatively small capacity EEPROM devices is usually to store configuration information, calibration values etc. It is therefore useful to be able to convert such data into a Byte data stream and then write the data one byte at a time to the device. The reciprocal read function is also required.

### DoubleToBytes Method

Create a new Method called DoubleToBytes that takes one parameter, the Double dub and returns the 8-byte data array. This method uses the .NET Framework MemoryStream class.

Add the following .NET Namespace Reference :-

```
.NET Namespace Reference System.IO
```

Create the DoubleToBytes Method as follows :-

```

Method DoubleToBytes [ Byte[] ] ( [ Double ] dub )
    Comment = "Convert Double value to array of 8
    bytes"
Local Symbols
    Method Parameters
        Method Parameter [ Double] dub      Method
Variables
    Variable [ MemoryStream ] ms
    Variable [ BinaryWriter ] bw
    Variable [ BinaryReader ] br
    Variable [ Byte[] ] byteArray = new Byte[ 8]
Method Code

```

```

Evaluate ms = new MemoryStream()
Evaluate bw = new BinaryWriter(ms)
Evaluate br = new BinaryReader(ms)
Evaluate bw.Write(dub)
Evaluate ms.Position = 0
For ( Int i = 0 ; i < 8 ; i = i + 1 )
Evaluate byteArray[ i ] = br.ReadByte()
Evaluate bw.Close()
Evaluate br.Close()
Evaluate ms.Close()
Return byteArray

```

### BytesToDouble Method

Create a new Method called BytesToDouble that takes one parameter, the array of Bytes byteArray and returns the Double data array. This method uses the .NET Framework MemoryStream class.

Create the BytesToDouble Method as follows :-

```

Method BytesToDouble [ Double ] ( [ Byte[] ]
    byteArray ) Comment = "Convert array of 8 bytes
    to Double value"
Local Symbols
    Method Parameters
        Method Parameter [ Byte[] ] byteArray
        Method

```

Variables

```

Variable [ MemoryStream ] ms
Variable [ BinaryWriter ] bw
Variable [ BinaryReader ] br
Variable [ Double ] dub

```

Method Code

```

Evaluate ms = new MemoryStream()
Evaluate bw = new BinaryWriter(ms)
Evaluate br = new BinaryReader(ms)
For ( Int i = 0 ; i < 8 ; i = i + 1 )
Evaluate bw.Write(byteArray[ i ] )
Evaluate ms.Position = 0
Evaluate dub = br.ReadDouble()
Evaluate bw.Close()
Evaluate br.Close()
Evaluate ms.Close()
Return dub

```

We can now create the Methods WriteDouble and ReadDouble as follows to write/read a double value starting at the specified EEPROM address.

## WriteDouble Method

Create a new Method called WriteDouble that takes two parameters, the Int startAddress and the Double value and returns nothing.

```
Method WriteDouble [ ] ( [ Int ] startAddr,[ Double ] value ) Comment = "Write Double value to 8 consecutive locations in EEPROM"
```

Local Symbols

Method Parameters

```
Method Parameter [ Int ] startAddr
```

```
Method Parameter [ Double ] value
```

Variables

```
Variable [ Byte[] ] byteArray = new Byte[ 8]
```

Method Code

```
Evaluate byteArray = DoubleToBytes(value)
```

```
For ( Int i = 0 ; i < 8 ; i = i + 1 )
```

```
    Evaluate WriteByte(startAddr+i,byteArray [ i ])
```

## ReadDouble Method

Create a new Method called ReadDouble that takes one parameter, the Int startAddress and returns the Double value.

```
Method ReadDouble [ Double ] ( [ Int ] startAddr) Comment = "Read Double value from 8 consecutive locations in EEPROM"
```

Local Symbols

Method Parameters

```
Method Parameter [ Int ] startAddr
```

Variables

```
Variable [ Byte[] ] byteArray = new Byte[ 8]
```

```
Variable [ Double ] value
```

Method Code

```
For ( Int i = 0 ; i < 8 ; i = i + 1 )
```

```
    Evaluate byteArray[ i ] = ReadByte(startAddr+i)
```

```
Evaluate value = BytesToDouble(byteArray)
```

Return value

We can also create the Methods WriteString and ReadString as follows to write/read a character string starting at the specified EEPROM address.

## WriteString Method

Create a new Method called WriteString that takes two parameters, the Int startAddress and the String str and returns nothing. This method uses the .NET Framework ASCIIEncoding class.

Add the following .NET Namespace Reference :-

```
.NET Namespace Reference System.Text
```

Create new Global Variable of type ASCIIEncoding as follows.

Global Symbols

Variables

```
Variable [ ASCIIEncoding ] encoder  
InitialValue = new ASCIIEncoding()
```

Create the WriteString Method as follows:-

```
Method WriteString [ ] ( [ Int ] startAddr,[ String ] str ) Comment = "Write String to successive
```

```
locations in EEPROM"
```

Local Symbols

Method Parameters

```
Method Parameter [ Int ] startAddr
```

```
Method Parameter [ String ] str
```

Variables

```
Variable [ Byte[] ] byteArray
```

Method Code

```
Evaluate byteArray = encoder.GetBytes(str)
```

```
For ( Int i = 0 ; i < str.Length ; i = i + 1 )
```

```
    Evaluate WriteByte(startAddr+i, byteArray[ i ])
```

```
Evaluate WriteByte(startAddr+ str.Length,0)  
Comment = "null to indicate end of string"
```

## ReadString Method

Create a new Method called ReadString that takes one parameter, the Int startAddress and returns a String. This method also uses the .NET Framework ASCIIEncoding class.

```
Method ReadString [ String ] ( [ Int ] startAddr )  
Comment = "Read String from successive locations in EEPROM (terminated by a 'null' character)"
```

Local Symbols

Method Parameters

```
Method Parameter [ Int ] startAddr
```

Variables

```
Variable [ Byte[] ] byteArray
```

```
Variable [ Byte ] byteVal
```

```
Variable [ Int ] n = 0
```

Method Code

```
Evaluate byteVal = ReadByte(startAddr)  
While ( byteVal != 0 )
```

```
    Evaluate n = n + 1 Comment = "increment character count until 'null' is detected"
```

```
    Evaluate byteVal = ReadByte(startAddr + n)
```

If ( n > 0 )

Then

```
    Evaluate byteArray = new Byte[ n]
```

```
    For ( Int i = 0 ; i < n ; i = i + 1 )
```

```
Evaluate byteArray[ i ] = ReadByte(startAddr + i)
```

```
    Return encoder.GetString(byteArray)
```

Else

```
    Return ""
```

To Write a Date and Time value to an EEPROM at a specific address we can now use the following code.

```
Evaluate WriteString(address, DateTime.Now.ToString())
```

Where address is the location in the EEPROM where the DATE/TIME String is to be stored.

## Libraries

It becomes even simpler as projects and solutions develop to place all the Methods that might be shared into appropriate Libraries. For example the above methods associated with using the PUC to read and write to an EEPROM can easily be placed in a PUCLibrary, in a Module called EEPROM. Accessing these Methods simply requires the Library to be imported into the user's Program and the Methods referenced by statements of the form :-

```
Evaluate PUCLibrary.EEPROM.WriteByte (address,  
data)
```

## Summary

It is relatively simple to utilise the I/O facilities on the Power and Utility Card (PUC) in order to control and communicate with simple devices where speed is NOT a critical requirement. The above methods are NOT suitable for high speed testing of large memory devices but may be used successfully where the amount of data being transferred is fairly small. Timing between successive interface signal state cycles is largely determined by the speed of the host computer. Explicit delays are only required to ensure completion of a particular interface state when testing for a handshake type of response as shown in the WriteByte Method where we are testing for the BUSY/READY state transition to ensure the current write cycle has completed.

**CHINA Beijing**

Tel: [+86] (10) 6539 1166  
Fax: [+86] (10) 6539 1778

**CHINA Shanghai**

Tel: [+86] 21 2028 3588  
Fax: [+86] 21 2028 3558

**CHINA Shenzhen**

Tel: [+86] (755) 3301 9358  
Fax: [+86] (755) 3301 9356

**FINLAND**

Tel: [+358] (9) 2709 5541  
Fax: [+358] (9) 804 2441

**FRANCE**

Tel: [+33] 1 60 79 96 00  
Fax: [+33] 1 60 77 69 22

**GERMANY**

Tel: [+49] 89 99641 0  
Fax: [+49] 89 99641 160

**HONG KONG**

Tel: [+852] 2832 7988  
Fax: [+852] 2834 5364

**INDIA**

Tel: [+91] 80 [4] 115 4501  
Fax: [+91] 80 [4] 115 4502

**JAPAN**

Tel: [+81] (3) 3500 5591  
Fax: [+81] (3) 3500 5592

**KOREA**

Tel: [+82] (2) 3424 2719  
Fax: [+82] (2) 3424 8620

**SCANDINAVIA**

Tel: [+45] 9614 0045  
Fax: [+45] 9614 0047

**SINGAPORE**

Tel: [+65] 6873 0991  
Fax: [+65] 6873 0992

**TAIWAN**

Tel: [+886] 2 2698 8058  
Fax: [+886] 2 2698 8050

**UK Stevenage**

Tel: [+44] (0) 1438 742200  
Fax: [+44] (0) 1438 727601  
Freephone: 0800 282388

**USA**

Tel: [+1] (316) 522 4981  
Fax: [+1] (316) 522 1360  
Toll Free: 800 835 2352

As we are always seeking to improve our products, the information in this document gives only a general indication of the product capacity, performance and suitability, none of which shall form part of any contract. We reserve the right to make design changes without notice. All trademarks are acknowledged. Parent company Aeroflex, Inc. ©Aeroflex 2011.

[www.aeroflex.com](http://www.aeroflex.com)  
[info-test@eroflex.com](mailto:info-test@eroflex.com)



Our passion for performance is defined by three attributes represented by these three icons: solution-minded, performance-driven and customer-focused.