Title:

# EZWare-500 Macro Language

**Article Number: TN1096**

## Information in this article applies to:

HMI500/Silver Series touch screens & EZware-500 software

| HMI Product(s) | Controller (PLC) Product(s) |
|---|---|
| **HMI500/Silver Series** | **All** |

## Summary

Occasionally, project programmers need to perform specific types of data transfer or conditional functions using Silver Series touchscreens that are not typically supported by EasyWare-500 configuration software.

## Solution

EZware-500's macro function allows programmers to design macros that perform simple transfers of data and/or simple to complex conditional logic and transfers not typically supported by EZware's listed features.

Macros are high-level language programs that can run inside the HMI to perform various data transfers and conditional logic. They are useful for doing tasks that are not supported in a standard object or feature of the HMI. The macro language combines syntax elements of C, C++ and a little of its own style.

Previous experience in programming with C or C++ would be helpful when developing macros. It is not the scope of this document to explain concepts of high-level language programming and the basics such as: variable data types, arrays, conditional logic, and variable passing. However, it is the scope of this document to discuss the commands and command structures that are required in building macros. This document will also include any known problems within the macro language.

# Table of Contents

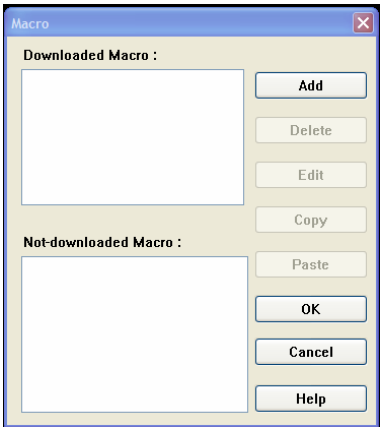
# 1. Macro Sample & Implementation

Macros can be used for customizing an application, or when a special operation is needed that the normal HMI objects (set bit, data transfer, etc) do not provide. A few possibilities are listed:

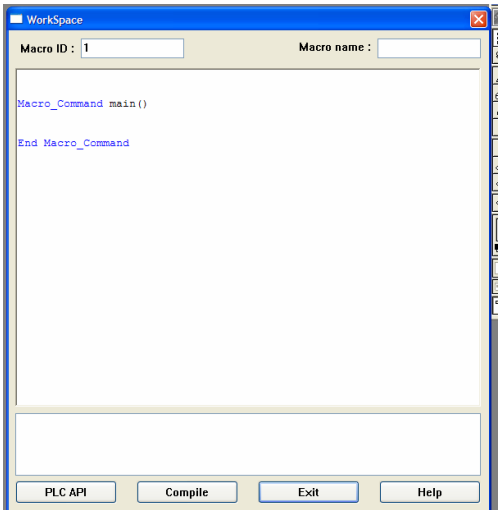
- Unusual or non-linear scaling of data
- Displaying the result of arithmetic operations on 2 or more PLC register values
- Indirect addressing (where a PLC register holds the address of the desired data)
- Lookup tables
- Transferring data once, instead of periodically
- Writing data to the PLC in a specific sequence
- Timed splash screens

▸ To call the macro workspace in order to create a macro…

1. Select **Tools-Macro…** The Macro dialog appears




2. Click the **Add** button to create the Macro workspace.



3. Provide a *name* in the **Macro name:** field.
4. Once the above steps have been completed, programming can be added. Upon completion of programming, the code must be compiled.
    - If the code compiles correctly with no errors, the macro name will be added to the **Downloaded Macro:** section of the Macro dialog.
    - If the macro does not compile correctly, error codes will be displayed, and the errors must be fixed before the macro is added to the **Downloaded Macro:** section.
    - If the programmer cancels the macro before a successful compile, the macro will appear in the **Not-downloaded Macro:** section of the Macro dialog. Items appearing in this section are not available as usable macros, but the text remains for later editing.
5. Click the **OK** button in the **Macro** dialog.

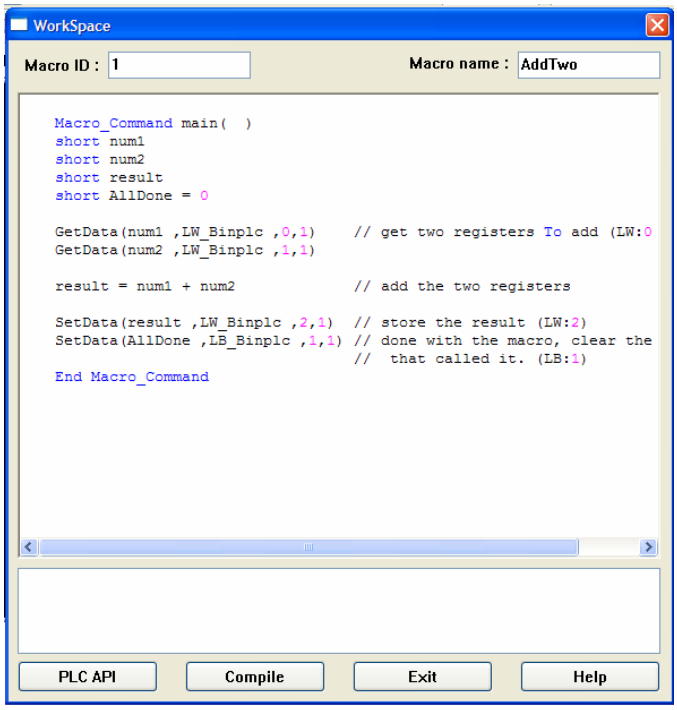
The macros are triggered by setting a bit. The bit is specified in the **PLC Control Object** list. The bit may be a local bit 'LB' or it may be a PLC bit. Each macro created has a unique name and bit. When the trigger bit is set high, it executes the macro. The macro's variables are initialized, the macro runs and will exit when finished.

---

*The bit that called the macro into execution should be cleared by the macro before it exits, otherwise the macro will immediately be entered again upon exit. If this immediate re-entry should happen, this will form an endless loop that affects screen updates and communications with the PLC.*

---

A sample macro is listed below which reads two local word registers and adds them together then stores them into another register location:

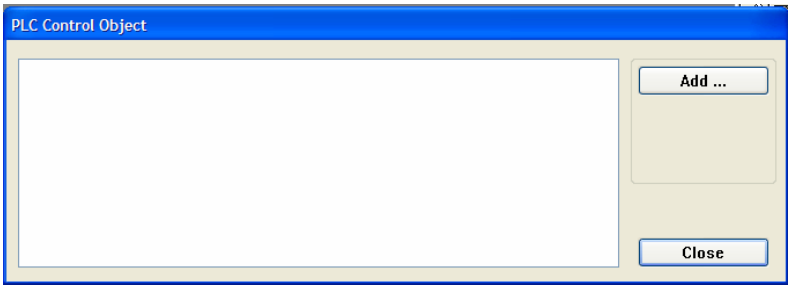


Note that **Macro Name:** is AddTwo and the bit that executes this macro is LB:1.

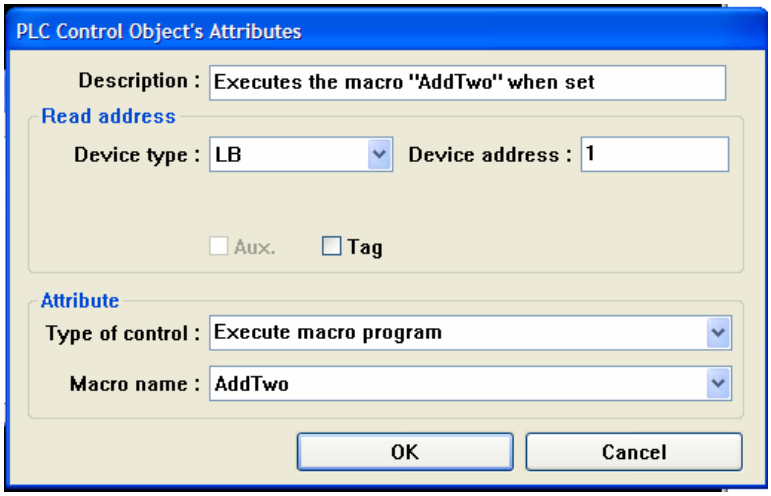
Next, a PLC control object is created for the macro.

**To create the AddTwo PLC control object:**

1. Select **Parts-PLC control**. The **PLC Control** dialog appears.

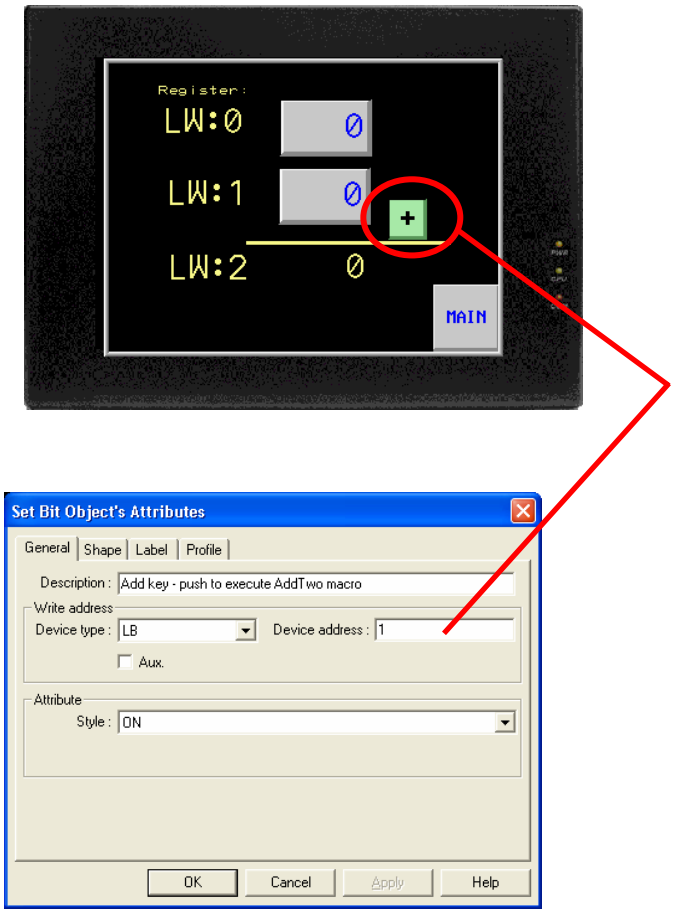



2. Click Add… The **PLC Control Object's Attributes** dialog appears.




3. In the **Attribute** section…
   a. Select **Execute macro program** from the **Type of control:** drop down list.
   b. Select **AddTwo** from the **Macro name:** drop down list.
4. In the Read address section:
   a. Select **LB** from the **Device type:** dropdown list.
   b. Type **1** in the **Device address:** box.
5. Type a *description of the macro* in the **Description:** box.
6. Click **OK**.

To execute the macro, a ‘set-bit’ object is placed on the screen as a ‘+’ button. It has the attribute set to ‘ON’. When this button is pushed, it forces LB:1 to ON and the ‘AddTwo’ macro executes, storing the result in LW:2 and turning LB:1 to OFF and the macro stops:




## 2. Variables, Declarations, and Memory Usage

Variables, constants and functions are all named by the programmer.

**Variables:**

- Variable names can be any character or number (a-z, A-Z, 0-9, or ‘_’).
- A maximum of 32 characters is allowed for the variable name.
- Variable names are not case sensitive, so ‘result’ is the same as ‘ReSult’ or ‘RESULT’.
- Variable names must always start with a letter (i.e. variables cannot begin with a number or the underscore ‘_’ character).

**Memory Usage:**

Macros use a minimum of 100 bytes, plus the memory required for each variable type, functions and conditional statements. Below is an approximate size:

- Each variable declaration (float, int, short, bool) takes apx. 14 bytes
- Each GetData() & SetData() functions takes apx. 34 bytes

- Each simple equation (c = A [+-*/] B) takes apx 22 bytes
- Each conditional statement takes an apx. minimum of 42 bytes

Macros use memory from the background task memory (defined as *Window 0* ) area. There is a default memory size of 320K available for macros (only 220K is allotted if the 'Fast Selection' task bar is enabled) and there are other background task objects that also use memory in this area, such as:

- Trends
- XY Plots
- Data Transfers
- Alarms
- Events
- PLC Controls
- Printer functions

---

*If the application exceeds the 320K limit, both the simulator and the HMI display a 'System Severe Error' message. To check the memory usage, right-click on the Simulator window and select* ***System Resource*** *from the menu. To add more memory to this area, reduce the number of total pop-ups possible. The setting is in the system parameters {General} tab → "No. of Windows". The default is 6; change this to 4 and the memory will increase (be re-allocated) for window 0 (background task) functions.*

---

**Variable Declarations:**

Each variable used in the macro needs to be declared as a specific type of register. The declarations are listed as the first part of the macro. Any declarations contained within the macro function are considered 'Local' and any variable outside the macro function are considered 'Global'.

- Local variables are only seen within the function in which they are declared.
- Global variables retain their values, and can be globally used by all functions & sub-functions.

Below, are listed the various data types that can be declared:

| Type | Description | Declared Size | Range |
|---|---|---|---|
| Float | Single-Precision Floating point variable | 32-bit signed, IEEE-754 format | Varies depending on decimal point use |
| Int | Integer variable | 32-bit signed | - 2,147,483,648 to 2,147,483,647 |
| Short | Short integer variable | 16-bit signed | -32768 to 32767 |
| Char | Character variable | 8-bit signed | -128 to 127 |
| Bool | Boolean variable | 1-bit | 0 to 1 |

**Variable Initialization:**

Initialize a value of variable in the declaration statement directly. (e.g.: int RPM = 75) Use the assignment operator (=) to initialize a value to the variable. Variables are declared and initialized in the following manner:

Stacked Example:

```
short a = 0
short b = 0
short c = 0
```

Inline Example (separate like-types with a comma):

```
short a=0, b=0, c=0
```

---

*Variables inside macros are initialized to all '1's as a default (i.e. 0xFFFF), so don't assume they are zero values when you enter the macro. It is good programming practice to initialize variables during declaration, or use assignment statements before they are used.*

---

**Constants:**

A constant is a numeric or Boolean value that does not change. The reserved keywords 'True' and 'False' are treated as constants. Constants may be written as any of the following:

| | Written as: | Examples |
|---|---|---|
| Decimal constant | 1234 | short MyVal = 1234 |
| Hexadecimal constant | 0xFA20 | short MyVal = 0xFA20 |
| ASCII code (character constant) | 'ABCD' | char String[4] = 'ABCD' |
| Boolean: True (not zero), False (zero) | True, 1, False, 0 | bool Done = 0, or, bool Done = False |

## 3. Local and global variables

- Local variables exist only within the function in which they are defined.
- Global variables exist throughout the entire macro, and are available for any function in the macro. Declare these variables outside and before any Sub() functions and the Main() function.

---

*If a local variable has the same name as a global variable, the compiler uses the local variable in the function instead. Global variables are global only within the macro file in which they exist. It is not possible to share variables between different macro files. To share data between macro files, read/write the data into the HMI's Local Word (LW) and/or Local Bit (LB) addresses using the macro's GetData() and SetData() functions.*

---

## 4. Creating Variable Arrays

- Arrays are fixed-depth and one-dimensional only.
- Arrays can contain up to 65,535 elements.

- The ***type*** is the declaration of the array of elements, such as an array of *ints*, or *shorts*, or *bools*, etc.
- The ***array_size*** is the number of elements to contain in the array, such as an array of [10] would contain 10 elements of the declared ***type***. The first register in an array always begins with the $0^{th}$ element and the last register is the declared number of elements minus one. Only one-dimensional arrays are supported. Arrays can be pre-initialized.

  The format is: ***[type] Array_Name[Array_Size]***

  Optional array initialization examples:

```
int MyArray[10] = {1,2,3,4,5,6,7,8,9,10}
char LetterArray[6] = 'MYWORD'
short Data[2]
```

  The initial values are written within the brackets {} and are divided by comma (,). These values are assigned in order from left to right starting from array index=0. Text characters are contained within (‘) apostrophes and are not separated.

  For example:

```
char table[100]
```

The above declaration defines an array of 100 type ‘*char*’ registers named ‘table’. The first element in ‘table’ is address 0 and the last is 99, so it is used as:

```
char FirstElement
char LastElement

FirstElement = table[0]
LastElement = table[99]
```

## 5. Reserved words:

The following symbols and names are keywords reserved for use by macros. They cannot be used (as a complete name) in any function name, array name, or variable name; however, the reserved words may be contained within a variable name such as: my_int, TheEnd, etc.

| | | | |
|---|---|---|---|
| And | Continue | int | Step |
| Bcdaux | down | Macro_Command | Then |
| Bcdplc | Else | next | To |
| Binaux | End | not | True |
| Binplc | False | Or | void |
| bool | float | return | wend |
| Break | For | select | While |
| Case | GetData | SetData | xor |
| char | If | short | |

## 6. Operators

Below is a list of the recognized operators.

| Group: | Name: | Sym: |
|---|---|---|
| Assignment operator: | Assignment: | = |
| | | |
| Arithmetic operators: | Addition: | + |
| | Subtraction: | - |
| | Multiplication: | * |
| | Division: | / |
| | Modulo: | % |
| | | |
| Comparison operators: | Less than : | < |
| | Less than or equal to: | <= |
| | Greater than: | > |
| | Greater than or equal to: | >= |
| | Is equal to: | == |
| | Not equal: | <> |
| | | |
| Logic operators: | Conditional 'AND': | AND |
| | Conditional 'OR': | OR |
| | Exclusive OR : | XOR |
| | Boolean NOT : | NOT |
| | | |
| Bitwise and shift operators: | Left shift: | << |
| | Right shift: | >> |
| | Bitwise AND: | & |
| | Bitwise OR: | \| |
| | Bitwise XOR: | ^ |
| | Bitwise complement: | ~ |

**Order of Precedence:**

The process order of many operators within an expression is called the 'order of precedence'. The priority of the same kind of operator is from left to right, and from top to bottom.

- Arithmetic operator: ^ ( * , / ) ( % ) ( + , - )
- Bitwise & Shift operators: From left to right within the expression
- Comparison operator: From left to right within the expression
- Logic operators: Not And OR XOR,
- Arithmetic operator is higher priority than a Bitwise operator
- Bitwise operator is higher priority than a Comparison operator
- Logic operator is higher priority than an Assignment operator

## 7. Main Function and Sub-Functions

The macro must have one and only one "**Macro_Command main()"** function which is the execution start point of any macro function. Any sub-functions must be pre-defined and written before the **main()** function.

The format is:

```
Macro_Command main( )
   // The main macro code goes here
End Macro_Command
```

Any other sub functions must be declared before the main() function can use it, for example:

```
Sub int SQR(int MyVar )
   // sub routine code goes here
End Sub

Macro_Command main( )
  // The main macro code goes here
  Result = SQR(MyNum)  //The sub function "SQR()" is called
End Macro_Command
```

## 8. Reading & Writing External Registers:

There are two main functions that communicate with external data registers and I/O.

- The function GetData() receives data from the PLC (or the HMI's local memory).
- The function SetData() sends data to the PLC (or the HMI's local memory).

The GetData(), or the SetData(), can be hand typed according to the format listed below, or the "PLC API" button located at the bottom-left of the macro editor dialog box will run a wizard dialog box to generate the proper GetData() or SetData() syntax by selecting the appropriate responses within the API dialog box. Be sure to declare all variables within the macro editor and be sure the curser is located after the declarations before using the 'PLC API' option so they will be available to use from the drop down list.

### GetData() Function:

This function reads data from an external register located either in the HMI's memory or in the PLC memory, and stores it into a pre-declared destination variable within the macro.

The format for the GetData() function is:

```
GetData(Parameters:DestVariable, Addr_Type, Address, NumElements )
```

The parameters are as follows:

| Parameter | Description | Example(s) |
|---|---|---|
| *DestVariable* | The name of the variable to store the data that is captured. This must be a pre-declared variable. | MyVar, InputArray[0] |
| *Addr_Type* | The device type and encoding method (binary or BCD) of the PLC data. See note below on formatting this parameter. | LW_Binplc, N7_Binplc, 0x_Binaux, |

| | | |
|---|---|---|
| | | RW_Bcdplc |
| *Address* | The address offset of the register type to read from. Must be a constant whole-number. Bit addresses that would normally be formatted with a decimal or slash such as B3:1.03 is to be written as a whole # (i.e: 103) | 0,<br>1,<br>103,<br>3048<br>32767 |
| *NumElements* | Number of elements to read/write (must be at least 1 and it must be a constant whole number) If the number of elements is more than 1, the 'DestVariable' parameter must be declared as an array. – Maximum allowed is 127 | 1,<br>16,<br>100<br>127 (Max.) |

## SetData() Function:

This function writes data from within the macro to an external register or registers that are located either in the HMI's memory or in the PLC memory.

The format for the SetData() function is:

```
SetData(Parameters: Variable, Addr_Type, Address, NumElements )
```

The parameters are as follows:

| Parameter | Description | Example(s) |
|---|---|---|
| *Variable* | The name of the variable that contains the data that is to be written. This must be a pre-declared variable. This may be an array. | MyVar,<br>OutputArray[0] |
| *Addr_Type* | The device type and encoding method (binary or BCD) of the PLC data. See note below on formatting this parameter. | LW_Binplc,<br>N7_Binplc,<br>0x_Binaux,<br>RW_Bcdplc |
| *Address* | The address offset of the register type to write to. Must be a constant whole-number. Bit addresses that would normally be formatted with a decimal or slash such as B3:1.03 is to be written as a whole # (i.e: 103) | 0,<br>1,<br>103,<br>4096,<br>32767 |
| *NumElements* | Number of elements to read/write (must be at least 1 and it must be a constant whole number) If the number of elements is more than 1, the 'DestVariable' parameter must be declared as an array. – Maximum allowed is 127 | 1,<br>16,<br>100<br>127 (Max) |

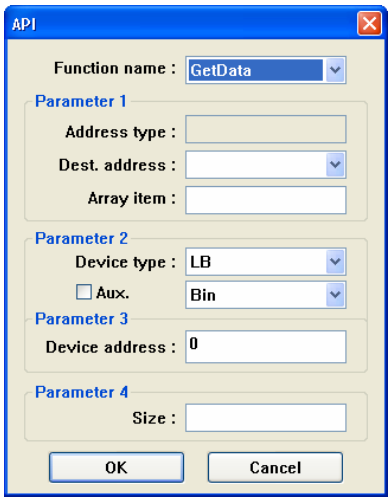
---

*Formatting the Address types are done automatically in the API. However if you type in the address type in the macro editor, the format is as follows:* ***[Register Type]_[Bin/Bcd][plc/aux]****.*

---

## Using the 'PLC API' Wizard:

The PLC API wizard formats the appropriate GetData() or SetData() syntax for all the parameters that are needed.




**To format the GetData or SetData syntax…**

1. Select the function from the **Function Name:** drop-down list:
    - **GetData** to read a register
    - **SetData** to write to a register
2. Set **Parameter 1**, which is a list of the pre-declared variables:
    a) Select the **Dest address:** from the drop-down list.
        - If some or all of the variables are not present in the list, cancel the wizard and place the curser below the declared variables and start the wizard again.
    b) If this variable was declared as an array, it will then allow you to enter the array element number to start with in the **Array item:** box; otherwise the box will be shaded out.
3. Set **Parameter 2,** which selects the register type:
    a) Select the **Device type** to read from (if a GetData) or write to (if a SetData).
    b) Select whether it is Binary format ‘Bin’ or Binary Coded Decimal format ‘Bcd’
    c) If this register is to use the PLC located on the Aux Serial Port, then check the “Aux” check-box
4. Set **Parameter 3:**
    a) Enter the *register’s address* in the **Device address:** box
5. Set **Parameter 4:**
    a) In the **Size:** box, enter the number of elements to write. Typically this is 1. A value of 0 is not valid. If reading/writing to an array of variables, then specify the number of elements to read/write.

## 9. Using 32-bit registers within macros

When reading or writing a 32-bit (2-word) register that is external to the macro using GetData() or SetData(), use an array of (2) 16-bit 'shorts'. Internally, the macro can use an 'int' type variable to store a value up to +/- 2,147,483,647 (a signed 32-bit register). The GetData() or SetData() functions only perform a 16-bit external transfer even though it may be declared as 'int' (32-bit).

```
// This routine writes Hex value 0x12345678 to a 32-bit holding register
short Data[2]
int value = 0x12345678

// split 32-bit value into (2) 16-bit values
   Data[1] = (value & 0xFFFF0000) >> 16    //store the MSW
   Data[0] = value & 0x0000FFFF            //store the LSW

// store the (2) 16-bit values into LW14 and LW15
   SetData(Data[0] ,LW_Binplc ,14,2)
```

```
// This routine reads a 32-bit holding register
short Data[2]

// grab a 32-bit register [(2) 16-bit] value from a PLC register address 4x:1
   GetData(Data[0] ,4x_Binplc ,1,2)
```

```
// This routine puts (2) 16-bit array variables into one 32-bit variable
Int LVal
short Data[2]

LVal = (Data[1] << 16) + Data[0]
```

## 10. Using floating point registers within macros

Reading and writing floating point registers can only be used within a macro as a variable holding register or to transfer a variable value from function to function. A variable can be declared as a 'float' type within the macro and used only within the macro itself to contain a float value.

The macro tool cannot read an external floating point register (using a GetData() function) and understand its value. The macro interpreter has no built-in functionality to interpret an external floating point register (IEEE-754 formatted). However, the macro can transfer an external floating point register from an external holding register to another external holding register.

When transferring (reading or writing) a 32-bit floating point (also call a 'real') register that is external to the macro using GetData() or SetData(), you must use an array of (2) 16-bit 'shorts'. The GetData() or SetData() functions only do a binary transfer of the data. See example code on the next page:

```
// **This routine will not work** A GetData() of more than one element requires
// a variable that is declared as an array of [2] shorts and the RealData will
// not interpret the value captured as a float value but a 16-bit integer value
// stored in a float type container (macro register).
float RealData
```

```
// get the floating point register and store it in 'RealData' variable
   GetData(RealData ,F8_Binplc ,1,2)  //<-- WRONG
```

```
// This routine will work to transfer an external float to an external float
// The HMI LW:20 would be a Numeric Data object set to 'Single Float' 2-words
short FPSource[2]
short FPDestination[2]

// get the floating point register and store it in 'RealData'
GetData(FPSource[0] ,F8_Binplc ,1,2)        // Get float register from PLC F8:1
SetData(FPDestination[0] ,LW_Binplc ,20,2)  // Put float register into HMI LW:20&21 memory
```

## 11. Using recipe memory

The macro utility can easily use the recipe memory area. There are 60,000 recipe word registers that are available to the user for general use (addresses RW:0 to RW:59999). Recipe memory at address RW:60000 and above are reserved and have special functions. See the EasyBuilder Help section or view technical note #TN1099 for the current reserved memory listing.

- Register types: RW, RWI, and the pointer LW:9000 are used to read or write the recipe memory area.
- 'RW' registers are used to directly address a recipe register. 'RWI' registers are used to indirectly address a recipe register by using the pointer in local word memory LW:9000. If indirect addressing is used, then the direct recipe address is determined by adding the value contained in LW:9000 to the address of the RWI register. For example, to indirectly point to the direct recipe address RW:20, the macro would store the value 15 into the LW:9000 pointer and then use RWI address 5 (RWI:5) to GetData() or SetData() {LW:9000 = 15 + RWI:5 → RW:20}.

---

*The pointer has a maximum limitation of 32767, so LW:9000 can indirectly address from an offset of 0 to 32767. To indirectly address recipe registers above 32767, you will need to have the RWI address set to a higher address.*

---

The following macro code example clears all general-use recipe memory area. General-use recipe address range is RW:0 to RW:59999. Because of the pointer's limitation (max of LW:9000 = 32767), it breaks up the incrementing of memory into two sections, the lower half (0 to 29999) and upper half (30000 to 59999).

```
// Macro to clear all general-use recipe memory area. Triggered by LB:1

Macro_Command main(  )
short Address = 0
short Initial = 0
short Done    = 0
short Section = 0

// LW:2 is used as an indicator in the project
SetData(Section, LW_Binplc,2,1)             // LW:2 = 0 (Lower Half)
```

```
For Address = 0 To 29999
  SetData(Address, LW_Binplc,9000,1)        // LW:9000 incr. through memory
  SetData(Initial, RWI_Binplc,0,1)          // RWI:0 indirect address: Low Bank
Next Address

Section = 1
SetData(Section, LW_Binplc,2,1)             // LW:2 = 1 (Upper Half)
For Address = 0 To 29999
  SetData(Address, LW_Binplc,9000,1)        // LW:9000 incr. through memory
  SetData(Initial, RWI_Binplc,30000,1)      // RWI:30,000 indirect address: Upper Bank
Next Address

SetData(Done, LB_Binplc,1,1)                // Turn off the bit that called the macro
End Macro_Command
```

## 12. Conditional Statements & Expressions

### "If … Then … Else" Statement:

This conditional statement evaluates a test condition to see if it is true.

- If the condition is true, then the set of statements that follow directly after the "Then" directive will be executed until the "Else" or "End If" directive is reached.
- If the conditional statement is evaluated as false (not true), then the set of statements that follow directly after the "Else" directive will be executed until the "End If" directive is reached.
- If there is no "Else" directive, and the statement is false, then no statements are executed and the program drops out of the If-Then statement.

| Format | Examples |
|---|---|
| **If [*TestCondition]* Then**<br>**[ *Statements* ]**<br>**Else**<br>**[*elseStatements*]**<br>**End If** | `If x == 1 Then`<br>`  y = y * 5`<br>`  z = z – 12`<br>`Else`<br>`  y = y * 2`<br>`  z = z – 1`<br>`End If`<br><br>`If (x == 0 OR x == 1) AND y == 0 Then`<br>`  z = 10`<br>`End If` |

*A **logic** or **conditional** statement processes a logic expression and branches depending on the result. The ending statement **'End If'** is written as two words, not one word. Note that there is a difference between the conditional equality noted by two equal signs '= =' (read as: 'is equal to') and the assignment equality noted by only one equal sign '='.*

### "Select Case" Statement:

This conditional statement will branch to the appropriate case statements according to the value contained in a variable.

| Format | Examples |
|---|---|
| **Select Case [*variable*]**<br>**Case [*testvalue1*]**<br>**[statement(s)] …**<br>**Break**<br>**Case [*testvalue2*]**<br>**[statement(s)] …**<br>**Break**<br>**Case Else**<br>**[else statement(s)]**<br>**End Select** | `Sub int TEST(int input)`<br><br>`Select Case input`<br>`Case 0`<br>`Return 1`<br><br>`Case 1`<br>`Return 2`<br><br>`Case 2`<br>`Return 3`<br><br>`Case Else`<br>`Return 0`<br>`End Select`<br><br>`End Sub` |

- If the 'Case [*testvalue*]' equals the 'Case [*variable*]' , then the set of statements that follow are executed.
- Each 'Case[*testvalue*]' is evaluated unless the program flow sees a 'Break', at which time the program exits out of the 'Select Case' (Jump past 'End Select').
- If no 'Case [*testvalue*]' is equal to the 'Case [*variable*]', then the 'Case Else' statements are executed.

---

***<< Avoid using "Case" statements for complex logic >>.***

*'Select Case' logic can cause processor lockup or incorrect branching. Case statements must be small in size. The Case variable must be declared as an int type. Best if used within a subroutine function, however a subroutine function may only be called once within the body of the main function otherwise PC computer lock-up may occur during simulation. Maple Systems recommends using 'If – Then – Else' statements instead.*

---

## "For … To … Step" Statement:

This program statement will loop through a progression of iterations until a conditional test becomes false.

| Format | Examples: |
|---|---|
| **For [*variable*] = [*InitialValue*] To/Down [*EndValue*] {Step *amount*}**<br>**[Statement…]**<br>**[Statement…]**<br>**[Statement(s)…]**<br>**Next [*variable*]** | `For loop = 0 To 20`<br>`loop = (loop + 1) *2`<br>`Next loop`<br><br>`For n = 100 Down 0 Step 5`<br>`n = n * 2`<br>`Next n` |

When program execution begins, it assigns the *InitialValue* to the *variable*. The program then performs the statement(s) contained within the For-Next loop. When the statements are finished and the program reaches the "Next [*variable*]" statement, then the program loops back and performs the following two functions:

1.) The variable is incremented or decremented by (1) unless the optional "Step *amount*" value is used.

2.) The variable's new value is then tested with the *EndValue*.

- If the variable is less than or equal to (if incrementing) OR greater than or equal to (if decrementing), then program execution will drop again into the For-Next loop and execute the statements.
- Otherwise the conditional test equates to false and the program jumps past the "Next" statement.

### "While … Wend" Statement:

This program statement performs a recursive loop in the program flow "while" its condition is true.

| Format | Examples |
|---|---|
| **While [*TestCondition*]**<br>**[Statement(s)…]**<br>**Wend** | While counter <= 100<br>SetData(counter, LW_Binplc, 20,1)<br>Wend<br><br>While counter<br>counter = counter - 1<br>Wend |

Entering the "While" statement will first perform a conditional test.

- If the test equates to TRUE, then the statements contained in the While-Wend statement are executed.
- If the test equates to FALSE, the program jumps past the "Wend" statement.
- If a variable alone is used as the "TestCondition", then any value is considered TRUE and only a value of zero (0) is considered FALSE.

## 13. Subroutine function calls and passing parameters

A subroutine function must be defined before its execution and is therefore placed before the main() macro function. If there is an entry attempt into a function before it is first defined in the macro program will create a compiler error 'Function not defined'. Only one subroutine is allowed to be called from within the main macro function.

The example code that follows defines a subroutine function called SQR(). This subroutine function allows one variable to be passed into it. The function then takes the value passed to it, multiplies its value by itself and then stores the result back into the same variable. The result is returned back to the calling function.

```
Sub short SQR(short x)   //This is the subroutine that squares a number
  x = x * x
  return x
End Sub
```

```
Macro_Command main( ) // This main function calls the SQR() function.
short i = 5
  i = SQR(i)
  SetData(i, LW_Binplc, 1,1)  // Write the result to LW:1
End Macro_Command
```

*Requesting more than one subroutine function from the main macro function can cause the PC computer to lockup during a simulation.*

## 14. Precautions, tips & tricks when using Macros

- The size of a macro in a compiled (*.eob) file is limited by the memory of the HMI.
- The maximum storage space of local variables in a macro is 4K bytes.
- A maximum of 256 macros are allowed in an EasyBuilder project.
- A macro may cause the HMI to lock up. Possible causes are:
  - A macro contains an infinite loop with no PLC communication.
  - The size of an array exceeds the storage space in a macro.
  - PLC communication time may cause the macro to execute slower than expected.
  - Some statement bug or limitation may have been exceeded.
- Random macro compiler errors can often be corrected by recompiling again, or canceling out of the macro editor and then reopen and compile again. NOTE: Always click on the {EXIT} button to get out of the macro editor, otherwise if you have a macro that does not compile properly and you click on the [X] button at the top-right, you will lose your macro completely and have to start over.
- To copy and paste code text within macro editor, use the mouse to highlight text, then on the keyboard use CTRL+ C to 'copy' the text or CTRL + X to 'cut' the text and then use the mouse to select the new entry location and use CTRL + V to 'paste' the text.
- When indenting text in the code, use 'spaces', not 'tabs'. A tab can cause a compiler error and will be hard to diagnose.

# Appendix A – Compiler Errors & Error Codes

When there are compile errors, the error description can be referenced by the compiler error message number.

Error message format: Macro_name(: Error_message_number ) Error_Message

| **Error descriptions** | **Possible reason(s)** |
|---|---|
| (1: )"Syntax error:" "'***identifier***' | There are many possibilities for the cause of this compiler error. Simply stated, the compiler found a problem with the syntax of the statement. If there is an identifier name associated, then it is listed. |
| (2:)Used without having been initialized. | Must define the size of an array during declaration. The array size declaration must be a constant whole number, not a variable. |
| (3:)"Redefinition error: " [name] | The name of variable and function within its scope must be unique.<br>`Macro_Command main( )`<br>`int g[10] , g  //<- illegal – redefinition of 'g'`<br>`For g[2] = 0 To 2`<br>`g[3] = 4`<br>`Next g[2]`<br>`End Macro_Command` |
| (4:) Function name error: '**identifier**' | Reserved keywords and constant can not be the name of a function<br>`Macro_Command If( )  //<- illegal – used 'If' in the name` |
| (5:) Parentheses have not come in pairs | The Statement is missing parentheses<br>`Macro_Command main )  //<- illegal – missing '('` |
| (6:) illegal expression with out matching 'If' | |
| (7:) illegal expression [no 'Then'] without matching 'If' | |
| (8:) Illegal Expression [no 'End if'] | The 'End If' is missing from a paired If –Then statement |
| (9:) Unfinished 'If' statement before 'End If' | |
| (10:) Illegal 'Else' | The format of "If" statement is:<br>If [logic expression]Then<br>[ Else [If [logic expression] Then ] ]<br>EndIf<br>Any format other than this format will cause a compile error.<br>** An 'Else' in a Select Case statement must be written "Case Else' |
| (11:) 'Case' expression not constant | Case branches must use constant numbers, not variables.<br>`Case 1  // valid`<br>`Case MyVar  // not valid` |
| (12:) 'Select' statement contains no 'Case' | Case statements must begin as follows:<br>`Select Case [variable]` |
| (13:) illegal expression without matching 'Select Case' | |
| (14:) 'Select' statement contains no 'End Select' | |
| (15:) Illegal 'Case' | 'Case' without 'Select' before it (ie. 'Select Case') |
| (16:)"Unfinished "Select" statement before "End Select" | The format of "Select Case" statement is:<br>`Select Case [expression]`<br>`Case [constant]`<br>`Case [constant]`<br>`Case Else`<br>`End Select`<br>Any format other than this format will cause a compile error. |
| (17:) illegal expression (no 'For') without matching 'Next' | |
| (18:) illegal variable type (not integer or char) | Example: keywords used as variables can produce this error, such as 'For' by itself will produce this error. |
| (19:) variable type error | The variable expected is of the wrong type. (possibly the '=' in an expression is missing) |
| (20:) Must be keyword 'To' or 'Down' | The keyword 'To' or 'Down' is missing from the For-Next statement |

| | |
|---|---|
| (21:) illegal expression (no 'Next') | The format of "For" statement is:<br>**For [*variable*] = [*initial value*] To [*end value*] {*Step*}**<br>**…**<br>**Next [*variable*]**<br>Any format other than this format will cause a compile error. |
| (22:) 'Wend' statement contains no 'While' | Compiler could not match up a beginning 'While' with the 'Wend' |
| (23:) illegal expression without matching 'Wend' | Compiler could not match up a ending 'Wend' with the beginning 'While' statement |
| (24:) Syntax error: 'Break' | • "Break" statement can only be used in "For", "While", or "Select Case" statements<br>• "Break" statement takes one line of Macro. |
| (25:) Syntax error: 'Continue' | • "Continue" statement can only be used in "For" statement, or "While" statement<br>• "Continue" statement takes one line of Macro. |
| (26:) syntax error | This error appears when there is an illegal assignment. For instance:<br>**Exponent = pow(n)**<br>When pow() function has not been declared yet. |
| (27:) Syntax error: | This error appears when there is an illegal operation of an object. The mismatch of the expected object's operation in an expression can cause this compiler error.;<br><br>For example :<br>**If count == Then //<- not a complete expression** |
| (28:) Missing "Sub" | |
| (29:) Missing "Macro_Command" | The format of function declaration is:<br>**Sub(Macro_Command) [data type] function_name( … )**<br>**End Sub(Macro_Command)**<br>Any format other than this format will cause compile error. |
| (30:) number of parameters is incorrect | The number of parameters in a GetData() or Setdata() is incorrect. There are four (4) parameters all separated by commas |
| (31:) parameter type is incorrect | The parameters of a function must agree with the data types passed to a function to avoid compile error. GetData() or SetData() must have 4 parameters (variable, valid register type, constant, constant). |
| (32:) variable is incorrect | The parameters of a function must be equivalent to the arguments passing to a function to avoid compile error. |
| (33:) '***identifier***' : undeclared function | The function requested by '*function_name*()' was not previously declared. The name of the function is the *identifier*. |
| (34:) illegal member of array | |
| (35:) invalid array declaration | Array declaration of size is invalid – ie:<br>**Array[1 + Total] // whole number only, no math allowed** |
| (36) illegal index of array | |
| (37:) undeclared identifier: *[name]* | Any variable or function should be declared before use. |
| (38:) PLC encoding method is not supported | The 2nd parameter of GetData( … ) , SetData( … ) should be a legal PLC register type for the protocol selected plus:<br>'_Binplc' or<br>'_Binaux' or<br>'_Bcdplc' or<br>'_Bcdaux' tacked to the end of the register type. Ex: 'N7_Binplc' |
| (39:) Should be integer, character or constant" ; | The format of array is:<br>Declaration: array_name[constant] (constant is the size of the array)<br>Usage: array_name[integer, character or constant]<br>Any format other than this format will cause compile error. |

| | |
|---|---|
| (40:) "Illegal Macro statement before declaration statement " | For example :<br>`Macro_Command main( )`<br>`int g[10]`<br>`For g[2] = 0 To 2`<br>`g[3] = 4 + g[9]`<br>`int h , k <- illegal – definitions must occur before any statements or expressions`<br>`Next g[2]`<br>`End Macro_Command` |
| (41:) float variables cannot be contained in shift calculation | Bitwise operations are not allowed on float variables i.e<br>`float MyFlt`<br>`MyFlt >> 8  // This will cause error #41, MyFlt is a float` |
| (42:) function must return a value | |
| (43) function should return a value | The function was not declared with a return data type. |
| (44:) float variables cannot be contained in calculation | |
| (45:) "Error PLC address" ; | |
| (46:) array size overflow (max. 4K) | The stack for array variables can not exceed 4k bytes total between all declared arrays.<br>Float (max 1019 elements)<br>Int (max 1019 elements)<br>Short (max 2039 elements)<br>chars (max 2039 elements)<br>bool (max 2039 elements)<br>Also, a 'While' statement by itself can produce this error. |
| (47:) macro command entry function is not only one | |
| (48:) macro command entry function must be only one | The only one main entrance of Macro is:<br>Macro_Command function_name( )<br>End Macro_Command |

# Appendix B – What's New

Version 2.7.0

- There are no macro code changes or fixes with this version.

Version 2.6.2

- **NEW:** The macro 'name' (instead of just its number) is listed in the PLC Control Object ('Execute Macro') Attribute dialog box and it is also listed in the list of PLC control objects.
- **NEW:** When the whole project is compiled (Tools→Compile), the macro size is listed separately from the other project memory-usage sections.
- **Fixed**: The GetData() function where it stored the wrong address.
- **Fixed**: If "extended mode" is selected under EDIT->System Parameters->{Editor} tab->Addressing, then the macro does NOT require that extended addressing format be used. If no extended addressing is put before the address (ie. 2#123), then the address selected is defaulted to the 'PLC Station No.' address in the System Parameters.
- **Bug**: Subroutine functions will crash the macro if called more than one time in a main() macro function.
- **Bug**: If a 16-bit external register is plugged into (i.e. GetData) a macro's INT declared variable, it will work in the simulator, but it will not always work when downloaded to the HMI. The work around is to change the declaration to a SHORT instead of an INT.
- **Bug**: 'Case' statements appear to not work correctly as expected. Avoid using CASE statements. Use nested 'If…Then' statements instead.
- **Bug**: A 'bool' declared variable does not function properly in conditional or logic statements. It is best to declare 'bool' type variables as 'ints' instead.
- **Bug:** Compiler does not clear the accumulator's carry bit after doing a calculation or bit rotation.

Version 2.6.0: (**Contains major changes to macros** )

- **NEW**: The register type now ends with "plc" or "aux" tacked on the end, for example: **GetData(variable, 4x_Bin, 123,1)** is now: **GetData(variable, 4x_Binplc, 123,1)**. "plc" is used if the register is used through the main serial ports or Ethernet port. "aux" is used if the register is obtained through the auxiliary port.
- **NEW**: If "extended mode" is selected under EDIT->System Parameters->{Editor} tab->Addressing, then the macros ***require*** that extended addressing format be used, otherwise you will get compiler errors. The format to use is **NodeAddress** + # **sign** + **address**, for example: **GetData(variable, 4x_Binplc, 1#123,1)** to address node #1, and address 123.
- **Fixed:** the ability to utilize LW:9000 as a pointer for using RWIs (Recipe Word Indexed) registers. RBIs still can not utilize the LW:9000 pointer.
- **Bug**: Inline conditional logic [ie. If `(x==1 AND y==2 AND z==3 Then …)`] will not work unless the variables contained in the conditional statements are defined as 'int's.
- **Bug:** If a GetData() uses an array that does not start on address [0] or [16] or [32] or some other number that is divisible by 16, then the bits can get stored to the wrong locations. If the starting element in the array is 0 or a a multiple of 16, then GetData() works correctly. If the starting element is any other value, the transfer ends at the first 16-bit boundary.

  In the following example, the local bits 300-363 are all set.

  ```
  GetData(result[0] ,LB_Binplc ,300,64)
  ```

  After execution, the first four bits in result[] are set, and the next 12 are cleared. The remaining bits (LB16-63) are all set as expected. To properly load result[], the following code is required:

  ```
  GetData(result[0] ,LB_Binplc ,300,4)
  GetData(result[4] ,LB_Binplc ,304,60)
  ```

  SetData() appears to work correctly.
- **New Limitation:** the 4th parameter of a GetData() or a SetData(), is the number of elements to read or write. This used to allow a number up to 255, it now will only allow a number up to 127.
- **New Limitation:** Macros that use extended addressing will only address nodes 0 to 7. They used to address up to 255 nodes.

Version 2.5.2

- **New**: Comments can now be added after any command line by using a double-slash '//' before the comment, for example:

  ```
  For n = 100 Down 0 Step 2     // Cycle through variables
  ```

- **Bug:** The macros do not have the ability to utilize LW:9000 as a pointer for using RWIs (Recipe Word Indexed) and RBIs (Recipe Bit Indexed) registers.

# Appendix C – Sample Macro Code:

### Floating Point to Integer conversion macro:

This macro reads a 32-bit floating point register and will round it to a 16-bit integer number.

```
Sub float pow (short exp)

  // this Sub Returns the number 2
  // raised To the specified power

  int x
  float r
  bool sign = 0

  // handle negative exponents
  If exp < 0 Then
     exp = 0 - exp
     sign = 1
  End If

  Select Case exp
     Case 0
        r = 1
        Break

     Case 1
        r = 2
        Break

     Case Else
        r = 4

        For x = 2 To exp - 1
           r = r * 2
        Next x
  End Select

  // a number raised To a negative exponent
  // is just the reciprocal of the same number
  // raised To the positive exponent
  If sign == 1 Then
     r = 1.00 / r
  End If

  Return r

End Sub



Macro_Command main(  )

   // this main() function takes a 32-bit floating point value
   // rounds it To a 32-bit integer value

   float fpValue = 0
   float Fraction = 0
   float Remain = 0
   int LVal = 0
   int intValue = 0
   short FPSource[2]
   short IntDest[2]
   short Exponent = 0
   short offset = 127
   short Registers = 25
   short MSW = 0
   short LSW = 1
   bool off = 0
   bool Sign = 0

   GetData(FPSource[0] ,LW_Binplc ,0,2)
```

```
   // check the Sign bit, it can cause problems when setting LVal
   If FPSource[1] & 0x8000 Then          // sign bit is Set...
      Sign = 1                           // save the sign bit here
      FPSource[1] = FPSource[1] & 0x7FFF // And clear it here
   End If

   // make a 32-bit version of the FP value
   LVal = (FPSource[1] << 16) + FPSource[0]

   // now deconstruct it:
   // ' fraction' is bits 00-22
   Fraction = 1.000 + ( (LVal & 0x7FFFFF) / 8388608.000 )

   // 'exponent' is bits 23-30
   Exponent = (LVal & 0x7F800000) / 8388608

   // take care of the bias
   Exponent = Exponent - offset

   // calculate FP value
   fpValue = Fraction * ( pow(Exponent) )

   // round the integer value
   intValue = fpValue           // truncate the decimal
   Remain = fpValue - intValue  // save the decimal amount

   // If the decimal part is greater than 0.5, round up
   If Remain >= 0.50000 Then
      intValue = intValue + 1
   End If

   // check the sign
   If Sign == 1 Then
      intValue = 0 - intValue  // negate the value
      Sign = 0               // clear the flag For Next time
   End If

   // write the 32-bit integer value To LW2 And LW3
   IntDest[1] = (intValue & 0xFFFF0000) >> 16
   IntDest[0] = intValue & 0xFFFF

   SetData(IntDest[0] ,LW_Binplc ,2,2)

   // turn off trigger bit
   SetData(off ,LB_Binplc ,5,1)



End Macro_Command
```

## PLC Control of a Recipe Transfer:

This macro allows the PLC to set a bit that will initiate a recipe download. In this example, An Allen Bradley PLC first loads the recipe index pointer into register N7:0. Then the macro is executed by setting an internal bit in the PLC (B3:1/00) to ON. The 100 recipe registers will be downloaded into the top 100 registers of an N15 file.

```
Macro_Command main()
// This macro is an example of how to use the PLC
// to transfer 100 words of recipe memory to the PLC.

int ON = 1
int OFF = 0
short IndexPointer
short RecipeItem[100]
short Alldone = 0

SetData(ON ,LB_Binplc ,500,1)              // Turn local bit 500 ON to indicate
                                           // a recipe transfer is in progress

GetData(IndexPointer ,N7_Binplc ,0,1)      // Get the index pointer from the AB PLC (N7:0)
SetData(IndexPointer ,LW_Binplc ,9000,1)   // Load that into the recipe pointer

// Transfer 100 recipes (RWI:0 to RWI:99):
GetData(RecipeItem[0] ,RWI_Binplc ,0,100)  // Get the first 100 recipes
SetData(RecipeItem[0] ,N15_Binplc ,0,100)  // Store recipes in PLC (starting in (N15:0))


SetData(OFF ,LB_Binplc ,500,1)             // Turn local bit 500 OFF (xfer done)

SetData(Alldone ,B3_Binplc ,100,1)         // Clear bit that called the macro (B3:1/00)

End Macro_Command
```

## Sub routine to convert a BCD number to a Decimal Number:

This subroutine can be added to a macro to convert a BCD value into a standard Binary value.

```
Sub int BCD2DEC (int bcd)

int decval = 0

  decval = 100000
  decval = decval + (((bcd >> 12) & 0xF) * 10000)
  decval = decval + (((bcd >> 8) & 0xF) * 1000)
  decval = decval + (((bcd >> 4) & 0xF) * 100)
  decval = decval + (((bcd) & 0xF) * 10)

  if (((bcd & 0xF) == 2) || ((bcd & 0xF) == 7)) Then
          decval = decval + 5
  End If

return decval

End Sub
```

## Analog Clock:

This is the macro used to generate meter "hands" to simulate a 12-Hour analog clock.

- LW:210 is the Seconds hand – The seconds meter object is set for 0 to 60 span
- LW:211 is the Minutes hand – the minutes meter object is set for 0 to 5959 span
- LW:212 is the Hours hand – the hour meter object is set for 0 to 1159 span
- LB:11 is an AM/PM indicator (0 = AM; 1=PM)
- LB:10 called the macro (place a set-bit object on the screen, set for: Periodic Toggle @1.0 Sec.)

```
Macro_Command main(  )

   short t[3]   // t[0] is seconds, t[1] is minutes, t[2] is hours
   short a
   short b
   bool PM
   bool off = 0

   // get the seconds, minutes, & hours
   GetData(t[0] ,RW_Bcdplc ,60000,3)

   // convert the BCD seconds To a binary value
   a = t[0] & 0xF
   b = (t[0] & 0xF0) >> 4
   t[0] = (b*10) + a

   // convert the BCD minutes To a binary value
   a = t[1] & 0xF
   b = (t[1] & 0xF0) >> 4
   t[1] = (b*10) + a

   // convert the BCD hours To a binary value
   a = t[2] & 0xF
   b = (t[2] & 0xF0) >> 4
   t[2] = (b*10) + a

   // adjust hours For 12-hour format
   If t[2] > 11 Then
      t[2] = t[2] - 12
      PM = 1
   Else
      PM = 0
    End If


   // adjust hours For more resolution:
   // hours will range from 000 To 1259
   t[2] = ( t[2] * 100) + t[1]

   // adjust minutes For more resolution:
   // minutes will range from 000 To 5959
   t[1] = ( t[1] * 100) + t[0]

   // LW210 is seconds, LW211 is minutes, LW212 is hours
   SetData(t[0] ,LW_binplc ,210,3)

   // AM/PM
   SetData(PM ,LB_Binplc ,11,1)

   // reset trigger bit
   SetData(off ,LB_Binplc ,10,1)

End Macro_Command
```

## Text Byte Swapping:

This macro will swap the byte order of two ASCII characters contained in a word (16-bit) register. For example: the characters: 'TEXT' is stored in two registers of the HMI

- LB:500 is the bit used to trigger the macro. This can be a set-bit or toggle switch object to be pushed ON by the user to do the transfer manually, or it can be change to be a PLC bit where the PLC can control when it does the transfer. You set this control bit in the 'PLC Control objects' list.
- LW:200 to LW:209 contains the text in the HMI. The text can be entered with an ASCII input object
- 4x:1 to 4x:10 is a PLC Modbus address that will contain the swapped text after the macro is executed.

```
// HMI to PLC Swap text bytes macro
Macro_Command main(  )

int off = 0

short Data1[10]      //creates an array For 20 characters
int x
int y
int i

   GetData(Data1[0] ,LW_Binplc ,200,10)   //Read 10 words from Local HMI Memory

   For i = 0 To 9
      x = Data1[i] & 0xFF
      y = (Data1[i] & 0xFF00) >> 8

      Data1[i] = (x << 8) + y
   Next i

   SetData(Data1[0] ,4x_Binplc ,1,10)    // Write the switched text To PLC memory


   SetData(off ,LB_Binplc ,500,1)        // This turns off the bit that called the macro

End Macro_Command
```

This macro does the reverse swap of the byte order of two ASCII characters contained in a word (16-bit) register from the PLC to the HMI

```
// PLC to HMI Swap text bytes macro

Macro_Command main(  )

int off = 0

short Data1[12]      //creates an array For 24 characters
int x
int y
int i

   GetData(Data1[0] ,RW_Binplc ,25,12)   //Read 12 words from PLC (change To your PLC address)

   For i = 0 To 11
      x = Data1[i] & 0xFF
      y = (Data1[i] & 0xFF00) >> 8

      Data1[i] = (x << 8) + y
   Next i

   SetData(Data1[0] ,LW_Binplc ,100,12)  // Write the switched text To local memory


   SetData(off ,LB_Binplc ,567,1)        // This turns off the bit that called the macro

End Macro_Command
```

## More Information

For more information regarding macros call Maple Systems technical support @ (425) 745-3229 ext. 258 or email ‘support@maple-systems.com’.

Additional query words: macros, macro language, programming

Keywords: macro, macros, programming

Last Reviewed: 2/23/2006



| FOR DOCUMENT CONTROL USE | | |
|---|---|---|
| Brief Description: EasyBuilder - Macros | | Author: John Goss |
| Changes Approved: No | Status: ☐ WIP | ☐REL |
| Comments: | | |