Application Note NAP170

Byte Reversal

Summary

1. Introduction

This document addresses the programming considerations and strategies necessary to handle the migration of systems with different memory organization architectures, specifically in the context of Programmable Logic Controllers (PLCs). The transition from a Big Endian architecture to a Little Endian one, or vice versa, can cause significant incompatibilities in program logics that perform memory access with explicit addressing or pointers due to differences in memory addresses.

The disparity in memory allocations between the two architectures is a potential source of error. When assigning a value to a variable, such as a two-byte word, the representation in memory differs drastically between Big Endian and Little Endian. For example, the value 255 would be stored as 0x00FF in Little Endian and as 0xFF00 in Big Endian.

This difference becomes critical when program logics depend directly on the memory addresses of bits or bytes. When migrating from one architecture to another, memory addresses change, leading to the breakdown of these preexisting logics. This occurs especially when larger parts of a variable are accessed, such as bits of a word or a word of a variable declared as a double word (DWORD) or long word (LWORD).

On the other hand, it is important to note that this breakdown does not occur when accessing bits of variables declared as byte or boolean. This is because byte reversal affects only at the byte level, not interfering with the order of the bits. Therefore, when accessing a specific bit of a byte variable, the logic remains intact regardless of the architecture.

In summary, migrating between different byte architectures in PLC systems presents significant challenges, especially when it comes to preserving the integrity of existing program logics. Understanding these differences is essential to avoid failures during the upgrade process and to ensure the proper functioning of the systems.

1.1 Visualizing Endianness Differences

Below, we can see the difference in how memory is organized when assigning the value 255 to a variable of type word/byte and "true" to a boolean variable in two different architectures. Consider the following variables initialized with the mentioned values:

```
PROGRAM UserPrg
VAR
    WORD_VAR AT %IW1000: WORD;
    BYTE_VAR AT %IB2000: BYTE;
    BOOL_VAR AT %IX3000.0: BOOL;
END_VAR
```

Access to bit addresses of each variables:

NX3030 (Big-Endian)	NX3008 (Little-Endian)
%IX1000.0 FALSE ; %IX1000.1 FALSE ; %IX1000.2 FALSE ; %IX1000.3 FALSE ; %IX1000.4 FALSE ; %IX1000.5 FALSE ; %IX1000.6 FALSE ; %IX1000.7 FALSE ; %IX1000.8 TRUE ; %IX1000.9 TRUE ; %IX1000.10 TRUE ; %IX1000.11 TRUE ; %IX1000.12 TRUE ; %IX1000.13 TRUE ; %IX1000.14 TRUE ; %IX1000.15 TRUE ; %IX2000.0 TRUE ; %IX2000.1 TRUE ; %IX2000.2 TRUE ; %IX2000.3 TRUE ; %IX2000.4 TRUE ; %IX2000.5 TRUE ; %IX2000.6 TRUE ; %IX2000.7 TRUE ; %IX3000.0 TRUE ; RETURN	%IX1000.0 TRUE ; %IX1000.1 TRUE ; %IX1000.2 TRUE ; %IX1000.3 TRUE ; %IX1000.4 TRUE ; %IX1000.5 TRUE ; %IX1000.6 TRUE ; %IX1000.7 TRUE ; %IX1000.8 FALSE ; %IX1000.9 FALSE ; %IX1000.10 FALSE ; %IX1000.11 FALSE ; %IX1000.12 FALSE ; %IX1000.13 FALSE ; %IX1000.14 FALSE ; %IX1000.15 FALSE ; %IX2000.0 TRUE ; %IX2000.1 TRUE ; %IX2000.2 TRUE ; %IX2000.3 TRUE ; %IX2000.4 TRUE ; %IX2000.5 TRUE ; %IX2000.6 TRUE ; %IX2000.7 TRUE ; %IX3000.0 TRUE ; RETURN

1.2 Example of Byte Reversal for Different Variable Sizes

The image below demonstrates the memory organization and byte reversal for the previously mentioned variables, as well as for variables of type DWORD and LWORD in each architecture when storing the word "CPUNEXTO":

MSB ← Little-endian → LSB								
BYTE	%QB7	%QB6	%QB5	%QB4	%QB3	%QB2	%QB1	%QB0
	C	P	U	N	E	X	T	O
WORD	%QW3		%QW2		%QW1		%QW0	
	CP		UN		EX		TO	
DWORD	%QD1				%QD0			
	CPUN				EXTO			
LWORD	%QL0							
	CPUNEXTO							
MSB ← Big-endian → LSB								
BYTE	%QB0	%QB1	%QB2	%QB3	%QB4	%QB5	%QB6	%QB7
	C	P	U	N	E	X	T	O
WORD	%QW0		%QW2		%QW4		%QW6	
	CP		UN		EX		TO	
DWORD	%QD0				%QD4			
	CPUN				EXTO			
LWORD	%QL0							
	CPUNEXTO							

2. Scenarios Where Byte Reversal Problems Can Occur in PLCs

As previously mentioned, the byte reversal problem can potentially occur when upgrading a PLC project from one using ARM processors (Little Endian) to one using Power PC (Big Endian), or vice versa, in applications that involve specific programming logics. The only scenario where different architectures exist and project updates are permitted is within the NX3xxx series of PLCs. Below are the controllers in this family that use each of the architectures:

Power PC (Big-Endian)	ARM (Little-Endian)
NX3003	NX3008
NX3004	
NX3005	
NX3010	
NX3020	
NX3030	

Therefore, any update involving PLCs from the first column to one from the second, or vice versa, is subject to potential endianism errors depending on how the application was developed.

3. Identifying and Resolving Endianness Issues in Projects

Before delving into the problematic memory access methods that lead to endianism issues, it's crucial to understand that whether the memory access the user is performing is problematic or not depends on the context in which that address is located within the controller's memory. Understanding this will aid us in investigating which memory accesses may be problematic or not.

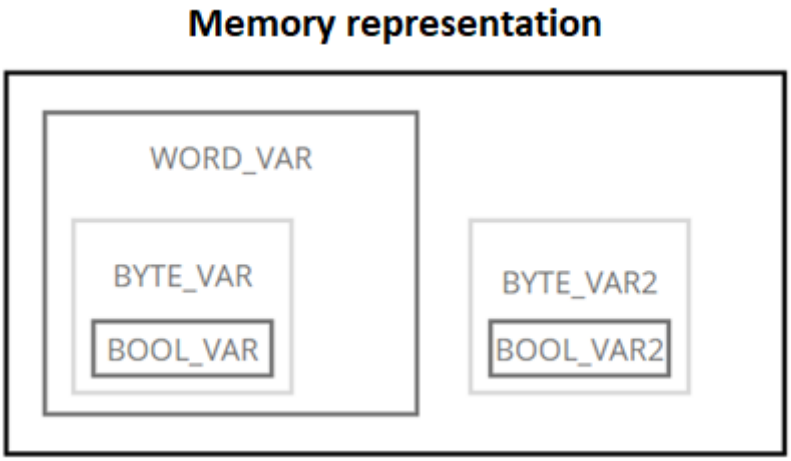
3.1 Variable Context

Suppose we have the following variables declared:

```
PROGRAM UserPrg
VAR
    WORD_VAR AT %IW0: WORD;
    BYTE_VAR AT %IB0: BYTE;
    BOOL_VAR AT %IX0.0: BOOL;

    BYTE_VAR2 AT %IB10: BYTE;
    BOOL_VAR2 AT %IX10.0: BOOL;
END_VAR
```

The following image provides a simple depiction of the context of each variable within the controller's memory:



We can observe that in this context, BYTE_VAR and BOOL_VAR are contained within WORD_VAR (which will have its bytes reversed in the architecture change), therefore, they are subject to endianism.

BYTE_VAR2 and BOOL_VAR2 are not contained within any variable larger than 1 byte, therefore, they are not affected by byte reversal. Similarly, WORD_VAR can also be accessed without issues, as it is not encompassed by any variable larger than itself.

3.2 Memory Access Methods Prone to Endianness Issues

When analyzing projects submitted by clients, three forms of direct memory address access used in the project have been identified as potentially causing endianism issues:

- Use of the "AT" Directive on a Symbolic Variable:
- Direct Use of Memory Addresses in Logic;
- Utilization of Symbolic Variables in IO Mapping Screens.

3.2.1 Use of the "AT" Directive on a Symbolic Variable

Example:

```
VAR1 AT %IW0: WORD;
```

If, after analyzing the context of the variables, it is identified that there are symbolic variables that may be subject to endianism issues, we can solve this problem in three ways.

a) Access to variable bits via "dot notation"

This type of access abstracts the controller's architecture. Therefore, when accessing bit 0, for example, we will obtain the same result in both controllers.

Example:

Suppose a variable `WORD_VAR AT %IW1000 : WORD;` initialized with the value 255.

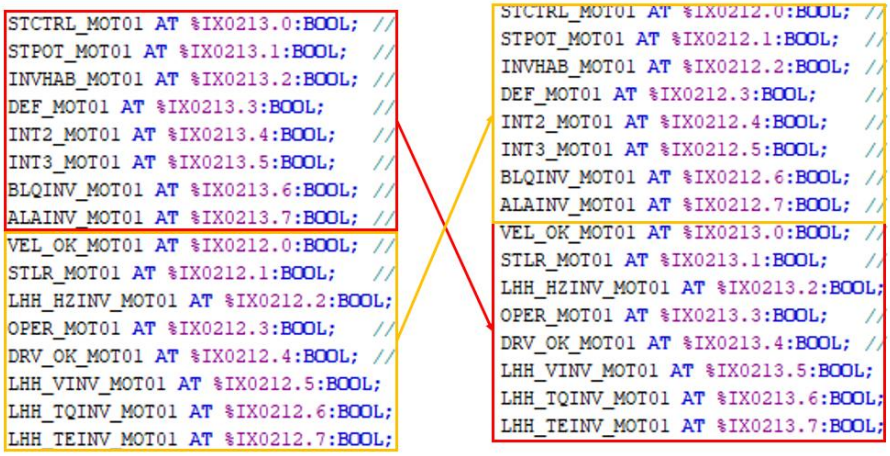
Here is the result of each bit for each of the controllers using the syntax ".<bit address>":

NX3030 (Big-Endian)	NX3008 (Little-Endian)
WORD_VAR 255 .0 TRUE ; WORD_VAR 255 .1 TRUE ; WORD_VAR 255 .2 TRUE ; WORD_VAR 255 .3 TRUE ; WORD_VAR 255 .4 TRUE ; WORD_VAR 255 .5 TRUE ; WORD_VAR 255 .6 TRUE ; WORD_VAR 255 .7 TRUE ; WORD_VAR 255 .8 FALSE ; WORD_VAR 255 .9 FALSE ; WORD_VAR 255 .10 FALSE ; WORD_VAR 255 .11 FALSE ; WORD_VAR 255 .12 FALSE ; WORD_VAR 255 .13 FALSE ; WORD_VAR 255 .14 FALSE ; WORD_VAR 255 .15 FALSE ; BYTE_VAR 255 .0 TRUE ; BYTE_VAR 255 .1 TRUE ; BYTE_VAR 255 .2 TRUE ; BYTE_VAR 255 .3 TRUE ; BYTE_VAR 255 .4 TRUE ; BYTE_VAR 255 .5 TRUE ; BYTE_VAR 255 .6 TRUE ; BYTE_VAR 255 .7 TRUE ; BOOL_VAR TRUE ; RETURN	WORD_VAR 255 .0 TRUE ; WORD_VAR 255 .1 TRUE ; WORD_VAR 255 .2 TRUE ; WORD_VAR 255 .3 TRUE ; WORD_VAR 255 .4 TRUE ; WORD_VAR 255 .5 TRUE ; WORD_VAR 255 .6 TRUE ; WORD_VAR 255 .7 TRUE ; WORD_VAR 255 .8 FALSE ; WORD_VAR 255 .9 FALSE ; WORD_VAR 255 .10 FALSE ; WORD_VAR 255 .11 FALSE ; WORD_VAR 255 .12 FALSE ; WORD_VAR 255 .13 FALSE ; WORD_VAR 255 .14 FALSE ; WORD_VAR 255 .15 FALSE ; BYTE_VAR 255 .0 TRUE ; BYTE_VAR 255 .1 TRUE ; BYTE_VAR 255 .2 TRUE ; BYTE_VAR 255 .3 TRUE ; BYTE_VAR 255 .4 TRUE ; BYTE_VAR 255 .5 TRUE ; BYTE_VAR 255 .6 TRUE ; BYTE_VAR 255 .7 TRUE ; BOOL_VAR TRUE ; RETURN

Disadvantages of this approach: It does not allow us to create different variables to represent each of the bits, which is a common practice in controller programming.

b) Manually inverting addresses

This approach involves identifying the addresses that, within the context, are susceptible to endianism, and then manually swapping the even addresses with the odd ones, as shown in the example below:



c) Inverting addresses using a script

To facilitate the task of inverting addresses for a large number of variables, a Python script has been developed that automatically inverts all variable addresses placed within a GVL named "InvertBytes."

To create the script, follow these steps:

i. Open Notepad;
ii. Copy the script content, which can be seen in Chapter 4;
iii. Click on "File" and then "Save as";
iv. Select the type as "All files";
v. Write a name for the file and add the extension ".py" to the end of the name;
vi. Click on "Save".

To use it, follow these steps:

i. Create a GVL named "InvertBytes";
ii. Copy all the code containing the inverted addresses into this GVL;
iii. Click on "Tools > Scripting > Run script file" and locate the script;
iv. Copy the result generated by the script into the "InvertBytes" GVL to the source GVL;

CAUTION:
The only function of the script is to convert even addresses to the next odd address and odd addresses to the previous even address for bit and byte mappings. Make sure the code pasted in the GVL should indeed be inverted by analyzing its contexts!

d) Using the "ROL" (Rotate Left) and "ROR" (Rotate Right) functions

These functions are used to shift a specified number of bits to the right (ROR) or to the left (ROL), which can be useful for resolving endianism issues. Below are examples of usage:

```
(*The main code inserted by the user and executed
PROGRAM UserPrg
VAR
    VAR1 AT %IW0: WORD;
    VAR2: WORD;
END_VAR
```

```
VAR2 := ROL(VAR1, 8); // Moves 8 bits to the left

VAR2 := ROR(VAR1, 8); // Moves 8 bits to the right
```

3.2.2 Direct Use of Addresses in Logic

This approach involves directly using addresses within the code to perform certain logic, as we can see in the example below:

```
IF %IX0.00 = TRUE THEN
    //do something
END_IF;
```

To address the direct use of addresses within the logic, we can utilize the same methods described in the sections "b) Manually Inverting Addresses" and "c) Inverting Addresses Using a Script" mentioned above. However, because it is considered poor practice to use addresses without an associated variable and because it significantly hampers traceability of the context in which the address is used (for example, if it comes from a communication driver), it is ideal to avoid using this type of code and refactor it when already present.

3.2.3 Use of Symbolic Variables in IO Mapping Screens

This type of access creates issues when assigning tags or symbolic variables on any mapping screen, regardless of the communication driver being used (Modbus, EtherNET/IP, PROFINET, etc.).

Consider the example below:

We have a PO7079 counter module connected via PROFIBUS to two controllers, one Power PC and the other ARM. In the Power PC controller, the bit that enables counting on the module is located at address %QX19.7. However, when updating the project to an ARM controller, the address to enable counting changed to %QX18.7. Therefore, setting the tag "Enable_Counter" will no longer work.

PO7079 (Power PC)

Variable	Mapping	Channel	Address	Type	Unit	Description
		Output1	%QW18			
		Word0	%QW18	WORD		
		Bit0	%QX18.0	BOOL		
		Bit1	%QX18.1	BOOL		
		Bit2	%QX18.2	BOOL		
		Bit3	%QX18.3	BOOL		
		Bit4	%QX18.4	BOOL		
		Bit5	%QX18.5	BOOL		
		Bit6	%QX18.6	BOOL		
		Bit7	%QX18.7	BOOL		
		Bit8	%QX19.0	BOOL		
		Bit9	%QX19.1	BOOL		
		Bit10	%QX19.2	BOOL		
		Bit11	%QX19.3	BOOL		
		Bit12	%QX19.4	BOOL		
		Bit13	%QX19.5	BOOL		
		Bit14	%QX19.6	BOOL		
Enable_Counter		Bit15	%QX19.7	BOOL		

PO7079 (ARM)

Variable	Mapping	Channel	Address	Type	Unit	Description
		Output1	%QW18			
		Word0	%QW18	WORD		
		Bit0	%QX18.0	BOOL		
		Bit1	%QX18.1	BOOL		
		Bit2	%QX18.2	BOOL		
		Bit3	%QX18.3	BOOL		
		Bit4	%QX18.4	BOOL		
		Bit5	%QX18.5	BOOL		
		Bit6	%QX18.6	BOOL		
		Bit7	%QX18.7	BOOL		
		Bit8	%QX19.0	BOOL		
		Bit9	%QX19.1	BOOL		
		Bit10	%QX19.2	BOOL		
		Bit11	%QX19.3	BOOL		
		Bit12	%QX19.4	BOOL		
		Bit13	%QX19.5	BOOL		
		Bit14	%QX19.6	BOOL		
Enable_Counter		Bit15	%QX19.7	BOOL		

Note: Notice that the inversion only occurs because these bits are contained within a word! This does not happen for mappings of the byte or bool type!

3.3 Communication Drivers

It is worth noting that all endianism issues that occur in the programming practices mentioned above, with user-created variables, will also occur for variables mapped in any communication driver (EtherNET/IP, Modbus, PROFINET, etc.).

4. Attachments

4.1 Python Script for Byte Swapping

```python
import re

# Function called for each match found
def replace(match):
    length = len(match.group(0))
    match_num = int(match.group(0))

    if match_num % 2 == 0:
        result = match_num + 1
    else:
        result = match_num - 1

    return str(result).zfill(length)

# Getting the contents of the "INVERTBYTES" GVL
proj = projects.primary
gvl_invert_bytes = proj.find('INVERTBYTES', recursive=True)[0]
content = gvl_invert_bytes.textual_declaration.text

# Defining search patterns
patterns = [r'(?<=%IX)[0-9]*', r'(?<=%IB)[0-9]*',
            r'(?<=%QX)[0-9]*', r'(?<=%QB)[0-9]*',
            r'(?<=%MX)[0-9]*', r'(?<=%MB)[0-9]*']

# Converting content
new_content = content
for pattern in patterns:
    new_content = re.sub(pattern=pattern, repl=replace, string=new_content)

# Replacing the contents of the GVL
gvl_invert_bytes.textual_declaration.replace(new_content)
system.ui.info('Logic converted!')
```