Motion Control Products

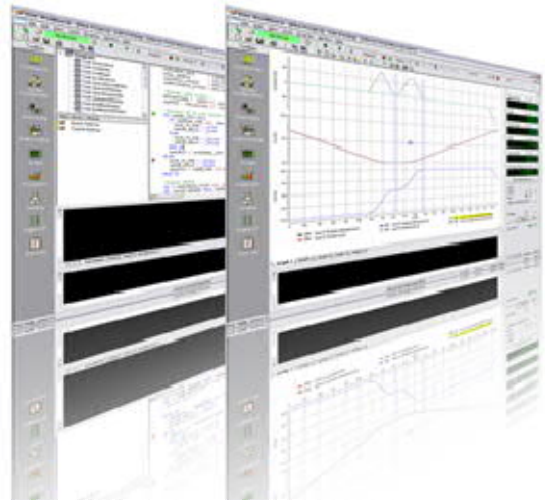
Application note
Simple PLC functionality

AN00167

Rev A EN

Mint is an easy to use high-level programming language rich in features, including facilities to write modular, block-structured programs. The Mint language includes subroutines, functions, tasks, structures (user-defined data types), conditional statements, looping statements, etc.

Mint also provides an extensive range of specialized functions to interface to the hardware of Baldor controllers and includes a variety of keywords to vastly simplify the implementation of complex motion control solutions.



Introduction

Mint is a sequential language (i.e. a particular line of code is not executed until the previous line has completed execution) that also provides the user with the ability for parallel processing (via multi-tasking) and event/interrupt driven operation (e.g. via digital input events) making the language as a whole incredibly flexible and particularly suited to machine and motion control applications

.

Traditional Programmable Logic Controllers (PLCs) are often programmed in a ladder logic style language. Typically these environments are cyclic in nature. The PLC reads the input image, processes all of the code in the program and finally sets the output image according to the logical results. After performing local housekeeping tasks it then restarts this cycle. The time to complete one pass of the PLC program is often known as the ‘cycle time’ or ‘scan time’ of the PLC.

Often the PLC has limited interrupt capability and requires special high-speed counter/encoder cards to deal with signals of duration shorter than the PLC cycle time.

This Application Note aims to describe how typical PLC data types and functions may be implemented in Mint. For the purposes of this Application Note we will use the Allen Bradley™ SLC500™ instruction set as a reference with screenshots from Rockwell Software’s RSLogix® programming software. The principles are of course exactly the same for many other PLC types.

Data Types

For the purposes of comparison we will define a lot of Mint variables in the examples within this Application Note using similar names to those used by the PLC (e.g. N7 for an integer variable). In reality there are very little restrictions on how variables are named in Mint and in a real application we would give these variables more meaningful names to make the program easier to read and understand. This can reduce the amount of comments required to document the program and help to speed program development in comparison with some PLC platforms.

e.g. nProductCount (an integer value storing the number of products produced)
fProductLength (a floating point value storing the length of a product being produced)

Refer to Application Note AN00104 for Baldor's recommended coding practices. This document includes suggestions on how to name your Mint variables.

Integer

The SLC500™ range of PLC supports 16 bit integer data (range of -32768 to +32767). Integer files must be created by the user and then assigned using the syntax Nf:e/b (where N indicates an integer file, f is the file number, : is the element delimiter, e is the element number [0-255], / is the bit delimiter and b is the bit number [0-15]).

Examples:

N7:4 – Word 4 of the N7 integer data area

N7:6/3 – Bit 3 of word 6 of the N7 integer data area

Mint controllers support 32 bit integer data (range -2147483648 to 2147483647). Integer data must be created by the user using the Mint DIM keyword and specifying As Integer in the declaration. Blocks of integer data can be defined if required using a Mint array (the array dimension determines the number of array elements provided).

Examples:

```
Dim nCounter As Integer     'Declare a single 32 bit integer value
Dim N7(0 to 9) As Integer   'Declare a group/array of 10 integers
N7(4) = 123                 'Set element 4 of the N7 integer array to 123
```

Mint integers declared in this way don't allow bit level access in the addressing syntax but it is still possible to use logical operands to read/write bits within an integer value.

Examples:

```
OUTX(0) = N7(8) & 01000   'Output 0 mimics Bit 3 of element 8 of the N7 integer array
N7(2) = N7(2) Or 01010    'Set bits 1 and 3 in element 2 of the N7 integer array
```

We will discuss other Mint style bit level data types and operations later in this document.

Float

The SLC500™ range of PLC supports 32 bit floating point data. Float files must be created by the user and then assigned using the syntax Ff:e (where F indicates a float file, f is the file number, : is the element delimiter, e is the element number [0-255]).

Example:

F8:1 – Word 1 of the F8 floating point data area

Mint controllers support 32 bit floating point data (range of $\pm 5.877 \times 10^{-39}$ to $\pm 3.4 \times 10^{38}$ on NextMove products). Float data must be created by the user using the Mint DIM keyword and specifiying As Float in the declaration. Blocks of float data can be defined if required using a Mint array (the array dimension determines the number of array elements provided)

Examples:

```
Dim fActualSpeed As Float   'Declare a single 32 bit float value
Dim F8(0 to 9) As Float     'Declare a group/array of 10 floats
F8(6) = 123.456             'Set element 6 of the F8 float array to 123.456
```

Bit Files

The SLC500™ range of PLC supports bit files where bits can either be addressed as a complete word or at a single bit level. Bit files must be created by the user and then assigned using the syntax Bf:e/b (where B indicates a bit file, f is the file number, : is the element delimiter, e is the element number [0-255], / is the bit delimiter and b is the bit number [0-15]).

Examples:

B3:5 – Word 5 (all 16 bits) of the B3 bit file

B3:5/13 – Bit 13 of word 5 of the B3 bit file

We have already seen how an individual bit (or bits) within an integer value may be accessed in Mint using logical operands. Additionally it is common to declare a Mint integer value and use it to store a binary (0 or 1) result.

With the release of Mint target format 14 (supported by firmware revision 5615 onwards on the Nextmove e100 platform) it becomes possible to create bit level data types using the new **BITFIELD** keyword. This keyword is very powerful as it allows the user to specify not only individual bits but also any combination of bits.

Example:

```
Bitfield Int32
  All As 0 to 31
  Bit0 As 0
  Bit1 As 1
  Bit2 As 2
  Bit3 As 3
  Nibble1 As 4 to 7
  Byte1 As 8 to 15
  Word2 As 16 to 31
End Bitfield

Dim B3_0 As Int32
```

In this example we have bit level access for bit 0 to 3 and can access other specific parts of the 32 bit value directly (either as 4, 8 or 16 bit values).

We could use this in the following ways…

```
OUTX(0) = (B3_0.Bit0 Or B3_0.Bit1) And B3_0.Bit2
B3_0.Word2 = 188 (this would set the top 16 bits to 0000000010111100)
B3_0.Byte1 = 01110000
```

When accessing specific areas of the bitfield the other data remains intact (providing data areas don't overlap).

We can of course also declare arrays of bitfields (to emulate large PLC bit type data tables)…

```
Dim B3(0 to 99) As Int32
B3(4).Bit0 = 1
B3(83).Bit2 = 0
```

User Defined Data Types

Some PLC's allow the user to define their own data types (i.e. structures). This is also possible in Mint using the **STRUCTURE** keyword. Structures provide a means of grouping data together, often of different types, into a single named entity for convenient handling.

Like an array, a structure is an aggregate (a collection) of values, but unlike an array the data type of each member may be different. Once a structure has been defined, it may be used as a template for defining other variables

Example:

```
Structure Product
  fLength As Float
  fWidth As Float
  fSpeed As Float
  nTargetCount As Integer
  sName As String * 20
End Structure

Dim Recipes(0 To 9) As Product

Recipes(3).sName   = "Strawberry"
Recipes(3).fLength = 100.1
Recipes(3).fWidth  = 50.2
Recipes(3).fSpeed  = 99.9
Recipes(3).nTargetCount = 10000
```

Ladder Logic

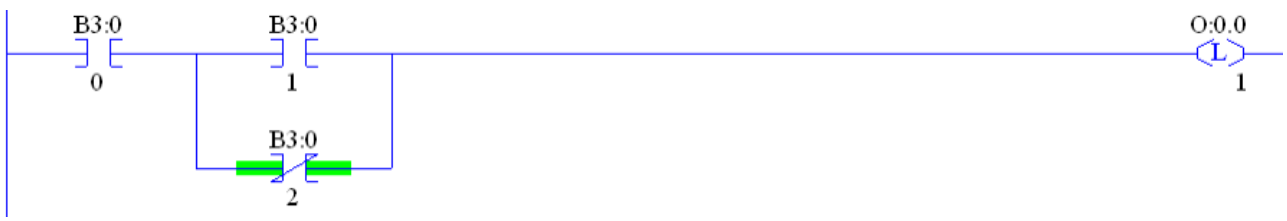
The following examples illustrate how common PLC functions can be implemented in Mint very easily. In each case a sample ladder rung is shown followed by the equivalent Mint code. All bit addresses have been implemented as Mint integer data types but these could equally be members of a bitfield as described earlier.

OTE (Output Energise)


```
Dim B3_0_0 As Integer
Dim B3_0_1 As Integer
Dim B3_0_2 As Integer

OUTX(1) = B3_0_0 AND (B3_0_1 Or B3_0_2)
```

OTL (Output Latch)




```
Dim B3_0_0 As Integer
Dim B3_0_1 As Integer
Dim B3_0_2 As Integer
```

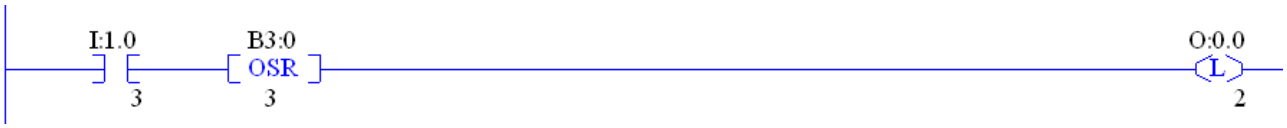
If B3_0_0 AND (B3_0_1 Or B3_0_2) Then OUTX(1) = _ON

OTU (Output Unlatch)


```
Dim B3_0_0 As Integer
Dim B3_0_1 As Integer
Dim B3_0_2 As Integer

If B3_0_0 AND (B3_0_1 Or B3_0_2) Then OUTX(1) = _Off
```

OSR (One Shot Rising)

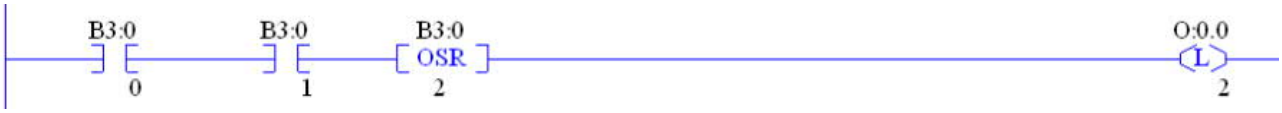



```
'Configure input for interrupt on rising edge
INPUTMODE(0)          = 000001000
INPUTPOSTRIGGER(0) = 000001000

Event IN3
  OUTX(2) = _On
End Event
```

In the case of digital inputs we can take advantage of Mint's extensive interrupt capability to provide a flexible, rapid, easy to use solution.

It requires a few more lines of code to recreate an OSR function when using variables instead of an input but it's still very straight-forward…




```
Dim B3_0_0 As Integer
Dim B3_0_1  As Integer
Dim B3_0_2 As Integer = _false

Loop
 If B3_0_0 And B3_0_1 Then
   If NOT B3_0_2 Then
     OUTX(2)= _true
     B3_0_2 = _true
   End If
 Else
   B3_0_2 = _false
 End If
```

```
End Loop
```

TON (Timer On Delay)

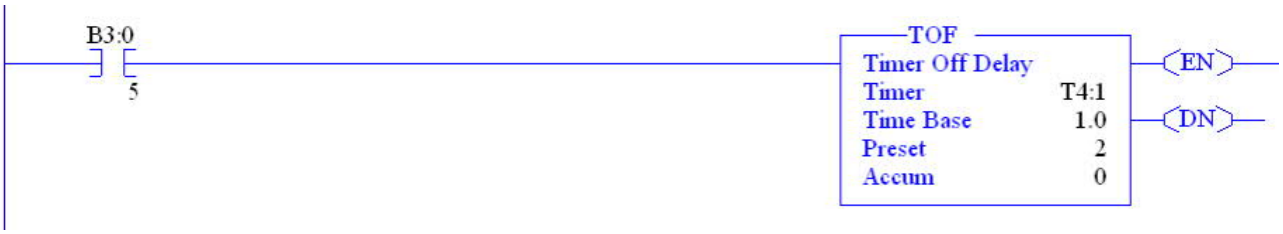

```
Dim B3_0_4 As Integer

If B3_0_4 Then
  If TaskStatus(T4_0) = _tskTerminated Then Run T4_0
Else
  End T4_0
End If

Task T4_0
   Dim bDone As Integer = _False
   Wait(3000)
   bDone = _True
   Pause _False
End Task
```

Timer timing can be derived from: (TaskStatus(T4_0) <> _tskTerminated) AND (T4_0::bDone = _False)
Timer done is defined by T4_0::bDone = _True
Timer enabled is defined by TaskStatus(T4_0) <> _tskTerminated.
Note that Mint also offers the ability to define variables as a TIME data type resulting in the ability to measure time etc… without the need to create a timer task. The WAIT instruction also allows program execution within a specific module to be suspended for a given duration (in ms) without the need to create any sort of time related variables/tasks.

TOF (Timer Off Delay)




```
Dim B3_0_5 As Integer

If Not B3_0_5 Then
  If TaskStatus(T4_1) = _tskTerminated Then Run T4_1
Else
  End T4_1
End If

Task T4_1
   Dim bDone As Integer = _False
   Wait(3000)
   bDone = _True
```

```
  Pause _False
End Task
```

Timer timing can be derived from: (TaskStatus(T4_1) <> _tskTerminated) AND (T4_1::bDone = _False)
Timer done is defined by T4_1::bDone = _True
Timer enabled is defined by TaskStatus(T4_1) <> _tskTerminated

Contact us

For more information please contact your
local ABB representative or one of the following:

new.abb.com/motion
new.abb.com/drives
new.abb.com/drivespartners
new.abb.com/PLC