Motion Control Products

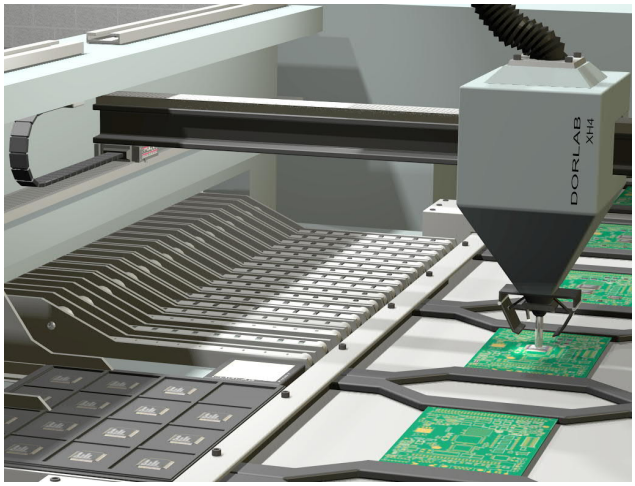
Application note
Pick and place example

AN00179

Rev C (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 ABB 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.
This application note provides an insight into some of the techniques which can be used to easily implement a pick and place machine such as that illustrated by the picture above.

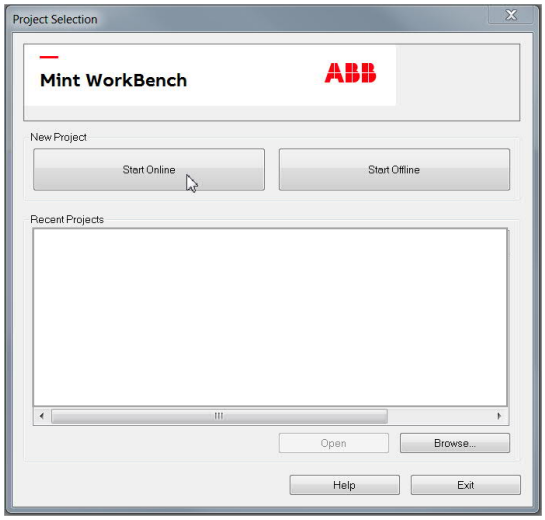
Application

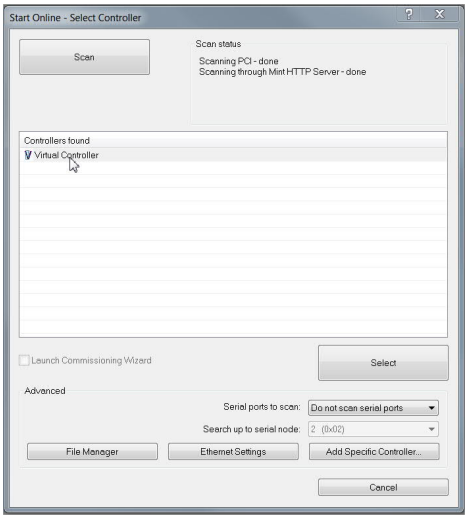
This system presumes a circuit board mounting type application. The mechanical components of this system would typically be the axes form a Gantry or 'H-Bridge' arrangement where two motors are used to control the Y axis position and one motor controlling the X axis cross travel. Its normal that the picker will be pneumatically operated (valves for up/down, vacuum on/off and grip in/out) with a ratchet type indexing system for the IC holders (again actuated via a solenoid valve)

The code that accompanies this application note is designed to run on the Mint virtual controller (part of Mint Workbench) allowing the user to run the code and simulate the machine without any real hardware. But in a real application the code could be run in a Nextmove controller if needed. The sample code assumes there are five IC racks and each PCB only uses up to five of each of these devices (although this is easily expanded via the use of Mint constants described later).

Using the virtual controller

Start Mint Workbench and click on Start Online.
Mint Workbench will find a virtual controller…





Highlight the virtual controller and click on the Select button…you will be taken online to the virtual controller. This provides control of 16 Virtual Axes and several virtual I/O banks. Mint programs can be downloaded and run on the virtual controller allowing applications to be developed without the need for external hardware. It is even possible to connect a host application to the virtual controller (via. “setVirtualControllerLink”).

From the File menu select Open File… and navigate to the sample Mint program included with this application note. Open this and the program will appear in the Editor window. To download and run this on the virtual controller select Program>Compile, Download and Run (or keyboard shortcut CTRL+F5).

The Mint Workbench Spy Window ‘Monitor’ tab can be used to select position of the controlled axes (0 to 4). As the application executes you will see these axes moving to the programmed locations.

Example code

The following sections cover the mint programs structure and functionality.

Axis configuration

The example application uses the following axes…

a. X axis (simulating the real X axis)
b. Master Y axis (in a real application this would actually be configured as **a virtual axis** – the two physical Y axes are slaved off of this axis. As both the real axes follow the same source they both have the same lag and therefore exhibit no lag with respect to each other, thereby ensuring there is no twisting effect on the gantry)
c. Y1 and Y2 axes (simulating the real Y axes)
d. I axis (simulating the indexing conveyor axis transporting the PCBs)
e. F axis (simulating the infeed axis carrying fresh boards to the conveyor)

These axes are configured in the Mint Startup block.
The Mint keyword SCALEFACTOR is used to set user units for each axis. The example assumes the motors are fitted with encoders and gearing which equate to 1000 counts per mm. By setting the SCALEFACTOR to 1000 the user unit for the linear axes becomes mm.

Similarly, the conveyor and infeed axes are assumed to be rotary motors. The indexing conveyor has 300 counts/steps per mm of linear travel and the infeed axis has 100 counts/steps per mm of linear travel.

As a result of setting scale factors the SPEED and ACCEL/DECEL settings become mm/sec and mm/sec^2 (e.g. the X axis accelerates at 5000mm/sec^2 to reach a speed of 500mm/sec in 100ms).

To configure the two Y axis linear motors to follow a virtual Y axis we need simply tell these axes what they're following (a demand position) and which axis/channel they're following (the virtual Y axis)…

```
'Configure how the Y axes follow their master reference...
MASTERSOURCE([_axY1,_axY2])  = _msDEMAND;
MASTERCHANNEL([_axY1,_axY2]) = _axYM;
```

Note: how the ";" character is used to 'apply this setting to all listed axes'.

Once the axes are enabled the physical Y axes can be told to track the virtual Y axis at a 1:1 ratio using the Mint FOLLOW keyword…

```
'Start the follow...
FOLLOW([_axY1,_axY2]) = 1;
```

Note: For the remainder of the application the code need only issue target positions for the X axis and the Virtual master Y axis, the real Y axes will simply follow the virtual Y axis.

Program constants

To make the code easier to read and to allow the program to be easily updated later some constants are used to define certain aspects of the application. The axis numbers are all allocated constants as well as the number of IC racks and the maximum number of each IC per board. By utilizing these constants, instead of 'hard coding' numerical data later in the program, we need only change the constant declaration should we wish to increase the machine capacity.

```
Const _nNoOfChipTypes As Integer = 5
Const _nMaxNoOfChips As Integer = 5
```

Variable declarations

The main data for this application is stored in three arrays:

- Details of how many of each chip are required for the PCB
- The XY co-ordinates from where each chip should be picked (this will be a single co-ordinate for each chip type)
- The XY co-ordinates for where each chip should be placed

It would be common for this data to be transferred to the application from a host PC that contains the manufacturing information for the PCB (e.g. via the Mint ActiveX control). For the purposes of this example we have hard coded the array contents.

```
Dim nChipQtys(1 To _nNoOfChipTypes) As Integer
Dim fPickLocs(0 To 1,1 To _nNoOfChipTypes) As Float
Dim fPlaceLocs(0 To 1,1 To (_nNoOfChipTypes * _nMaxNoOfChips)) As Float
```

The pick and place arrays are two dimensional allowing X and Y co-ordinates for each operation to be stored in a single array. To illustrate this, consider the fPickLocs array. This array stores the XY co-ordinates the axes should move to when retrieving an IC from a storage rack. As there are five racks in this example (_nNoOfChipTypes) then we need to store five separate co-ordinates (the pick location at the end of each rack). The following table illustrates how the array contents are arranged to store this data:

	Location 1	Location 2	Location 3	Location 4	Location 5
X Axis	10 [0,1]	10 [0,2]	10 [0,3]	10 [0,4]	10 [0,5]
Y Axis	200 [1,1]	250 [1,2]	300 [1,3]	350 [1,4]	100 [1,5]

In square brackets we have shown how a specific element of the array would be addressed (e.g. the Y axis co-ordinate for location 3 of 300 would be element 1,3 of the array). By declaring the array indices for the first dimension as 0 to 1 we are able to match these to the axis number. By declaring the array indices for the second dimension as 1 to _nNoOfChipTypes we are able to match these to the location number.

Therefore, generically each element can be represented as (axis number, location number). This allows us to iterate through the array contents using For….Next loops based on these two parameters.
If we examine the code for moving to the Pick location in detail, we can see this in operation…

```
'Move to pick location...
VECTORA(_axX,_axYM) = fPickLocs(0,i),fPickLocs(1,i)
GO(_axX,_axYM)
Pause IDLE([_axX,_axYM])
```

In this code 'i' is taken from a 'For…Next' loop and represents each of the chip types (1 to 5). The VECTORA command performs an interpolated move (in this case on the X and Y master axes) that ensures the two axes start and complete the move at the same time. This move is triggered by issuing GO. We can wait for the move to complete via the PAUSE IDLE statement.

The same principle is used for the place location data. This time the number of elements in the second dimension is determined by the maximum possible number of chip types multiplied by the maximum possible number of each chip to be placed (i.e. 5 * 5 = 25).

Pick and place subroutine

To modularize the code a common routine is used to manipulate the solenoid valves on the pick head. The Mint Define keyword is also used to give meaningful names to digital outputs (again to make the code easier to read and easy to modify later – e.g. if the output assignment changes).

```
Define opPICK = OUTX(0)
Define opGRIP = OUTX(1)
Define opVAC = OUTX(2)
```

The following table illustrates the operating sequence for a pick and a place:

Order	Output	Pick routine	Place routine
1st	opPICK	Pick head down	Pick head down
2nd	opVAC	Vacuum on	Vacuum off
3rd	opGRIP	Grip in	Grip out
4th	opPICK	Pick head up	Pick head up

If we look at this in detail, we can see the sequences are generally the same other than for a pick the vaccum and grip are turned on (1) and for a place they are turned off (0). We can therefore use a subroutine to group together the common code and use a parameter to pass a 0 or 1 as required to the routine.

```
Sub doPicker (ByVal bGrip As Integer)
  opPICK = _on
  Wait(100)
  opVAC = bGrip
  opGRIP = bGrip
  Wait(100)
  opPICK = _off
  Wait(100)
End Sub
```

Again to make the code easier to read we have defined two Mint constants instead of using the numbers 0 and 1..

```
Const _nPlace As Integer    = 0
Const _nPick As Integer     = 1
```

So, to pick up a device we use…

```
doPicker _nPick
```

…and to place a device we use…

```
doPicker _nPlace
```

Chip carrier index subroutine

Digital outputs 3 to 7 are used to power the solenoid valves used to shuttle an IC to the end of a chip carrier rack. Pulsing an output for 100ms is assumed to be sufficient to perform an index.
Again by using a subroutine for this operation the code can be modularized and a parameter can be used to derive the output to be operated from the supplied row number.

```
Sub doIndex(ByVal nRow As Integer)
  'The chip carrier indexes by pulsing the digital
  'output associated with a particular row...
  PULSEOUTX(2+nRow) = 100
End Sub
```

The PULSEOUTX instruction pulses a digital output for a specified time (in ms).

Indexing the conveyor and feeding a blank board

Both axes perform a relative move in this example. The Mint MOVER instruction is used for this. In the case of the indexing conveyor the move must not occur if the infeed axis is still in motion.
This interlock is achieved by waiting for the infeed axis to complete any motion it is performing (i.e. it is IDLE) before issuing the MOVER on the indexing conveyor…

```
Pause IDLE(_axF)
MOVER(_axI) = 600
GO(_axI)
```

The indexing conveyor moves 600mm in this example.

Once the indexing conveyor has finished moving the infeed conveyor can feed in a new blank board. Because the indexing conveyor is stationary at this point in the cycle we do not need to wait for the infeed to complete its move before continuing with the pick and place operations…

```
MOVER(_axF) = 650
GO(_axF)
```

The boards must be fed 650mm by the infeed to place them onto the Indexing conveyor.

Contact us

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

new.abb.com/drives/low-voltage-ac/motion
new.abb.com/drives
new.abb.com/channel-partners
new.abb.com/plc