



NextMove PCI ✓ NextMove BX ✓ MintDrive II ✓ Flex+Drive II ✓

## Overview

This application note describes Baldor’s recommended practices for coding MintMT applications. Adopting these practices ensures that all Mint application developers write code with a consistent style thereby improving quality, portability, readability and maintainability.

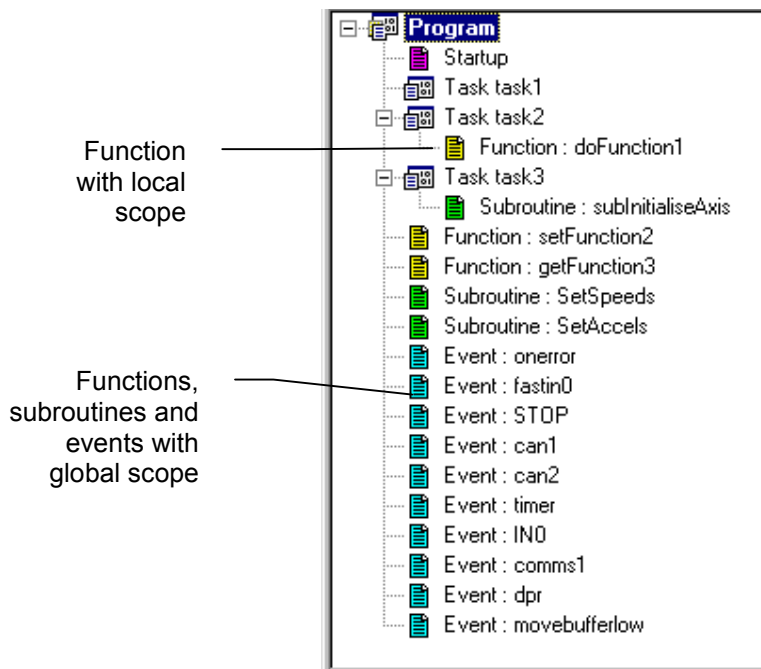
## Program File Structure

MintMT program layout should conform to the following overall layout:

- Auto keyword (for automatic execution on power-up if applicable)
- Program Header
- Macro definitions
- Global constants
- Global variables
- Main Parent Task
- Startup block
- Child Tasks
- Global Functions
- Global Subroutines
- Events (in priority order – starting with the highest)

Note that functions and subroutines may have global scope (i.e. can be called from any task) or scope local to a specific task.

An example of this structure and the effects of global and local scope on functions and subroutines, as illustrated by the Workbench v5 Program Navigator, are shown below:





## Program Header

Include a program header to enable others to easily identify the purpose of the application. This header can also be used to document subsequent revisions to the program. It is good practise to document the firmware revision used as part of a known 'working' system.

```

\=====
\
\ Customer      :
\ Project      :
\ FileName     :
\ Date        :
\ Author       :
\ Firmware Rev:
\
\ Description  :
\
\ Revision History:
\=====
    
```

## Comments/Documentation

The program comments should describe what is happening and how it is being done. Avoid comments where the functionality is clear from the code. Short comments should be *what* comments, such as "compute mean value", rather than *how* comments such as "sum of values divided by n".

Putting a comment at the top of a 3-10 line section telling what it does overall is often more useful than a comment on each line describing micro logic.

MintMT supports either the **REM** keyword or the ' character as a valid comment- ' is the preferred method for MintMT.

Comments that separate sections of code should use the '=' character as a highlight:

```

\=====
\ Program Macro definitions.
\=====
    
```

If the comments exceed more than one line, you should use the comment block style:

```

\-----
\ Variable nLedState used to indicate LED status
\ Allows either _ON or _OFF
\-----
Dim nLedState As Integer
    
```

Single line comments may be written as:

```

\ Variable nLedState used to indicate LED status
Dim nLedState As Integer
    
```

Comments may also appear on the same line as the code they describe. This is not the preferred method however.

```

Dim nLedState As Integer \ Variable used to indicate LED status
    
```



If code is temporarily commented out, use “If 0” and include a comment to explain why the code is being commented out. This allows large blocks of code to be commented out and re-instated where necessary with ease (just requires addition/removal of two lines of code rather than adding/removing the comment character to every line).

```

-----
` Code commented out during debugging
-----
If 0 Then
... ` Code to comment out `
End If
    
```

### Declarations and Data Types

There are only two data types provided by MintMT at present (string data types may be supported at a future date):

```

Integer          signed 32 bit data
Float            32 bit floating-point data
    
```

When declaring variables, constants and arrays the data type should always be included:

```

Dim nCount As Integer
Dim fLength As Float
    
```

A **modified form** of Hungarian notation should be used for all variable and constant declarations. The following pre-fixes are suggested:

Prefix	Type/Description	Example
_	Modifier - Constant Value	<code>Const _gnMaxspeed As Integer = 3000</code>
a	Modifier - array of	<code>Dim anAxes ( )</code>
b	Boolean (true/false)	<code>Dim bEnabled</code>
f	Float	<code>Dim fFollowingError As Float</code>
g	Modifier - Global value/variable	<code>gnMaxPosition, gfLength</code>
n	Integer	<code>Dim nProductCount</code>
s	String	<code>Dim sVersion</code>
ax	Controller Axis	<code>Const _axNipRolls = 3</code>
bt	Defines bitwise mask (singular bit)	<code>Dim btRXD = 0x40</code>
cm	Comms element	<code>Const _cmAction = 1</code>
er	Error codes	<code>Const _erOutOfRange = 1</code>
md	Mode of operation	<code>Const mdStopped = 0</code>
mk	Bitwise mask (multiple bits)	<code>Dim mkLowNibble = 0x15</code>
no	Number of	<code>Const _noAxes = 8</code>

Any variable whose initial value is important must be explicitly initialised.

```

Dim nCounter As Integer = 0
    
```

Where a global variable or local variable has a series of potential values, rather than using numerical values in code, define constants for these values. This makes the code easier to update at a later date should the numerical values used require modification. The constant definitions should be placed after the variable declaration and not within the program’s main global constant section.



```

-----
\ Machine Status variable
-----
Dim nMachineState As Integer

-----
\ Codes associated with nMachineState
-----
Const _nStopped = 0
Const _nJogging = 1
Const _nRunning = 2
    
```

## Functions

Functions should be defined in a meaningful order where possible (for example an initialisation function should be defined at the beginning of the function section of the program).

Functions should be given meaningful names in lower case where each word within the name begins with an upper case character (the function name itself must start with a lower case letter). It is recommended that function names should be pre-fixed with either *set*, *get* or *do*.

In MintMT, functions requiring parameters must include parameter declarations as part of the function definition. Functions with no parameters are declared using empty parentheses. Function declarations must include the return type of the function.

```

Function doFunc1(nAxis As Integer, fValue As Float) As Float
Function doFunc2() As Integer
    
```

Each function declaration should be preceded by a block comment prologue that gives a short description of what the function does and how to use it (if not clear). It should also describe the parameters and the function return value.

```

-----
\ FunctionName
\
\ A brief description of the function.
\
\ Argument list
\ Describe the function parameters.
\
\ Return value
\ Describe the functions return value.
-----
    
```

Local variables should be defined at the top of the function indented by 2 whitespaces. There must be no naming conflict between a local variable and a global variable.

## Subroutines

Subroutines should be defined in a meaningful order where possible (for example an initialisation subroutine should be defined at the beginning of the subroutines section of the program).

Subroutines are to be given meaningful names in lower case where each word within the name begins with an upper case character.

In MintMT subroutines requiring parameters must include parameter declarations as part of the subroutine definition. Subroutines with no parameters are declared using empty parentheses.

```

Sub CheckRange(nAxis As Integer, fValue As Float)
Sub CalculateRatio()
    
```



Each subroutine declaration should be preceded by a block comment prologue that gives a short description of what the subroutine does and how to use it (if not clear). It should also describe the parameters.

```

-----
\ SubroutineName
\ A brief description of the subroutine.
\
\ Argument list
\ Describe the subroutine parameters.
-----
    
```

Local variables should be defined at the top of the subroutine, indented by 2 whitespaces. There must be no naming conflict between a local variable and a global variable.

### Variable Naming Conventions

All variable names should be meaningful.

All variable names should be mixed case, capitalising each new word.

Variable names should follow the modified Hungarian notation as defined.

Underscores are not to be used within names with the exception of a leading underscore for constant data and as a separator between words in a macro or bits in a binary number.

### Macros

Macros can be used to replace MintMT commands, or sequences of commands. The MintMT keyword `Define` is used to declare a command macro.

Macros should be prefixed with, ideally, a two/three letter *lower case* mnemonic to identify their use. The following mnemonics are suggested:

Mnemonic	Function	Example
ip	Controller Input	<code>Define ipSTART_BUTTON = inx.5</code>
cm	Comms location	<code>Define cmOP_MODE = Comms(6)</code>
fn	Functional code segment	<code>Define fnENABLE_DRIVE = Cancel:DE=1</code>
nv	Non-volatile	<code>Define nvPRODUCT_COUNT = NVLONG(1)</code>
op	Controller Output	<code>Define opMACHINE_READY = outx.3</code>

Macros should be given meaningful names and, with the exception of the leading mnemonic characters, should be upper case. Words within the Macro name should be separated by underscore.

```
Define fnSIGNAL_HEALTHY = Outx.3 = 1
```

### Constants

It is good practise not to directly code numerical constants (often known as *magic numbers*). The `_` character should be used in combination with the `Const` keyword to declare constants with meaningful names. Symbolic constants make the code easier to read. Defining the value in one place also makes it easier to administer large programs since the constant value can be changed uniformly by changing only the declaration.

The `_` character should be followed by the appropriate prefix for the data type of the constant.



Constants should be defined consistently with their use; e.g. use `540.0` for a float instead of `540` with an implicit float cast.

```
Const _fMaxLength As Float = 4060.3
```

There are some cases where the numerical constants 0 and 1 may appear as themselves instead of as defined constants. For example if a For loop indexes through an array, then

```
For i = 0 To _nMaxArray  
...  
Next i
```

is reasonable.

## Formatting

Use vertical and horizontal whitespace generously.

Tabs should not be used for indenting or alignment as this can change from PC to PC.

Indenting should be implemented via 2 spaces.

```
For nIndex = 1 To 10  
  
    nArray(nIdx) = idx * nScaler  
  
    If nArray(nIdx) > _nMaxValue Then  
  
        nArray(nIdx) = _nMaxValue  
  
    End If  
  
Next nIndex
```

There should be a space either side of all operators (binary/logical/mathematical).

For example:

```
If inx.3 = 1 Then  
    bitmask = in & 0111_1010  
End If
```

rather than:

```
If inx.3=1 Then  
    bitmask=in&0111_1010  
End If
```

Expressions using mixed operators should be parenthesised where operator precedence is not immediately clear. For complex expressions, consider splitting the expression across multiple lines.

Particularly when using MT (as the program is compiled off line) the use of short-form keywords is not recommended. There is no speed advantage to be gained under MintMT by using the keyword abbreviations and the use of full keyword syntax results in a more readable program.



## General Practice

It is good MintMT programming practise to observe the following points:

- Document the firmware revision used as part of a known 'working' system. This can be found in SupportMe.
- Goto's should be used rarely, if ever. Always clearly comment the use of goto to make its intent known to other readers
- MintMT compiler warnings should not be ignored and should be resolved