



MintMT

Introduction to MintMT

Contents

1	General Information	1-1
2	Introduction	2-1
2.1	What is MintMT?	2-1
2.2	Why should I use MintMT?	2-2
3	Programming Environment	3-1
3.1	Introduction	3-1
3.1.1	Toolbox and work center front end	3-2
3.1.2	Project files	3-3
3.1.3	Program Navigator	3-3
3.1.4	Integrated help file	3-4
3.1.5	Capitalization and color coding	3-4
3.1.6	Editor	3-5
3.1.7	Command line	3-5
3.1.8	Data capture screen and fine tuning	3-6
3.1.9	Spy window	3-7
3.1.10	Digital I/O configuration	3-7
3.1.11	Product support information	3-8
4	MintMT Language	4-1
4.1	Introduction	4-1
4.2	New features	4-2
4.2.1	Modular programming and scope	4-2
4.2.2	Startup block	4-3
4.2.3	Subroutines	4-4
4.2.4	Functions	4-5
4.2.5	Multi-tasking	4-7
4.2.6	Events	4-11
4.2.7	Select Case statement	4-13
4.2.8	Constant declarations	4-14
4.2.9	Long lines	4-14
4.3	Enhanced features	4-15
4.3.1	If statement	4-15
4.3.2	Looping	4-15
4.3.3	Identifiers	4-16
4.3.4	Variable declarations and data types	4-16
4.3.5	Input statement	4-17
4.3.6	Scientific notation	4-17

4.3.7	Macros	4-18
4.3.8	Remarks	4-18
4.3.9	Square brackets	4-18
4.3.10	Auto statement	4-18
4.3.11	Array initialisation	4-19
4.3.12	Array assignment	4-19
4.3.13	Number of axes	4-19
4.3.14	New MintMT functions	4-19
4.4	Program structure	4-21
4	Mint v4 Compatibility	5-1
5.1	Introduction	5-1
5.1.1	Deleted keywords and features	5-1
5.1.2	Modified keywords and features	5-2
5.1.3	Features not yet included	5-2
5.1.4	Other changes	5-3

Copyright Baldor (c) 2001. All rights reserved.

This manual is copyrighted and all rights are reserved. This document or attached software may not, in whole or in part, be copied or reproduced in any form without the prior written consent of BALDOR.

BALDOR makes no representations or warranties with respect to the contents hereof and specifically disclaims any implied warranties of fitness for any particular purpose. The information in this document is subject to change without notice. BALDOR assumes no responsibility for any errors that may appear in this document.

Mint™ is a registered trademark of Baldor.

Windows 95, Windows 98, Windows ME, Windows NT and Windows 2000 are registered trademarks of the Microsoft Corporation.

UL and cUL are registered trademarks of Underwriters Laboratories.

Limited Warranty:

For a period of two (2) years from the date of original purchase, BALDOR will repair or replace without charge controls and accessories which our examination proves to be defective in material or workmanship. This warranty is valid if the unit has not been tampered with by unauthorized persons, misused, abused, or improperly installed and has been used in accordance with the instructions and/or ratings supplied. This warranty is in lieu of any other warranty or guarantee expressed or implied. BALDOR shall not be held responsible for any expense (including installation and removal), inconvenience, or consequential damage, including injury to any person or property caused by items of our manufacture or sale. (Some countries and U.S. states do not allow exclusion or limitation of incidental or consequential damages, so the above exclusion may not apply.) In any event, BALDOR's total liability, under all circumstances, shall not exceed the full purchase price of the control. Claims for purchase price refunds, repairs, or replacements must be referred to BALDOR with all pertinent data as to the defect, the date purchased, the task performed by the control, and the problem encountered. No liability is assumed for expendable items such as fuses. Goods may be returned only with written notification including a BALDOR Return Authorization Number and any return shipments must be prepaid.

Baldor UK Ltd
Mint Motion Centre
6 Bristol Distribution Park
Hawley Drive
Bristol, BS32 0BF
Telephone: +44 (0) 1454 850000
Fax: +44 (0) 1454 850001
Email: technical.support@baldor.co.uk
Web site: www.baldor.co.uk

Baldor Electric Company
Telephone: +1 501 646 4711
Fax: +1 501 648 5792
Email: sales@baldor.com
Web site: www.baldor.com

Baldor ASR GmbH
Telephone: +49 (0) 89 90508-0
Fax: +49 (0) 89 90508-492

Baldor ASR AG
Telephone: +41 (0) 52 647 4700
Fax: +41 (0) 52 659 2394

Australian Baldor Pty Ltd
Telephone: +61 2 9674 5455
Fax: +61 2 9674 2495


Baldor Electric (F.E.) Pte Ltd
Telephone: +65 744 2572
Fax: +65 747 1708


Baldor Italia S.R.L.
Telephone: +39 (0) 11 56 24 440
Fax: +39 (0) 11 56 25 660


Safety Notice


Only qualified personnel should attempt starting, programming or troubleshooting MintMT compatible equipment ("the equipment"). The equipment may be connected to machines that have rotating parts or parts that are controlled by the equipment. Improper use can cause serious or fatal injury.


Precautions


 **WARNING:** Do not touch any circuit board, power device or electrical connection before you first ensure that no high voltage is present at the equipment or other equipment to which it is connected. Electrical shock can cause serious or fatal injury. Only qualified personnel should attempt to start-up, program or troubleshoot the equipment.

 **WARNING:** Be sure that you are completely familiar with the safe operation and programming of the equipment. The equipment may be connected to other machines that have rotating parts or parts that are controlled by the equipment. Improper use can cause serious or fatal injury. Only qualified personnel should attempt to program, start-up or troubleshoot the equipment.

 **WARNING:** The stop input to the equipment should not be used as the single means of achieving a safety critical stop. Drive disable, motor disconnect, motor brake and other means should be used as appropriate. Only qualified personnel should attempt to program, start-up or troubleshoot the equipment.

 **WARNING:** Improper operation or programming may cause violent motion of the motor shaft and driven equipment. Be certain that unexpected motor shaft movement will not cause injury to personnel or damage to equipment. Peak torque of several times the rated motor torque can occur during control failure.

 **CAUTION:** The safe integration of the equipment into a machine system is the responsibility of the machine designer. Be sure to comply with the local safety requirements at the place where the machine is to be used. In Europe these are the Machinery Directive, the ElectroMagnetic Compatibility Directive and the Low Voltage Directive. In the United States this is the National Electrical code and local codes.

 **CAUTION:** Electrical components can be damaged by static electricity. Use ESD (electro-static discharge) procedures when handling the equipment.

2.1 What is MintMT?

MintMT is the latest release of the motion programming language Mint, provided free with all compatible products. MintMT introduces features found in many modern programming languages, which makes it even easier to use and support.

Many changes have been made to increase MintMT's effectiveness at providing a powerful, yet easy to use development environment. One of the major developments is in the area of multi-tasking (MT). The syntax of the language has been extensively enhanced in line with modern programming practices. These enhancements include true subroutines and functions (that can take a number of parameters) and a "Select Case" statement. The block "If" statement has also been extended to allow "Else If" clauses. To enable the best use of these new features, there is also a new version of Mint WorkBench. This builds on the features of the Mint v4 WorkBench and includes a syntax-highlighting editor, PC based compilation of MintMT programs and much more.

These are just a few of the new features to be found in MintMT. The remainder of this document provides a more detailed overview and how these features will help you. It should be read in conjunction with the new online help file, found by pressing F1 within Mint WorkBench. The online help provides full details on the new keywords introduced with MintMT.

2.2 Why should I use MintMT?

The modular structure of MintMT allows a much simpler approach to motion programming. Using proven techniques from similar visual languages, large projects can be divided into smaller sections. This means that if a change is required to one section of the program, only this part needs to be changed and the rest of the program will remain unaffected. Modification and debugging of programs becomes much simpler too - a problem can be quickly located in one small section of code, rather than having to search through an entire program.

When writing new programs, it becomes possible to copy common functions and subroutines already written for previous MintMT applications. Common tasks such as terminal I/O and input handling can be written as individual subroutines, with complex mathematics written as individual functions. These can then be reused in the new program - cutting development time significantly.

With programs now being compiled by WorkBench on the PC, the file can be checked for errors, compiled and downloaded to the controller extremely quickly, virtually eliminating the time taken to test each small change to a program - especially important during system installation and testing.

It is now much easier to learn MintMT, with the introduction of an on-screen help file. Can't remember how to use a keyword? Just press F1 to instantly display a complete reference to the MintMT language. Search for keywords and follow links to quickly find the information you need - and it's free on any machine running the new Mint WorkBench!

Lastly, for people with existing Mint v4 programs already in use, MintMT is fully keyword compatible. Simply run your old code until you have time to upgrade it to MintMT.

The benefits of these changes are obvious: faster development time, faster time to market, faster modifications and more satisfied customers!

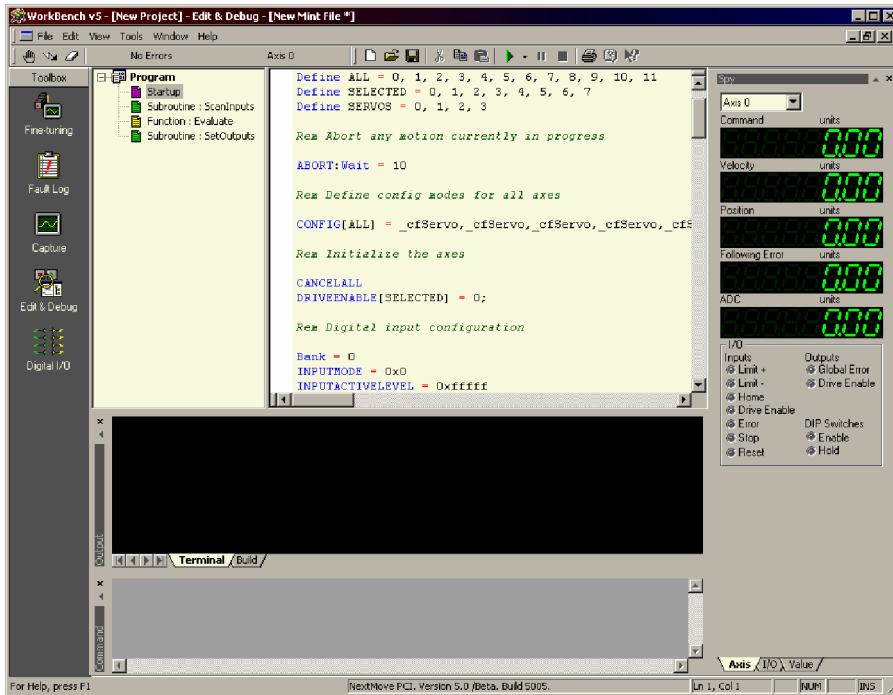
MintMT is currently supported on four products - NextMove PCI, NextMove BX, Flex+Drive^{II} and MintDrive^{II}.

New firmware must be downloaded to enable MintMT to be supported; this can be done very easily from Mint WorkBench. Other MintMT compatible products will be available soon.

3.1 Introduction

MintMT is composed of two main components, the Mint WorkBench v5, which runs on the host PC and the firmware, which runs on the controller.

The new Mint WorkBench v5 is designed around a work center concept, with a toolbox on the left of the screen taking you to common functions, such as the editor, data capture and I/O setup. This leaves the screen uncluttered, allowing you to concentrate on the current task. It provides many of the features found in other programming environments such as an editor with syntax highlighting, auto-indentation, auto-completion, syntax checking as you type, a program navigator and online help. However, WorkBench v5 also retains many of the functions of WorkBench v4, such as watch windows (now called the Spy window), a command line (the Command window), data capture and graphing. The Mint Configuration Tool has now been incorporated into WorkBench v5 to form the Commissioning Wizard.



The new Mint WorkBench v5

Under MintMT, Mint WorkBench v5 compiles the program. This is a significant difference between MintMT and Mint v4; under Mint v4 the program was downloaded to the controller, which then compiled it prior to execution. The host PC is much better suited to compilation than the controller, and it means firmware space on the controller is not required for a compiler.

The controller's firmware is composed of two main components, the Mint Virtual Machine (MVM) and the Mint Motion Library (MML).

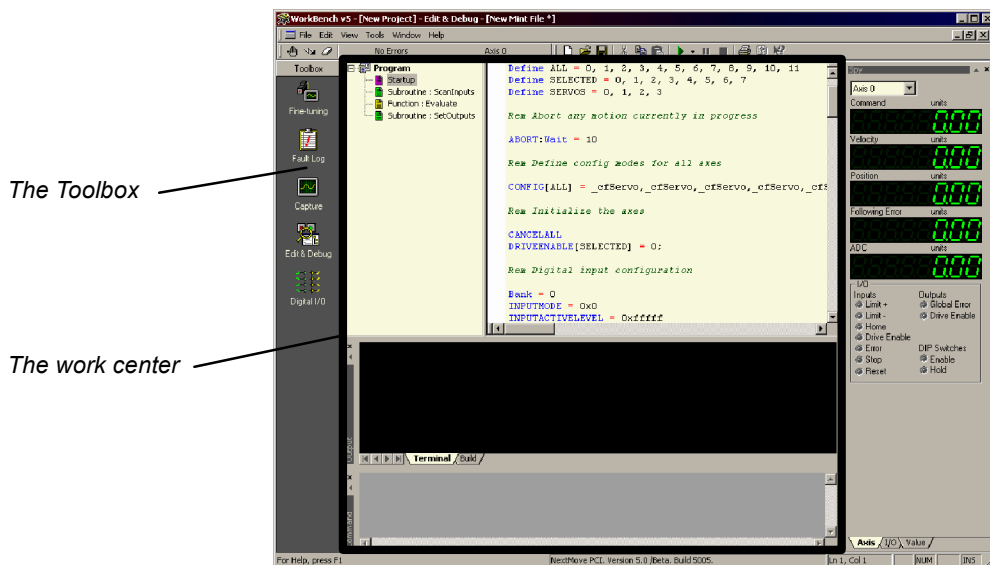
The compiler performs the complex task of decoding the structured, human readable source program into an executable file containing code targeted at the MVM. The MVM has a uniform instruction set across all Baldor controllers and drives. Its job is to process instructions by decoding them and then acting on them. The MVM also provides a run-time environment that enables the execution of multi-tasking programs. Although MintMT compatible products typically support only one central processing unit (CPU), the inclusion of a scheduler in the MVM enables rapid switching between tasks (time slicing) giving the illusion of concurrency.

Because one compiler is used for all controllers, any enhancements to the syntax or improvements to the code generation will be immediately available to all controllers supporting the MVM. One side effect of these changes is that it is no longer possible to program a MintMT based controller using a 'dumb terminal', though it is still possible to run a MintMT program.

The MML provides a common interface to most aspects of the hardware, but with an emphasis on providing motion control. The MML can be utilized with an embedded C program or with a MintMT program. This document focuses on MintMT and Mint WorkBench v5.

3.1.1 Toolbox and work center front end

Mint WorkBench v5 provides a cleaner and simpler interface to the controller. On the left hand side of the screen is a Toolbox. Selecting a tool in the Toolbox will change the view in the center of the screen. For example, you can change the view to Edit & Debug to edit a Mint program, or to Capture to view captured data.

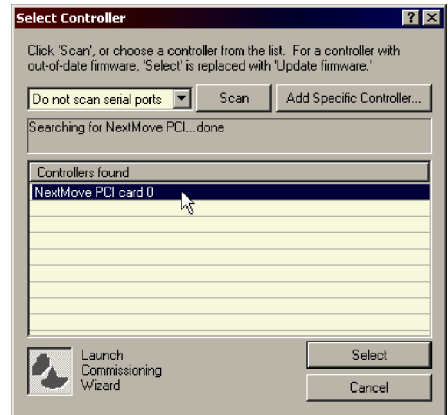
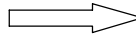


3.1.2 Project files

Mint WorkBench v5 uses project files. A project contains the program code you have written, together with information about the controller for which it was written. Every time WorkBench v5 starts, the opening dialog box is shown. This allows you to start a new project, load a previously saved project or run WorkBench in demo mode. When starting a new project, WorkBench will search for any installed controllers or drives connected to your computer. Projects use the **.wbx** filename extension.



Startup dialog

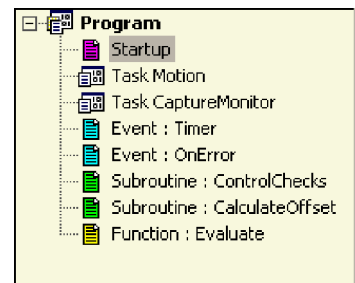


Selecting your controller

3.1.3 Program Navigator

MintMT now displays source code as a tree structure, which allows easy navigation of subroutines, functions, events and tasks. Just click on the object's name and it is immediately displayed in the editor. With syntax highlighting and automatic capitalization of keywords, the Mint MT programming interface is now even easier to use.

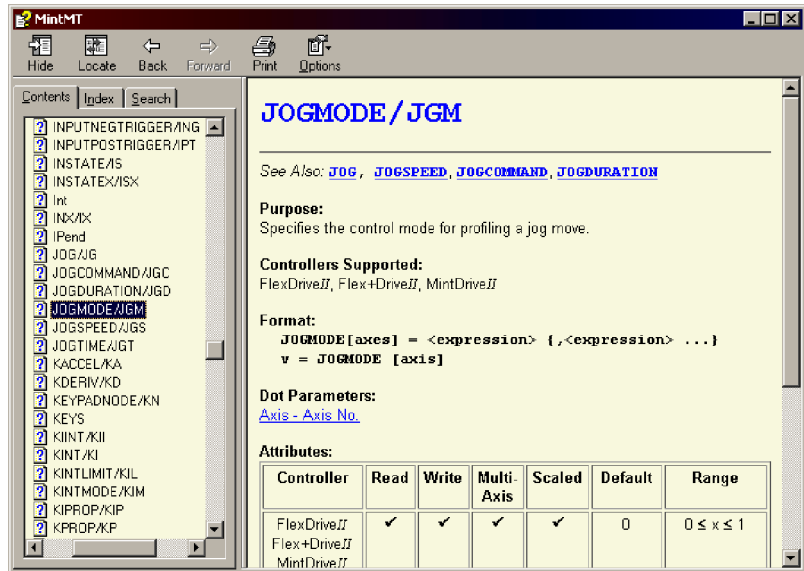
Tree structure of the program is shown in the new Program Navigator.



3.1.4 Integrated help file

A major addition to Mint WorkBench v5 is the inclusion of an on-line help file. This allows you to find all the information you need about MintMT and motion programming without needing to refer to printed programming manuals. To display the help file press the F1 key.

The new Mint WorkBench on-line Help file



Now, any PC running Mint WorkBench v5 will have all the necessary MintMT programming information in one place, ready to view.

The help file is in the familiar HTML Help format, which allows you to search the entire file very quickly for words and phrases. One of the most useful areas when programming will be the Keyword section. This is a complete listing of all the MintMT programming keywords, with full descriptions and examples in a similar format to previous programming manuals. The help file also provides information about using the new Mint WorkBench application.

3.1.5 Capitalization and color coding

As you will notice from the code samples in other sections, the capitalization of the standard core keywords has been changed. Previously, all keywords were shown in WorkBench in UPPER CASE. Core programming keywords are now generally converted to Initial case, although some keywords such as ErrAxis have special capitalization for clarity. Specialized motion keywords are still displayed using UPPERCASE for easy identification.

In addition, keywords, operators, variables and comments are highlighted in different colors. Combined with the changes to capitalization, MintMT code is now much easier to read, and is more similar to other programming environments.

3.1.6 Editor

The onboard editor used in Mint v4 products does not exist in MintMT products, so a number of commands are no longer needed, including `INS`, `DEL`, `EDIT`, `LIST`, `PROG`, `CON`, and `NEW`.

The PC running Mint WorkBench (the 'host') now provides the environment for editing programs, together with the ability to get memory usage information from the controller.

The configuration and program files are now combined to form a single program file.

3.1.7 Command line

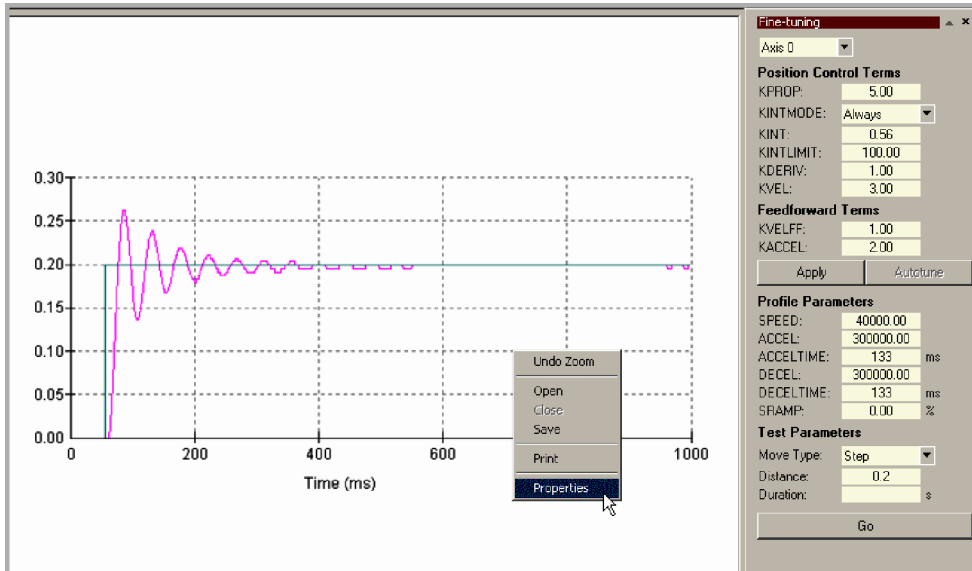
The command line remains in MintMT (using the Command window), although WorkBench now intelligently compiles each command before downloading it to the controller. Currently, commands can only be issued from the command line if the interpreter (in the controller) is not running a program, or while execution is paused at a breakpoint.

Because MintMT no longer has separate configuration and program files, the command line "`P>`" and "`C>`" prompts have been replaced with "`xx>`" where `xx` is the controller's node number. With no defined node number, the prompt will be just "`>`".

3.1.8 Data capture screen and fine tuning

When data has been captured from the controller, it can be displayed by the Capture screen. Additional new features allow the appearance of each series to be modified, a grid to be shown and captured data to be saved to a comma delimited file. This can be loaded back into WorkBench for later viewing, or can be used in spreadsheets and tables in other packages.

In Fine-tuning mode, the Capture screen is shown together with the Fine-tuning parameters window to allow the effect of changing MintMT gains to be monitored. The Fine-tuning window allows direct access to all the relevant MintMT gains needed to perform additional tuning of the system.

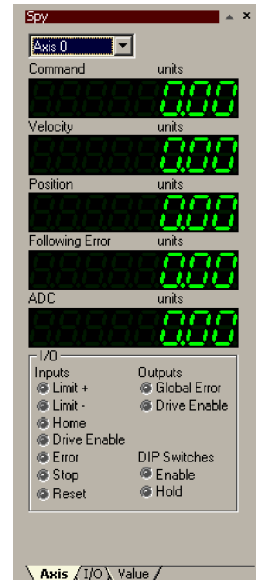


The Fine-tuning window and Capture screen in Fine-tuning mode. Right-clicking on the graph allows quick access to graph options.

3.1.9 Spy window

The new Spy window provides real-time display of parameters using virtual LED displays.

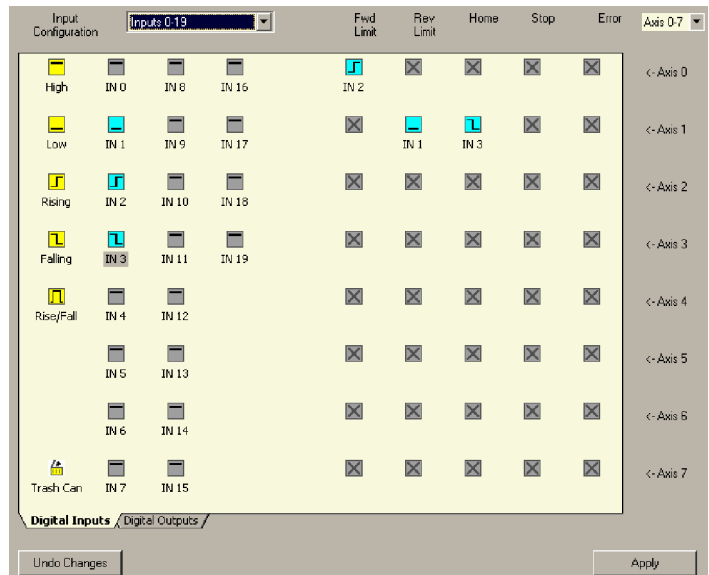
- The Axis tab shows multiple parameters for a chosen axis.
- The Value tab simultaneously shows the same chosen parameter on each axis.
- The Axis tab and I/O tab both show virtual LEDs to indicate the state for all inputs, outputs and limits on any chosen axis.



The Spy window

3.1.10 Digital I/O configuration

WorkBench v5 now includes an I/O configuration window similar to the one previously used in the Mint Configuration Tool. This allows you to configure all digital I/O using a simple drag-and-drop process on the screen.



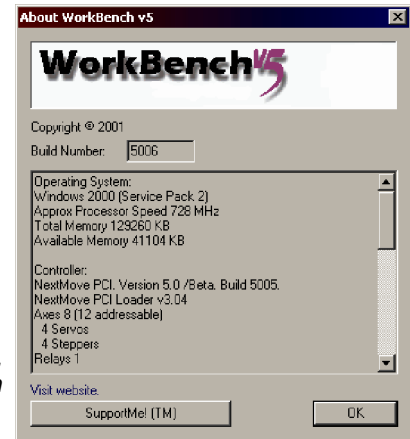
Digital I/O mode

3.1.11 Product support information

Using the Help, About box, WorkBench will now give you more information about your system including the Windows version, the Mint version and the number of axes supported.

If you run into any problems, just click the SupportMe (TM) button at the bottom of the window and send us the information - it can be invaluable to our support teams when solving your problem.

*The new About box,
showing extra information*



4.1 Introduction

The syntax of the MintMT language has been extended in a number of areas to provide a more feature rich environment. These enhancements enable the development of more robust code and make MintMT simpler to learn, as it is now much more like other popular programming languages. This means that the syntax of many commands has been changed. However, backward compatibility has been an important consideration so your old Mint v4 programs should still work with only a few exceptions.

When creating new programs or modifying old ones, it is highly recommended that you use the new-style commands. This will allow you to create simpler, easier to read programs.

The following sections describe in detail the new features of the language and how best to use them. There is also a description of all the enhancements made to existing features and details of compatibility issues with Mint v4.

4.2 New features

This section describes all the features that are completely new in MintMT such as subroutines and functions, multi-tasking, integer data and much more.

4.2.1 Modular programming and scope

MintMT introduces the concept of modular programming, which is a technique that allows problems to be divided into smaller, more manageable elements that can be assembled to form the complete solution. This strategy is sometimes referred to as “Divide and Conquer”, and is a simple but powerful technique.

MintMT supports this by providing various module types, such as subroutines and functions. These modules are designed to make modular programming feasible by allowing each module to have its own variables, constants, macros, etc. that are invisible outside the module. These items are effectively hidden inside the module and can only be accessed from inside that module. The concept of scope means that items declared within a module are said to have a scope ‘local’ to the module. When a module is exited, everything defined inside the module is said to go out of scope, and therefore cannot be accessed. Anything declared at the outer level is said to have ‘global’ scope and can be accessed from anywhere.

Scopes are automatically nested, so that the global scope may have one or more local scopes open in addition to its own, such as a task that has a local subroutine. The scoping rules work by looking for an object in the currently open scope, and if it is not there, it looks in the enclosing scope and then the scope that encloses that. This process is repeated until the object is found or there are no enclosing scopes remaining. This is best illustrated by an example, and while this example includes items not yet discussed, it depicts a parent task with a task nested within it (`myTask`) that has a subroutine (`mySub`) nested within it.

```
Dim a=1.1, b=2.2, c=3.3
Print a; b; c
Run myTask
Pause TaskStatus(myTask)=_tskTerminated
End

Task myTask
  Dim b=4.4, c=5.5
  Print a; b; c
  mySub

  Sub mySub()
    Dim c=6.6
    Print a; b; c
  End Sub
End Task
```

It can be seen that each module has variable declarations, and that these variables share names. This has been done on purpose but is not a problem, as each variable that shares a name has been declared in a different module and therefore has a different scope. The `Print` statement in the parent task (main program) has direct access to variables ‘a’, ‘b’ and ‘c’ and so will display “1.1 2.2 3.3”. The `Print` statement in ‘`myTask`’ has direct access to variables ‘b’ and ‘c’, but has to look in the enclosing scope to access variable ‘a’, and so will display “1.1 4.4 5.5”. Finally, the `Print` statement in

'mySub' has direct access to variable 'c', but has to look one scope up to access variable 'b' and two scopes up to access variable 'a', and so will display "1.1 4.4 6.6".

This may seem complicated at first, but it does allow modules to be written independently without having to worry about names conflicting with each other.

4.2.2 Startup block

The configuration file no longer exists and has been replaced by the Startup block. This can be placed anywhere in the program and is designed to contain program lines for drive configuration purposes. It has the special property that no events can occur while it is executing. An example of its syntax is shown below:

```
Startup
... `Configuration data
End Startup
```

When Run is executed within a program, the startup code will be ignored, in the same manner as Mint v4's configuration file. The following example shows the beginning of a typical startup block:

```
Startup
Define ALL = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11
Define SELECTED = 0, 1, 2
Define SERVOS = 0, 1, 2

`Define configuration modes for all axes
CONFIG[ALL] = _cfServo, _cfServo, _cfServo, _cfOff, _cfOff, _cfOff, _
_cfOff, _cfOff, _cfOff, _cfOff, _cfOff, _cfOff

`Initialize the axes
CANCELALL
DRIVEENABLE[ALL] = 0
DEFAULTALL

`Digital input configuration
Bank = 0
INPUTMODE = 0x0
INPUTACTIVELEVEL = 0xffff
INPUTPOSTTRIGGER = 0x0
INPUTNEGTRIGGER = 0x0

`Analog input configuration
...
... `further configuration data here
...
End Startup
```

Note that any items declared within the Startup block are local to it and will not be visible outside the Startup block. This is a feature of the block structuring of MintMT that enables modular programming techniques to be efficiently applied. If any items are required to be visible in the parent task and the startup block, then they must be declared in the parent task, which makes them global.

4.2.3 Subroutines

MintMT introduces subroutines and functions into the language, common with all modern programming languages.

```
Sub mySub([myInputVariable As type]) [As type]
...
...           'Code for the subroutine
...           'is entered here
...
End Sub
```

When you type the first line of the subroutine, `Sub mySub` for example, WorkBench v5 automatically sets up a separate area for entering the code, complete with the required `End Sub` line.

Previous versions of Mint used the `#label` definition for subroutines. MintMT now supports true subroutines and functions, which can accept parameters and use local data. This makes it easier to write well structured code and removes the need to use global variables for transferring data to the subroutine. Subroutines and functions allow for code reuse as all the functionality of the subroutine is held within the subroutine. The main features are the ability to pass parameters and local data. The new format replaces the use of `#`, `RETURN` and `GoSub`. For example:

MintMT	Mint v4
<pre>move 1000,100... Sub move(distance, moveSpeed) SPEED = moveSpeed MOVEA = distance GO End Sub</pre>	<pre>DIM Distance DIM MoveSpeed MoveSpeed = 1000 Distance = 100 GOSUB move ... #move SPEED = moveSpeed MOVEA = distance GO RETURN</pre>

The subroutine is called simply by using its name; `GoSub` should not be used. To exit the subroutine before it has ended, `Exit Sub` is used; `Return` should not be used. The new format for subroutines has a number of benefits:

- Parameters can be passed. There is no longer the need to use global variables.
- The subroutine will be shown in the Program Navigator.
- Data (variables) can be declared locally.

The following example shows the subroutine structure:

```

Sub Menu(override As Integer)
  Loop
  Cls
  Print "Select Option:"
  Print "1 .. Move"
  Print "2 .. Home"
  Print "3 .. Set Speed"
  If override Then Print "4 .. Override"
  Print "0 .. Exit"
  Pause Inkey
  Select Case LastKey
    Case '1'
      MoveAxis      `Call move subroutine
    Case '2'
      HomeAxis      `Call homing subroutine
    Case '3'
      SetAxisSpeed  `Call speed setting subroutine
    Case '4'
      If override Then doOverride  `Call override subroutine
    Case '0'
      Exit Sub
  End Select
End Loop
End Sub

```

Like subroutines in previous versions of Mint, the new subroutines can be called anywhere within a Mint program. Subroutines can also call other subroutines in a similar way to `GoSub`. When you type the first line of the subroutine, `Sub MySub` for example, `WorkBench v5` automatically sets up a separate area for entering the code, complete with the required `End Sub` line.

4.2.4 Functions

Functions are similar to subroutines because they perform a series of actions. Like subroutines, the function can receive one or more arguments (if required), but returns a single value. The addition of functions in `MintMT` means that commonly used calculations can be separated from other code, and can be called from any point in the `MintMT` program. Functions are defined using the structure:

```

Function myFunction([myInputVariable As Integer]) [As type]
  ...
  ...           `Code for the calculation
  ...           `is entered here
  ...
  myFunction = ...   `Assign return value to the function
End Function

```

When you type the first line of the function, for example `Function myFunction`, `WorkBench v5` automatically sets up a separate editing area for entering the code, complete with the required `End Function` line.

For example, consider a generic function that range checks an input variable. If the variable value is outside the range given by the min and max values, it will be set to the min or max value.

```
Function rangeCheck(x As Float, min As Float, max As Float) As Float
  If x < min Then
    rangeCheck = min
  Else If x > max Then
    rangeCheck = max
  Else
    rangeCheck = x
  End If
End Function
```

In the example above, you can see that the function's return value is specified by assigning it to the function's name. When the function exits, this return value is used by whatever required it. Note that it is the responsibility of the user to ensure that all paths through the function assign a return value, otherwise it will be undefined.

To call a function, the name of the function is used with any parameters enclosed in brackets:

```
Speed.0 = rangeCheck(mySpeed, 10, 1000)
```

When the function exits, its value is returned in place of the 'rangeCheck(mySpeed, 10, 1000)'. Alternatively, a call to a function can also be embedded in an expression, for example:

```
myResult = 15 * rangeCheck(mySpeed, 10, 1000)
```

An exception to enclosing parameters in brackets is when a function takes no parameters. In this instance, you have the choice of either calling the function using brackets with nothing between them or using no brackets at all. This rule is true for both user defined functions and built in functions. For example, Rnd is a special function built in to MintMT that returns a random value:

```
If Rnd<=1/3 Then
  ...
Else If Rnd()<=2/3 Then
  ...
Else
  ...
End If
```

4.2.5 Multi-tasking

Perhaps the most important new feature of MintMT is the ability to setup multiple tasks that can run at the same time, also known as parallel or concurrent processing. This means that a program does not have to be written to make it regularly branch to different sections of code to perform other duties.

For example, in a typical scenario where a program is required to control the motion of an axis, check the state of inputs and update an output display, separate tasks could be written to perform each duty. This makes the program much easier to read and debug, as each task is clearly separated.

Within MintMT, there is always a task called the parent task that contains all the statements at the outer level, i.e. statements not nested within a task, event, subroutine, function or start-up block. The parent task is equivalent to a Mint v4 program and therefore allows a Mint v4 program to be imported into MintMT. In addition to the parent task, you may declare as many additional tasks of your own as are required. These tasks are nested within the parent task and are called child tasks.

Tasks are written using the structure:

```
Task myTask
  ...           'Code for this
  ...           'task is
  ...           'entered here
End Task
```

When you type the first line of the task, for example `Task MyTask`, WorkBench v5 automatically sets up a separate editing area for entering the code, complete with the required `End Task` line.

The act of declaring a task does not mean that it will automatically start when the program is run. To make a task start, the `Run` command must be used together with the task's name, as shown in the example below:

```
Run myTask
```

It is also possible to start a number of tasks at once using one statement:

```
Run myTask, checkInputs, updateDisplay
```

A task will finish after it executes the last statement in the task. Therefore, tasks that must run indefinitely should use `Loop..End Loop`, while a task such as a keypad monitoring task should use a conditional loop to exit only when specified.

The parent task is unique because anything declared in it is global and can be accessed from anywhere. User defined tasks may contain declarations of variables, constants and macros, but these are local to their task and cannot be accessed from outside the task. For example:

```
Const value = 2000      'A global constant

Task taskOne
  Const value = 1000    'A constant local to taskOne
  Dim var1 = value      'A variable local to taskOne

  Print var1
```

End Task

```
Task taskTwo
  Dim var1 = value      'A variable local to taskTwo

  Print var1
End Task
```

taskOne will display a value of 1000, whereas taskTwo will display 2000. The variable var1 in taskOne is independent of the variable var1 in taskTwo, as is the constant value. This can be useful where tasks are used to control two parts of a machine supporting similar functionality. This concept of scope allows tasks to be written in isolation, knowing that they will not interfere with each other. Once the tasks have been written and debugged (possibly by different programmers) it is usually possible to join them together using some code in the parent task to create a program to control the entire machine.

Similarly, tasks may also contain declarations of subroutines and functions, and because these are declared within a task they are also local and only accessible from within their task. However, there is a limit to the module types that can be nested within a task. As all events are global in nature and completely unrelated to any specific task, they must be defined at the outer level. Similarly, machine start-up is a global process and so the start-up block must also be declared at the outer level. An example showing the use of global and local subroutines is shown below:

```
Axes[1]          'Set the default axis string for the parent task
Run myTask      'Start task
Loop
  globalSub     'Call to a global subroutine (localSub not visible)
  MOVEA=0
  GO
  Wait=500
End Loop
End             'End of main program (parent task)

Task myTask     'Begin task declaration
  Axes[2]      'Set the default axis string for myTask
  Loop
    localSub   'Could call globalSub or localSub here
    MOVER=0.5
    GO
    Wait=150
  End Loop

  Sub localSub 'This subroutine is local to task 'myTask'
    Pause IDLE 'IDLE will use the default axis string
  End Sub
End Task       'End task declaration

Sub globalSub  'This subroutine is global (declared outside a task)
  Pause IDLE   'IDLE will use the default axis string
End Sub
```

In this example, if `myTask` called `globalSub` instead of `localSub`, it would not alter execution at all. This is because each task has its own environment that is inherited by any subroutines or functions that are called by the task, even if they were declared globally. The important concept here is that the task performing the call *supplies its environment* to the subroutine or function. Even if a subroutine or function is declared globally, it will only use the parent task's environment when it is called from the parent task.

A task, whether it is a user defined task or the parent task, is essentially a complete program in its own right. Each has its own dedicated memory space, called the task environment (sometimes known as the process control block). The task environment contains data regarding the execution state of the task and items of direct interest to the user, such as default values for the axis string, I/O bank, CAN bus and terminal bitmap. These parameters are local to each task and are manipulated using the keywords `Axes`, `Bank`, `Bus` and `Terminal`. For example, the program below shows how two tasks can be made to operate on different axes:

```
Task taskOne
  Axes[0,1] 'Control axes 0 and 1
  ...
End Task

Task taskTwo
  Axes[2,3] 'Control axes 2 and 3
  ...
End Task
```

As was previously mentioned, when a program is run any user-defined tasks are not automatically started; it is the responsibility of the parent task to start them. Conversely, when the parent task terminates, all child tasks are automatically terminated. This behavior reflects the fact that the parent task is in complete control of scheduling tasks, and child tasks can only run while under the control of the parent task. It is therefore very important that the parent task waits for child tasks to terminate before it allows itself to terminate. For example, if there were five child tasks, four controlling axes and one monitoring a keypad, it would be reasonable to wait for the keypad task to terminate, which would only happen when the user selected the quit option. Alternatively, the parent task could be used as the keypad task, which may be simpler.

To determine the state of a task, the `TaskStatus` function is used, which returns either `_tskTerminated`, `_tskRunning`, `_tskSuspended` or `_tskWaiting`. When a task terminates, its status will change to `_tskTerminated`. The `TaskStatus` function can be used in conjunction with the `Pause` statement to wait for a task to terminate, as shown in the following example:

```
'Parent task
Run keypad, lateral, longitudinal, vertical, spindle
Pause TaskStatus(keypad)=_tskTerminated

'Task definitions here
```

A terminated task can be restarted using `Run`, but it is not possible to run more than one instance of a task at the same time. Trying to do this will cause the task to be run the first time and then re-executed from its beginning the second time. If two instances of a task are required to run in parallel, then two identical, but differently named, tasks must be declared. This may seem like it requires duplication of code, but by using a shared global subroutine to do the work, this can be avoided, as illustrated below.

```

Sub controlMotion()
  Loop
    'Statements to control motion
    ...
  End Loop
End Sub

Task motion1
  Axes[0, 1]
  controlMotion
End Task

Task motion2
  Axes[2, 3]
  controlMotion
End Task

```

In addition to the `Run` command, there are other commands and functions that are also used to control tasks. These are:

```

Run [task-list]
End [task-list]
TaskSuspend [task-list]
TaskResume [task-list]
TaskStatus(task-name)
TaskPriority task-name, priority
TaskQuantum task-name, quantum-size

```

If the `Run` command is issued from within a running program (with no parameters), the execution of the program starts from the beginning, but without executing the start-up block. When supplied with one or more task names as parameters, it starts execution of the specified task(s). If it is required to run the program from the beginning, and also run the start-up block, then the `Run Startup` command should be used.

The `End` command, when issued with no parameters, terminates the parent task and all child tasks. When supplied with one or more task names as parameters, only the specified task(s) are ended.

The `TaskSuspend` command is used to temporarily suspend execution of the specified tasks and the `TaskResume` command is used to resume execution of the specified tasks.

```

TaskSuspend myTask
...
TaskResume myTask      'Resume execution of myTask

```

The `TaskPriority` command is used to alter the priority of a task, which may be necessary when there are multiple tasks running that have different speed requirements. For example, a user interface task would typically be given a lower priority than a motion control task. Priority level is specified using an integer value, which defines the relative importance of each task. This is used by the scheduler to determine which task to switch to next. For example:

```

TaskPriority myTask1, 10  'Medium priority
TaskPriority myTask2, 20  'Higher priority
TaskPriority myTask3, 1   'Very low priority

```

If no priority is specified, the default priority of 10 is assumed. Because the priorities are relative, there is no difference between priorities of 10, 20 and 1 compared to 100, 200 and 10, so long as it is remembered that any tasks whose priorities are not adjusted will continue to use the default priority of 10.

The `TaskQuantum` command is used to alter the number of instructions that are executed for the specified task before rescheduling to the next task. The default quantum size for all tasks is 10, which should only require adjustment in rare circumstances.

4.2.6 Events

An event is called in response to the special event to which it is assigned. Events in Mint v4 used the `#` symbol with a predefined label name. For example, to create an event on digital input 0, the following construct was used:

```
#IN0
...
RETURN
```

MintMT events are defined using the `Event` statement. Using the example above, this becomes:

```
Event In0
...
End Event
```

When you type the first line of `Event`, WorkBench v5 automatically sets up a separate area for entering the code, complete with the required `End Event` line. The event will be shown in the Program Navigator. The new format for events makes them more similar in appearance to other types of module. The available events are listed in the following table:

Event	Priority	Description
ONERROR	1 (Highest)	Called in the event of an error condition
CAN1	2	Called in response to a specific CAN1 event
CAN2	3	Called in response to a specific CAN2 event
STOP	4	Called in the event of the stop input on any axis becoming active
FASTIN	5	Called in response to a fast interrupt
FASTIN _x	6	Called in response to a specific fast interrupt
TIMER	7	Called in response to a timer event expiring
IN _x	8	Called in the event of a change of state of a specific digital input
COMMS _x	9	Call in response to a specific comms location changing (1-5 only)
DPR	10	Called in the event of a change to a DPR location
MOVEBUFFERLOW	11 (Lowest)	Called in the event of free spaces exceeding MOVEBUFFERLOW

Within their own priority levels, the `FASTINx`, `INx` and `COMMSx` events give higher priority to events with lower values of `x`.

All events have a higher priority than tasks so that when an event is triggered it will be called immediately (after the currently executing instruction) and will run to completion. An active event can be interrupted by a higher priority event that will itself run to completion prior to resuming the interrupted event. In this context, running to completion means that no task switching will occur while an event is executing. A task interrupted by an event will automatically resume after all active events have been executed.

Events can be exited before the `End Event` line by using the command `Exit Event`. This replaces the `Return` keyword.

An enhancement in MintMT is that events use a copy of the parent task's environment. This means that on entry to an event, the axes, bus, bank and terminal will be configured to those of the parent task. They will not be related to any random task that may have been active at the time the event became active. This means that there are no problems caused by the processing of an event at an unexpected time, and because a copy of the parent task's environment is used, there will also be no side effects in the parent task due to the execution of an event. The following example shows how this works:

```
Loop
  ...
  Bank = 0  `The error handler may get called after setting the bank,
  I = INX.0 `in which case the bank could be wrong (but not in MintMT)
  ...
End Loop

Event onerror
  ...
  Bank = 1
  ...
End Event
```

4.2.7 Select Case statement

The addition of the `Select Case` statement provides an alternative to multiple `If` statements for testing an expression. The `Select Case` statement allows an expression to be tested against a number of criteria, with appropriate code being executed if a condition is true. `Select Case` uses the structure:

```
Select Case expression
  Case expression-list
    ... 'statements
  Case expression-list
    ... 'statements
  Case Else
    ... 'statements
End Select
```

The first line, `Select Case...`, defines the expression to be tested which is typically the name of a variable. This expression is then tested using each of the following `Case` statements, in the order in which they are listed. When an expression is found to be true, the statements following it are executed up to, but not including, the following `Case` statement. For example, assume the value of `myVariable` is currently 25 and the following code is encountered (running on NextMove BX):

```
Select Case myVariable
  Case 1, 2
    Print "Value is too small"
  Case 3 To 100
    Print "Number is in range"
    LED = -2
    LEDDISPLAY = 34864
  Case 25
    Print "Exact match"
  Case Else
    Print "Number out of range"
End Select
```

The first expression, `Case 1, 2`, tests if the value of `myVariable` is 1 or 2. It is not, so the program continues to the next `Case` statement. This statement tests if the value of `myVariable` is between 3 and 100 inclusive. This statement is true, so "Number is in range" is printed to the Terminal window and the NextMove BX front panel LED display is made to show the "=" symbol.

Because a true statement has been found, this pass through `Select Case` is now finished. Any following expression that also may have been true, for example `Case 25`, will not be executed. If none of the `Case` statements are true then the optional `Case Else` statement is executed. If used, this must always be the last `Case` statement to appear before `End Select`.

4.2.8 Constant declarations

These are similar to normal variable declarations but are preceded by the `Const` qualifier and have their value defined by an expression. For example:

```
Const name [As type] = expression[,...]
```

The expression must be composed of literal values or other constants - variables cannot be used. For example:

```
Const X = 20           `Defines a constant X
Const Y = 30           `Defines a constant Y
Const XtimesY = X*Y    `Defines a constant XtimesY
```

If the `As` type is omitted, the constant will be given the data type that is most appropriate for the expression.

4.2.9 Long lines

Long lines can now be extended to the next line by use of the underscore character (`_`). This is placed at the end of the line to inform MintMT that the line continues onto the next line.

```
MoveA = length(1) * 100, length(2) * 200, length(3) * 300 _
        length(4) * 400
```

4.3 Enhanced features

This section describes features that existed in Mint v4 but which have been changed and improved in MintMT.

4.3.1 If statement

In Mint v4, the block `If` statement was ended with the single keyword `EndIf`. This is still supported in MintMT for backward compatibility but the block `If` can now be ended using `End If`, as shown in the example below:

```
If condition Then
  [statements]
Else If condition Then
  [statements]
Else
  [statements]
End If
```

The `Else If` clause is another new addition to MintMT and helps to make MintMT more like other programming languages. The `Else If` and `Else` clauses are optional, but there may be as many `Else If` clauses as required. However, only one `Else` clause is allowed and it must be the last `Else` statement before the `End If` (see example above).

The `Else` clause can also be used on single line `If` statements, for example:

```
If condition Then [statement] Else [statement]
```

4.3.2 Looping

Certain changes have been incorporated to make looping statements more readable and flexible.

In line with the block `If` statement, `Loop` and `While` may now be terminated with `End Loop` and `End While` respectively. The `For` loop has been enhanced by allowing the name of the loop counter to be optionally specified in the `Next` delimiter, for example `Next i`.

In Mint v4, loops could be terminated using the `Exit` keyword. This has been extended in MintMT so that the type of loop to exit may be specified. For example, `Exit For` would terminate the closest surrounding `For` loop and `Exit Repeat` would terminate the closest surrounding `Repeat` loop. Using an unqualified `Exit` statement terminates the closest surrounding loop of any type. The example below shows how `Exit` is used:

```
For i=...
  Repeat
    Loop
      If ... Then Exit For 'This exits the For..Next loop
      If ... Then Exit 'This exits the Loop..End loop
    End Loop
  Until ...
Next i
```

In line with the C language, there is a `Continue` statement that may be used to continue the next iteration of a loop. As with the `Exit` statement, the `Continue` statement can be optionally qualified with the type of loop to continue. For example:

```
For i=...
  Repeat
    Loop
      If ... Then Continue For 'This continues the For Next loop
    End Loop
  Until ...
Next i
```

4.3.3 Identifiers

In Mint v4, identifiers could be a maximum of 10 characters long, which could lead to problems with identifiers that had different names being treated as the same because the first 10 characters of each were the same. Under MintMT, identifiers can be up to 255 characters long.

```
Dim identifier1=-101.675
Dim identifier2=291.926
Print identifier1; identifier2
```

In Mint v4, the above program would have displayed “291.926 291.926”, because it is unable to distinguish between the two names, while under MintMT it would correctly display “-101.675 291.926”.

4.3.4 Variable declarations and data types

Under Mint v4 all variables were by default floating-point. Under MintMT, variables may be declared as either floating-point or integer. If no type is specified, floating-point is assumed. Scalar variables are declared in the following way:

```
Dim name [As type][=value][,...]
```

For example:

```
Dim myvar1 'Floating-point variable (uninitialized)
Dim myVar1 As Integer = 10 'Integer variable (initialized)
Dim myvar1 As Float = 3.14 * 2 'Floating-point variable (initialized)
```

The final example above shows how expressions can be used in the initialization of a variable.

Arrays are declared in the following way:

```
Dim name(range-list) [As type][=value-list][,...]
```

where,

```
range-list = index-range[,...]
index-range = [low To] high
literal-list = value{[,value],;}
```

Multi-dimensional arrays are supported via the range-list, and there is no limit (memory permitting) on the number of dimensions allowed. By default, the first element of the array is indexed from 1 (the array base). If no type is specified, floating point is assumed.

Some examples of variable declarations are shown below:

```

Dim a(10) As Float      `Floating-point array indexed from 1 to 10
Dim b(0 To 3) As Float  `Floating-point array indexed from 0 to 3
Dim c(10, -5 To 5) As Float `Floating-point array indexed from 1 to 10
                          `and -5 to 5

```

The table below shows the valid data types and the permissible range of values:

Type	Size (bytes)	Range
Integer	4	-2147483648 to 2147483647
Float	4	-3.402823E+38 to 3.402823+E38

4.3.5 Input statement

The Input statement has been enhanced to provide context sensitive data entry. What this means is that it is now impossible to enter a badly formed number, for example one with multiple decimal points or signs. The allowable contexts depend on the type of data being used in the Input statement. For example, it is not possible to enter a floating-point value when the destination variable is an integer.

For integer data, the input statement allows the entry of binary and hexadecimal values in addition to decimal. This is accomplished by prefixing the value with a "0" for binary values and "0x" for hexadecimal values. Once the base has been specified, only characters valid for that base may be entered. If the Using clause is used, numbers may only be entered in decimal format.

Another enhancement is that the input statement may now accept an indexed array, which was not possible before. The example below illustrates this:

```

Dim data(10) As Float
Dim i As Integer
For i=1 to 10
    Input "Enter value: ", data(i)
Next i

```

4.3.6 Scientific notation

Decimal floating point numbers can now be specified using scientific notation, which has the format:

```
decimal-float = [+|-]{digits}[.{digits}][[e|E[+|-]digits]
```

The number after the 'E' is called the exponent, and is read as "times ten to the power of". for example, -3.85E+03 is read "minus 3.85 times ten to the power of three", and represents the number -3850.

Scientific notation (...E+03) is an addition to MintMT.

To enable the output of numbers in scientific notation, a new output modifier has been added called *Sci* to complement the existing *Dec*, *Bin* and *Hex* modifiers. When used with the *Using* clause, the two values specified for formatting represent the field width and number of decimal places.

4.3.7 Macros

Macros have been enhanced in MintMT so that the substitution may now include commands that are themselves macro substitutions, as shown in the following example:

```
Define axis=1
Define ready=IDLE.axis
Define notReady=Not ready
...
If notReady Then
  `Some code prior to pausing
  Pause ready
End If
...
End
```

4.3.8 Remarks

Remarks cause all further text up to the end of the line to be ignored. A remark can now be entered by using a single quote:

```
REM This is an old-style remark
`And this is a new-style remark
```

4.3.9 Square brackets

In Mint v4, square brackets could only be used to specify a number of axes for an axis related command to use. For example, the command:

```
POS[0,1,2,3] = 0;
```

will zero the position of axes 0, 1, 2 and 3. This functionality has been extended in MintMT to allow it to be used for commands that are not axis related. For example, the command:

```
OUTX[0,3,4,7] = 1;
```

will turn on outputs 0, 3, 4 and 7 of the current bank.

4.3.10 Auto statement

In Mint v4, the Auto statement had to be on the first line of a program for it to function. In MintMT, this restriction has been removed, and the Auto statement can be placed anywhere and will be recognized.

4.3.11 Array initialisation

Mint v4 allowed dimensioned arrays to be initialized in the `Dim` statement only. MintMT has been enhanced to allow this in an assignment statement that may be placed anywhere in the source code. For example, the following is now valid:

```
Dim a(10)
...
a=0; 'Initialize all elements to zero
...
```

4.3.12 Array assignment

MintMT now allows an array to be assigned to another array in the same way that other variables can be assigned. If the source and destination sizes differ, then only the number of elements required by the destination are copied. Assignment completes when the source array's last value is reached. Also, array assignment treats an array as a simple sequence of values and so does not limit assignment to arrays of the same number of dimensions. For example, the following is now valid:

```
Dim a(10), b(10), c(3,8)
...
a=b 'Copy the contents of array 'b' into array 'a'
b=c 'Arrays of different structure may also be assigned
...
```

4.3.13 Number of axes

MintMT has been extended to handle a maximum of 32 axes (previously 12 in Mint v4). Although this extra capacity cannot currently be used, it means that MintMT will be capable of interfacing to future motion controllers without further changes.

4.3.14 New MintMT functions

These new functions enhance those supported in Mint v4 and form part of the core MintMT language. For full details see the Mint WorkBench v5 help file, described in section 3.1.4.

`Atan2`

This is the same as the existing keyword `Atan`, but takes two parameters and will return the angle in the correct quadrant. Importantly, the result is a value expressed in degrees, not radians.

`Float`

Converts an expression to a floating-point value.

`Frac`

Returns the fractional part of an expression.

`Log10`

Returns the common (base 10) logarithm.

Rnd

Returns a floating-point pseudo-random number in the range $0 \leq x < 1$ (greater than or equal to zero but less than 1).

Round

Similar to the `Int` function, but returns the integer value of an expression by rounding to the nearest whole number.

Sgn

Returns an integer indicating the sign of the argument.

For negative numbers, -1 is returned; for positive numbers, +1 is returned; for zero, 0 is returned.

4.4 Program structure

The Program Navigator in WorkBench v5 can help you structure your code more efficiently. While it is possible to structure a program in an arbitrary manner, it can result in code that is difficult to understand and which requires more effort to maintain. As a guide, the recommended structure for a MintMT program is shown below.

```
\-----  
\Place macro declarations here  
\...  
  
\-----  
\Place constant declarations here  
\...  
  
\-----  
\Place global variable declarations here  
\...  
  
\-----  
\Start tasks and set priority if necessary  
Run taskOne, taskTwo  
  
\-----  
\Pause while the tasks are running  
Pause TaskStatus(taskOne)=_tskTerminated & _  
      TaskStatus(taskTwo)=_tskTerminated  
End  
  
\-----  
\Place configuration code here in the startup block  
Startup  
  Auto   'Auto start the program on power up  
  
  'Configuration code here  
  ...  
End Startup  
  
\-----  
\Place task declarations here  
Task taskOne  
  ...  
End Task  
  
Task taskTwo  
  ...  
End Task  
  
\-----  
\Place global subroutines and functions here  
\...  
  
\-----
```

`Place event declarations here

Event OnError

...

End Event

`-----`

`End of Program

5.1 Introduction

MintMT has introduced many major improvements to the Mint language, and whilst every effort has been made to ensure compatibility with Mint v4, there are some important changes that have occurred. Some features found in previous version of Mint have also been removed completely or superseded by new keywords. This section details all the changes that impact on the compatibility of MintMT with Mint v4.

5.1.1 Deleted keywords and features

A number of keywords have been permanently removed from MintMT, as they are either no longer necessary or have been replaced by other keywords or WorkBench v5 functionality. These include:

- The keywords `CON`, `DEL`, `ECHO`, `EDIT`, `INS`, `KEYPADNODE`, `LIST`, `NEW`, `PROG`, `VER` and `VIEW` have been removed. These have either been replaced by WorkBench v5 functionality (`DEL`, `EDIT`, `INS`, `LIST`, `NEW`, `VER` and `VIEW`), are no longer relevant (`CON` and `PROG`), or have been replaced by new functionality (`KEYPADNODE`).
- While there is a command line built into the WorkBench v5, it is no longer possible to issue commands from a dumb terminal, such as a VT100.
- There is now no need for the Squash function so it has been removed from MintMT. With the program code now being compiled in WorkBench v5 (not within the controller as with Mint v4 products), only a small program and a heavily compressed version of the source code are downloaded to the controller. If memory still becomes limited, WorkBench v5 provides the option to not download the compressed source code.

5.1.2 Modified keywords and features

During the development of MintMT, some changes were required to make MintMT more logical. These changes are described below.

- While the use of a bit pattern to manipulate the terminal has not changed, the meanings of the bits have changed. Instead of each bit representing a specific terminal device, such as an RS232 or CAN port, each bit now represents a generic terminal, which can be configured to be any of the ports supported by the controller.
- The keywords `IN`, `OUT`, `INSTATE` have been harmonized with the `REMOTEIN` and `REMOTEOUT` keyword so that they operate on a specified bank. There are now three new keywords called `INX`, `OUTX` and `INSTATEX` that operate on the specified channel in the same way as `REMOTEINX` and `REMOTEOUTX`.
- The `DO` keyword has changed slightly. It is now treated as a synonym of the `Then` keyword. This change means that when the `DO` keyword is used, it does not immediately imply a block-if statement unless followed by a statement separator (a colon or carriage return). Normally, this would not pose any problems, but when a block-if is required on a single line, such as is the case within a `Define`, then a compilation error will result when the `EndIf` keyword is encountered unless a colon is placed after the `DO` keyword.
- The power operator `^` has a new use in MintMT. In Mint v4, it was used as a delimiter, but this is not required in MintMT. It is now used to raise values to a power. For example:

```
x=y^2
```

raises variable “y” to the power of two.

- A scalar cannot be assigned to an array as in Mint v4, for example:

```
Dim MyArray(10)
MyArray=10
```

In Mint v4, the above code would copy the value 10 into a scalar variable that shares the same name as the array. However, under MintMT, the value 10 will be copied into the first element of the array.

- Use of some keywords on the command line such as `BANK` and `BUS` may result in erratic behavior. This is because they are processed on a line-by-line basis by `WorkBench v5` but are not necessarily sent to the controller. If this seems to be happening, try writing the commands on one line, separated by colons (:). Alternatively, write the commands into a short program that can be compiled and run in the usual way.

5.1.3 Features not yet included

- Array upload and download.
- `NODESCAN.0`
- Access to variables in a downloaded program using the command line.

5.1.4 Other changes

Variables now behave more like a conventional compiled language, such as C++. This means that a variable is not automatically initialized to 0. Therefore if a variable is defined using the `Dim` statement but is not assigned an initial value, its initial value will be unknown and erratic. For example:

```
Dim x
Print x
```

could result in an unusual value being returned, for example 259.01.

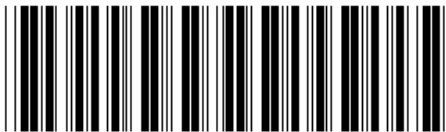
However if the following code is used, the expected result of 0 will be returned:

```
Dim x=0
Print x
```


BALDOR[®]
MOTORS AND DRIVES

Baldor Electric Company
P.O. Box 2400
Ft. Smith, AR 72902-2400
Tel: (479) 646-4711
Fax: (479) 648-5792

www.baldor.com



* 1 9 0 0 - 0 2 0 2 *

Printed in UK
Baldor UK Ltd