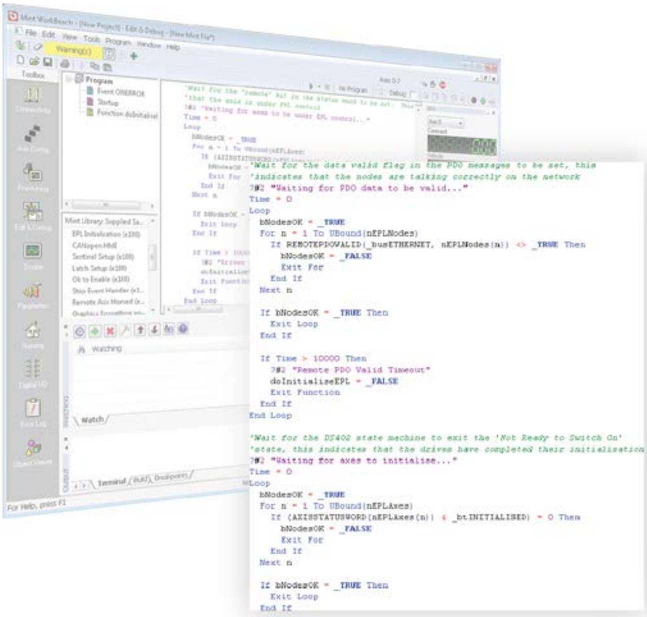
ABB motion control

Reference manual
Mint Basic Programming



```
'Wait for the data valid flag in the PDO messages to be set, this
'indicates that the nodes are talking correctly on the network
?#2 "Waiting for PDO data to be valid..."
Time = 0
Loop
  bNodesOK = _TRUE
  For n = 1 To UBound(nEPLNodes)
    If REMOTEPDOVALID(_busETHERNET, nEPLNodes(n)) <> _TRUE Then
      bNodesOK = _FALSE
      Exit For
    End If
  Next n

  If bNodesOK = _TRUE Then
    Exit Loop
  End If

  If Time > 10000 Then
    ?#2 "Remote PDO Valid Timeout"
    doInitialiseEPL = _FALSE
    Exit Function
  End If
End Loop

'Wait for the DS402 state machine to exit the 'Not Ready to Switch On'
'state, this indicates that the drives have completed their initialisation
?#2 "Waiting for axes to initialise..."
Time = 0
Loop
  bNodesOK = _TRUE
  For n = 1 To UBound(nEPLAxes)
    If (AXISSTATUSWORD(nEPLAxes(n)) & _btINITIALISED) = 0 Then
      bNodesOK = _FALSE
      Exit For
    End If
  Next n

  If bNodesOK = _TRUE Then
    Exit Loop
  End If
```

MN1955WEN

Power and productivity
for a better world™

You can find manuals and other product documents in PDF format on the Internet. See section *Document library on the Internet* on the inside of the back cover. For manuals not available in the Document library, contact your local ABB representative.

Contents

ABB Ltd
Motion Control
6 Bristol Distribution Park
Hawkley Drive
Bristol, BS32 0BF

Telephone:	+44 (0) 1454 850000
Fax:	+44 (0) 1454 859001
E-mail:	motionsupport.uk@baldor.com
Web site:	www.abbmotion.com

See rear cover for other international offices.

2.1 Introduction

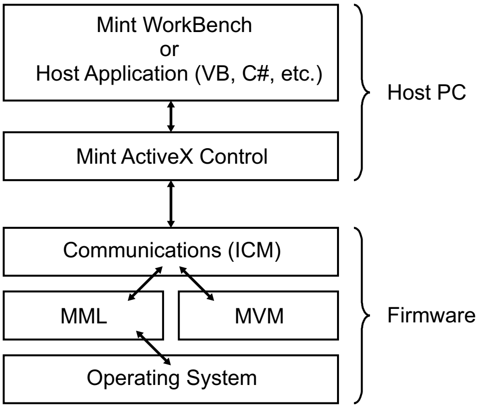
This document provides a thorough description of all the features of Mint Basic. Concepts are introduced in a staged manner and care has been taken to illustrate how to make the best use of the language and how to avoid potential pitfalls. A set of tutorials is included that guide the user from writing the simplest of applications all the way to a complex motion control application.

2.2 What is Mint?

Mint is the environment used to operate a range of ABB motion controllers and drives.*
Mint is composed of a number of elements:

- Mint WorkBench – This is the integrated development environment (IDE) used to configure, query and program the controllers and drives.
- Mint ActiveX controls – These allow applications to be written, typically in C++ or Visual Basic, which run on the host PC.
- Mint Motion Library – This provides a direct interface to the functionality of the hardware and resides in the firmware. The functionality available varies between products.
- Mint Basic – This is the language used to control the hardware, the functionality of which also resides in the firmware, hence allowing hardware to operate autonomously. Mint Basic executes programs using a virtual machine called the MVM, which allows a consistent range of features on all products that support programming.

The following diagram shows the various components that make up Mint.



Note that on some systems, the MVM can be replaced with an embedded C application.

** Motion controllers and drives originally produced under the Baldor brand name, for example: NextMove ES, NextMove ESB-2, NextMove PCI-2, NextMove e100, MicroFlex e100, MicroFlex e150, MotiFlex e100, et al.*

2.3 Mint Basic

This document focuses on Mint Basic, which is a dialect of Basic that has much in common with Visual Basic. It is rich in features, allowing the development of modular, block-structured programs. These include subroutines, functions, structures, conditional statements and looping statements. In addition to these features, Mint Basic also includes the capability to define tasks that execute in parallel with other tasks, allowing isolation of distinct operations from each other and simplifying program design.

While these features provide all that is required for purely programmatic purposes, they do not address the requirement to control the hardware directly. For this reason, Mint Basic allows access to an extensive library of specialized functions that interface to the functionality embedded into the controllers. This library is called the ‘Mint Motion Library’ (MML), and allows the manipulation of inputs, outputs and motion control, etc. The functions available within the MML are specific to a particular controller and its firmware revision, so not all functions are common to all platforms or all firmware revisions.

Mint Basic is a compiled language that targets a proprietary ‘virtual machine’, the Mint Virtual Machine (MVM), resident in the controller’s firmware. The Mint Basic compiler is integrated into the Mint ActiveX and is most easily used within Mint WorkBench, but can also be used by a host application. Since the MVM is a part of the controller’s firmware, it provides an efficient interface to the MML. This has a number of advantages when compared to an equivalent host application written in Visual Basic or C++ that uses the Mint ActiveX control, since there is no communication overhead, programs can start executing as soon as the controller is powered up, and no intervention is required from a costly host computer.

3.1 Introduction

There are a number of basic concepts that need to be understood before a program can be written. It is important to read and understand these concepts, since they form the fundamental building blocks on which the remainder of the language is built.

3.2 Identifiers

Identifiers are names used to uniquely identify items within a program such as variables, subroutines, functions and constants. These items will be discussed later, but the format of an identifier is the same whatever the item. Names should be chosen carefully to aid readability. Identifiers must start with either an alphabetic character or an underscore, and may be followed by any combination of alphanumeric characters and underscores. An identifier ends when a character is encountered that does not fit the above criteria, such as a space or a symbol.

The case of characters in an identifier is not significant, so ‘abc’ is considered to be the same name as ‘ABC’. All characters within an identifier are significant and there is no specific limit to the length of an identifier, though using very long names is not recommended.

Certain names have a special meaning to the language, and these are called reserved words or keywords. These include `If`, `Then`, `Float`, `Loop`, etc., a full list of which can be found in *Reserved words* on page 13-34.

The following examples are valid identifiers:

```
_a
__b
c_
size
belt_speed
x0
_100
_loop
a_strange_but_perfectly_valid_identifier
```

The following are invalid identifiers:

`sub`	(a reserved keyword)
`20x`	(starts with a digit)
`xor`	(a reserved keyword)
`piston length`	(contains a space, which creates two identifiers)
`n-points`	(contains a hyphen, which makes it an expression)

Identifiers must be unique within the current scope (discussed in *The concept of locality* on page 8-5), so it is not permitted for any two declarations in the same module to share the same name, such as a constant and a variable.

3.3 Literals

The term literal is used to describe an item that is literally represented in the program, such as a number, character or string. Since it represents something literally, it does not need a name or declaration to determine what it is. Literals are often called constants, though this term is preferred for use only with named constants (see *Constants* on page 5-1).

For example, to work out the circumference of a circle the numeric literals 2 and 3.1415927 can be used:

```
circumference = 2 * 3.1415927 * radius
```

String literals are often used in print statements, for example:

```
Print "Circumference = ", 2 * 3.1415927 * radius
```

3.3.1 Numbers

Mint Basic supports a variety of numeric formats to allow values to be expressed in the most natural way possible for the problem in hand. These vary from integer to floating-point and from decimal to other number bases and also formats suitable for expressing time durations and internet protocol (IP) addresses.

3.3.1.1 Decimal

A simple number begins with a digit and ends when a non-numeric character is encountered. The following are valid numbers:

```
1
-250
65535
```

Note that negative values appear to break this rule, but they are considered to be a simple expression composed of a unary minus operation on a positive value. There are a few other exceptions to this rule, which will be described in following sections.

The numbers described so far represent whole numbers (integers), but Mint Basic can also represent fractional values by using a decimal point. These are called floating-point values, examples of which are shown below:

```
2.718
-3600.5
0.0000001
10.0
```

When floating-point values get very large or very small, the number of digits required to represent them can become large. When this happens, scientific notation can be used to specify an exponent to scale the value. Scientific notation factors the value by 10^x, which is read as 'times ten to the power of x', where the value x is called the exponent. Exponents greater than zero increase the value and exponents less than zero reduce the value. For example, 0.0000001 can be expressed as 1×10^{-7}, and 1000000 as 1×10^{6}.

In Mint Basic, the exponent term is specified by immediately appending a value with an 'e' or 'E' and placing the exponent's value (in decimal) immediately after this, so 1×10^{-7} becomes

`1e-7` and 1 x 10^6 becomes `1e6`. Note that the exponent is always expressed in decimal. The following are valid numbers:

```
1.0365e-5
3.766224E-07
13.4485e6
9.80665e+0
```

The following are invalid numbers.

`2.76807 e1`	(numbers cannot contain spaces)
`3.769908e 2`	(numbers cannot contain spaces)
`1.0360795e+ 2`	(numbers cannot contain spaces)
`3.28084D0`	(invalid exponent character)

When a whole number exceeds the range of a signed integer (i.e. is outside the range -2147483648 to 2147483647), it is automatically assumed to represent a floating-point value, so `2147483648` represents `2.147483648e+9`.

3.3.1.2 Non-decimal

Usually numbers make most sense if they are decimal, but sometimes it is convenient to use other number bases. This is achieved by prefixing the number with a specification of its base, and the following table shows how this is done.

Decimal	Binary		Octal	Hexadecimal	
9	2#1001	01001	8#11	16#9	0x9
15	2#1111	01111	8#17	16#f	0xf
16	2#10000	010000	8#20	16#10	0x10
20	2#10100	010100	8#24	16#14	0x14
255	2#11111111	011111111	8#377	16#ff	0xff

The number formats that are prefixed with '0' for binary and '0x' for hexadecimal are for compatibility with older versions of Mint Basic. Those prefixed with 'base#' are derived from the IEC 61131-3 standard and is the preferred notation. Note that the base is always expressed in decimal.

Unlike decimal literals, based literals are assumed to represent an unsigned value, and therefore, so long as their size is within the range of an unsigned integer, its type will remain integer. However, since Mint Basic does not have an unsigned integer data-type, it is stored as a signed integer, so while the value `16#FFFF_FFFF` represents the unsigned value `4294967295`, it is stored as -1.

In the IEC 61131-3 standard only the bases 2, 8 and 16 may be used in numeric literals, but Mint Basic allows any base to be used[1], thus allowing decimal to be used when the intention is to specify an unsigned decimal value. The previous example using 16#FFFF_FFFF could be written using 10#4294967295, which would similarly be stored as -1. Note here the disparity between the statements `Print 10#4294967295` and `Print 4294967295`, which would display -1 and 4294967000.0000 respectively. This is because the base prefixed value is unsigned whereas the simple decimal value is signed. While the unsigned

1.Up to 36, as this is when the alphanumeric digits become exhausted.

value is within its valid range (0 to 4294967295), the signed value is outside its valid range (-2147483648 to 2147483647) thus making it be interpreted as a float. If the base prefixed value was one bigger, then it too would overflow its valid range and be interpreted as a float. Note that an integer can be displayed in unsigned notation using the C format string “%lu"; see *u: unsigned decimal* on page 13-16.

As a further extension to the IEC 61131-3 standard, numeric literals of arbitrary base may be used to define floating-point values. This broadly follows the rules used by Ada (and its derivative, VHDL), but without the requirement for a closing # character unless an exponent is used (because 'E' is a valid digit in the higher number bases). The exponent, like the base, is always expressed in decimal and represents the base to the power of the given value. So an exponent n will represent 10^n in a decimal number, 2^n in a binary number, etc.

The following examples illustrate its format:

`16#FFF`	integer value 4095 (IEC 61131-3 compliant)
`16#FFF#`	integer value 4095 (Ada compliant)
`16#F.FF`	floating-point value 15.99609 (Mint specific)
`16#F.FF#`	floating-point value 15.99609 (Ada compliant)
`16#F.FF#E+2`	floating-point value 4095.0 (Ada compliant)

Representing floating-point values in bases other than 10 is not normally required, but it does have its uses. For example, it can be used to advantage if it is required that an integer value be used as a floating-point value, but without the value-changing type reinterpretation, which can be achieved by simply appending a decimal point[2]. This means `16#FFFF_FFFF.0` represents the floating-point value `4.294967E+9`, but without the complication of having to work out its decimal digits. Another use is in the specification of a floating-point value that must be absolutely precise, for example when testing borderline cases in an algorithm. An example of this might be the representation of the largest and smallest floating-point values, which are `16#F.FFFFF#E+31` and `16#4.00000#E-32` respectively. Note how the mantissa uses only six hexadecimal digits, as there are only 24-bits available in the IEEE floating-point format (one hexadecimal digit occupies 4 bits). This example shows that using a based floating-point literal avoids having to guess how many digits of precision are required in decimal notation. The following table shows how some commonly used values are defined in different domains for both the IEEE 754 and the TI C31/33 DSP standards:

2. More correctly termed a radix point, which is the base independent term for the point that separates the integral and fractional components. In base 10 it would be the familiar decimal point, in base 2 the binary point, in base 16 the hexadecimal point, etc.

Parameter	Value			
	Float		Integer	
	Decimal	Hexadecimal	IEEE 754	TI C31 DSP
Maximum	$3.402823466 \times 10^{38}$	$F.FFFFF_{16} \times 16^{31}$	$7F7FFFFF_{16}$	$7F7FFFFF_{16}$
Minimum	$-3.402823466 \times 10^{38}$	$-F.FFFFF_{16} \times 16^{31}$	$FF7FFFFF_{16}$	$7F800000_{16}$
Smallest (normal)	$1.175494351 \times 10^{-38}$	$4.0_{16} \times 16^{-32}$	00800000_{16}	82000000_{16}
Smallest (denormal)[a]	$1.401298464 \times 10^{-45}$	$8.0_{16} \times 16^{-38}$	00000001_{16}	81000001_{16}
Epsilon[b]	$1.192092896e \times 10^{-7}$	$2.0_{16} \times 16^{-6}$	34000000_{16}	$E9000000_{16}$
Zero	0.0	$0.0_{16} \times 16^{0}$	00000000_{16}	80000000_{16}
One	1.0	$1.0_{16} \times 16^{0}$	$3F800000_{16}$	00000000_{16}

a. This parameter does not map to the TI C31 DSP, which has a smallest value of around 5.87747×10^{-39}, which has a hexadecimal representation of $2.0_{16}\times16^{-32}$ and a bit pattern of 81000000_{16}.
b. The term ‘epsilon’ has a few meanings, but here it represents the smallest value that can be added to 1.0 and register a difference.

Arbitrary base floating-point values are only supported in target formats 13 and above.

3.3.1.3 Time duration

Time durations follow the IEC61131-3 specification by using the `T#` or `TIME#` prefix. The prefix is followed by sections for the number of days, hours, minutes, seconds and milliseconds, suffixed by `d`, `h`, `m`, `s` and `ms` respectively. The only rules are that these sections must be in descending order of size and that no sections may follow one that has a fractional value. Note that all section values are decimal, and any fractional value of milliseconds will be truncated. The following are examples of valid time durations:

`T#3m25s`	(equivalent to 205000 ms)
`T#2.8s`	(equivalent to 2800 ms)
`T#2d12h`	(equivalent to 216000000 ms)
`T#2.5d`	(equivalent to 216000000 ms
`T#3m25000ms`	(equivalent to 205000 ms)
`T#3m25000.9ms`	(equivalent to 205000 ms)

The following are invalid time durations:

`T#2.0s500ms`	(milliseconds after fractional seconds)
`T#12h2d`	(hours before days)
`T#24d21h`	(value out of range)

Time literals always represent an integer value and so their type will not be automatically coerced to floating-point when they become too large to fit into an integer. The permissible range is -24d20h31m23s648ms to 24d20h31m23s647ms and an error will be generated if a value falls outside these limits. Time durations are only supported in target formats 13 and above.

3.3.1.4 Internet protocol

IP addresses are specified using the IP# prefix, which must be followed by four numbers in the range 0 to 255, each separated by a period, as shown below:

`IP#17.34.51.68` (equivalent to 287454020 and 16#11223344)

IP addresses represent a signed integer value, so `IP#255.255.255.255` has the decimal value `-1`. IP addresses are only supported in target formats 13 and above. Note that an integer can be displayed in IP notation using the C format string “%lq”; see *q: four octet (IP address) notation* on page 13-15. IP addresses are only supported in target formats 13 and above.

3.3.1.5 Use of underscores

Adjacent digits in the body of a number may be separated by a single underscore character, but the underscore cannot be used to initiate or terminate a sequence of digits or be used in a base prefix. This is purely to aid legibility and does not alter the value being represented. Below are examples of numeric literals that use the underscore:

```
2#1101_0001
16#8000_0000
3.141_592_654
31_225_001
1_225.500_183
```

Illegal uses of underscores are shown below.

`_100`	(underscore cannot start a number)
`100_`	(underscore not enclosed in digits)
`0_x7fff`	(underscore not enclosed in digits)
`1.275e_4`	(underscore not enclosed in digits)
`1.275_e10`	(underscore not enclosed within digits)
`12._6`	(underscore not enclosed within digits)
`12_.6`	(underscore not enclosed within digits)
`2#11__00`	(adjacent underscores)
`1_6#FF`	(underscore not allowed in base prefix)

3.3.2 Characters

Character data is defined by enclosing the required character within single quotes. For example, the character representing the question mark is expressed as ‘?’. The following code will print a question mark.

```
Print '?'
```

Character data is stored as an integer value with the range 0 to 255 encoded in ASCII (American Standard Code for Information Interchange), so it is possible to use characters whenever an integer is expected. Since the ASCII character set only specifies encodings in the range 0 to 127, any characters that lie outside this range will have a system dependent appearance. The following code assigns to a variable named ‘digit’ the value of a numeric character stored in a variable named ‘i’ (i.e. digit will be 0 for character ‘0’, 1 for character ‘1’ etc.).

```
digit = i – '0'
```

The following code will display 65, the ASCII code for the character ‘A’.

```
Print Int('a')
```

Note that the ASCII code for ‘A’ was displayed, not the value for ‘a’. This is because Mint Basic always assumes characters to be upper case, so ‘a’ has the same value as ‘A’. This behavior can be altered using either `Option CharCase`, or by adjusting the default setting in the Compiler Options dialog in Mint WorkBench. The following statement can be used to make character data retain its case.

```
Option CharCase 2
```

Characters not generally available on a keyboard, like control codes (those less than 32, and character 127) and non-ASCII characters (greater than 127), should be represented simply as integers. If required, these can be converted to character form, typically for output, using the `Chr` function.

Note that the single-quote character is also used to initiate a comment, and a consequence of this is that if by accident more than one character is specified between the single quotes, then this will be interpreted as a comment.

3.3.3 Strings

String data is defined by enclosing a sequence of characters with double quotes, for example:

```
"This is a string"
```

Note the use of double quotes rather than the single quotes that are used to represent character data. The following example shows a comment rather than a string or a character:

```
'This is not a string'
```

Unlike individual characters, the characters in a string are always stored in the case in which they were entered. An empty string is represented by two adjacent double quotes.

To allow the inclusion of double-quotes within a string, the backslash character is used immediately ahead of a double-quote, which is then inserted into the string rather than being used to terminate it. This syntax stops the inclusion of the backslash character in a string, but this is overcome by using two successive backslashes. Thus, `"\""` represents a string that contains a single double-quote character, and `"\\"` represents a string that contains a single back-slash character.

Strings may contain any character, including those that do not appear on the keyboard (such as the null character or a carriage return character). To define such a string, the notation backslash, hexadecimal character, hexadecimal character is used to specify the character (the case of the hexadecimal digits is not significant). For example, `"\00"` defines the null character and `"\ff"` defines character 255. The following example can be used to define a string constant that contains the two characters for carriage return and line feed:

```
Const _crlf = "\0D\0A"
```

Note that two hexadecimal characters must be used, so the leading zero must be present for values less than 16 (decimal). This notation is applicable to all characters, so an alternative (although impractical) way of printing “Hello world” would be to use the following code:

```
Print "\48\65\6c\6c\6f\20\57\6f\72\6c\64"
```

Note that the backslash notation is only relevant to strings, and cannot be used in character data.

3.4 Data types

Section 3.3 described the existence of different types of data, namely integer, floating-point, character and string data. As a character is an integer (with limited range), this leaves three fundamental types, which in Mint Basic are called `Integer`, `Float` and `String`.

3.4.1 Integer

Integer data is stored as a signed 32-bit integer using two's complement encoding, and so can store whole numbers in the range -2^{31} to $2^{31}-1$ (-2147483648 to 2147483647).

3.4.2 Float

Floating-point data is stored as a 32-bit word in the native format supported by the run-time environment of the controller. Usually, this is composed of 24 bits for the mantissa and 8 bits for the exponent, both including a sign bit. The implementation is usually close to the IEEE 754 standard, and so the range is of the order $10^{\pm 38}$ with a precision of around 7 digits. It is important to remember that floating-point values have a finite precision, and so are inherently inexact. It follows that computations involving floating-point values are also inexact because each operation is subject to truncation or rounding. This is discussed in more detail in *Floating-point limitations* on page 4-12.

3.4.3 String

String data is stored using a header followed by data words. The header is a 32-bit word containing the maximum size (in characters) in the least significant 16-bits, and the number of characters contained in the string in the most significant 16-bits. The data words are each 32-bits in size, with each word having four characters packed into it. The string data contains as many data words as necessary to completely contain the maximum size specified by the header. The maximum number of characters that a string can contain is specified using the notation shown below.

```
String*12   'A 12 character string
String*256  'A 256 character string
```

The maximum number of characters allowed is 65535.

3.4.4 Time

The `Time` type is stored as a 32-bit integer. It is very much like the `Integer` type, except that it constantly changes value to match the elapsed time in milliseconds. This effect is achieved by storing an offset relative to the continuously counting system millisecond timer. This offset is automatically generated on assigning a value to a variable of type `Time` and automatically mapped back on reading the contents of a variable of type `Time`.

To illustrate this mechanism, if a variable of type `Time` were set to zero, then the system millisecond counter would be read and this value stored in the variable. On reading the contents of the variable, its contents would be subtracted from the system millisecond counter, giving the elapsed time in milliseconds. This transformation is automatically performed so that variables of type `Time` can generally be used as if they were integers. Due to the data being stored as a 32-bit integer, variables of type `Time` will wrap approximately every 49.71 days. The `Time` data type is only available when `Option MintV5.5Keywords` (see page 7-4) is enabled, and only for target formats 12 or above.

3.4.5 Controller

The `Controller` type is a structure that contains two 32-bit integers called `nBus` and `nNode` that represent the bus and node. Variables of type `Controller` are used to make redirected MML calls, which are discussed in *Redirection* on page 11-4.

The `Controller` data type is only available when `Option  MintV5.5Keywords` (see page 7-4) is enabled, and only for target formats 12 or above.

3.4.6 Semaphore

This is a structure with two members, a 32-bit integer and an array of tasks indexed from 1 to size. Variables of type `Semaphore` cannot be initialized or manipulated in any way other than using them in a `Semaphore` block, so the names of the members are irrelevant. The number of resources that the semaphore controls is specified using an asterisk followed by an integer (the size), as shown below:

```
Semaphore * 2   'A 2 resource semaphore
```

Variables of type Semaphore are used to synchronize access to resources used by multiple tasks, which is an advanced subject that is discussed fully in *Semaphore block* on page 8-23.

The semaphore data type is only available when `Option MintV5.5Keywords` (see page 7-4) is enabled, and only for target formats 14 and above.

3.4.7 User defined

Mint Basic allows the user to define their own data types, which may be either a structure or a bitfield of a given name and members. In most circumstances, these types can be used in exactly the same way as the intrinsic types, the only exceptions being that structures can't be passed by value or returned by a function.

3.4.7.1 Structures

Structures are used to group data of arbitrary type together as a single entity, often called an aggregate, which may be composed of simple or structured types.

```
Structure TMoveInfo
  distance As Float
  id As Integer
End Structure
```

This is an advanced topic, and is fully discussed in *Structures* on page 5-6. The `Structure` construct is only available when `Option  MintV5.5Keywords` (see page 7-4) is enabled, and only for target formats 12 or above.

3.4.7.2 Bitfields

The bitfield type is used to give structure to a single 32-bit integer by using named members that each have a specific contiguous bit range. While this looks like an aggregate (see *Structures* above), it is simply a means of decoding the bit pattern of a single value, allowing data to be accessed directly without having to perform manual masking operations that can be complex and error prone.

```
Bitfield TBytes
  byte0 As 0 To 7
  byte1 As 8 To 15
  byte2 As 16 To 23
  byte3 As 24 To 31
End Bitfield
```

This is an advanced concept, and is fully discussed in *Bitfields* on page 5-8. The `Bitfield` construct is only available when `Option MintV5.5Keywords` (see page 7-4) is enabled, and only for target formats 14 or above.

3.4.8 Memory usage

The following table shows how much memory is used by each of the intrinsic data types.

Type	**Size (bytes)**
`Integer`	4
`Float`	4
`Time`	4
`Controller`	8
`String`	68
`String * 1`	8
`String * 65535`	65540
`Controller`	8
`Semaphore`	20
`Semaphore * 4`	32
`Bitfield`	4

Calculating the memory usage for the `String` type is more complex than other types. A string is composed of a 32-bit word that contains the maximum size in characters and the actual number of characters in it, and this is followed by enough 32-bit words as required to store the maximum possible number of characters. In general:

- Divide the size of the string by four, as there are four characters per 32-bit word.
- Extract the integer part, and if the result was fractional add one to it.
- Add one to account for the header word.
- Multiply by four to convert words to bytes.

For example, a string that is sized to contain only one character will be composed of the 32-bit header, and one 32-bit word that will only have one byte used. A string that is sized to contain 65535 characters will be composed of the 32-bit header and 16384 32-bit words, each of which containing four characters and the last word containing three characters.

The semaphore type is also a little complex as it is composed of an integer and an integer array of size equal to the semaphore's size, so the memory used by a semaphore of size *n* is 16 + 4 x *n* bytes.

3.5 Variables

Data is stored and retrieved using a named area of memory called a variable. Variables are declared using the `Dim` statement and may be any data-type supported by Mint Basic, and may be a single value or have many values if the variable is an array or a structure. The term ‘scalar’ is used to describe a single valued variable and the term ‘aggregate’ is used to describe a many valued variable.

```
Dim a As Float
Dim b As String * 256
Dim c As Semaphore
```

The declaration of variables is discussed in *Variables* on page 5-3.

3.6 Statements

Statements fall into one of the following categories:

- Declaration: Used to declare a named entity for later use, for example a variable or a subroutine.
- Action: Used to do something, like perform an assignment or make a subroutine call.
- Directive: Used to direct the behavior of the compiler, like setting the optimization level or specifying whether a program should auto-run on power-up.
- Comment: Used to annotate the operation of the program.

Programs are mostly composed of declaration and action statements, and a well written program will also contain a moderate number of comments to document its operation. Statements can be either simple, like an assignment, or structured, like a loop or a subroutine declaration. An example of simple statements is shown below, the first being an assignment, the second a subroutine call and the third a `Print` statement:

```
a = 2 * x + 1
calcTimings beltSpeed, productsPerSecond
Print "a = ", a
```

An example of a structured statement, a `While` loop, is shown below:

```
While Speed(0) > 10
  ...
End While
```

The statements used in declarations are discussed in *Declaration statements* on page 5-1, and those used to do something are discussed in *Action statements* on page 6-1.

3.6.1 Statement separation

Statements require some means of defining when one ends and the next begins, and this is achieved using either a new line or a colon. Below are examples of statement sequences:

```
a = (b + 1) / 2
b = Sqrt(b * (1 – c))
```

And the same two statements with the statement separator:

```
a = (b + 1) / 2 : b = Sqrt(b * (1 – c))
```

It is not common practice to use the statement separator other than in a single line `If` statement, since its use can make the program harder to read.

3.6.2 Line continuation

Sometimes, a single statement can be so long that it becomes desirable to make it span multiple lines. This can be achieved with the line continuation character, which is an underscore. Below is an example of its usage with a subroutine call that has three parameters, each being non-trivial expressions:

```
calculateForces d + x * (c + x * (b + x * a)), _
                Sqrt(1 + Sin(y) ^ 2), _
                (1 – z) / (1 + z)
```

Note that anything after the line continuation character is completely ignored up to the end of the line. This differs from Visual Basic, which will issue an error if anything exists after the line continuation character. A consequence of this is that the source reformatting facility will not include any characters that exist after the line continuation character.

3.7 Modules

The term ‘module’ is used to describe the declaration of a named block that contains statements, which when invoked, causes the statements it contains to be executed in sequence. The `Startup` and `Shutdown` modules are the only exceptions in that they do not require a name, since there may only be one of each in a program and they already have a unique name.

Mint Basic supports a number of different module types to simplify the design and implementation of a program. Their declaration and use is discussed in *Modular Programming* starting on page 8-1.

3.8 Program

A program is an ASCII file composed of a sequence of statements. The statements at the outer level are implicitly contained in a task called 'ParentTask'.

3.8.1 Layout

The general rule is that constants, defines and variable declarations should precede the executable statements, and the module declarations should follow. Specifically, it is recommended that you order your statements and modules in the following manner:

- Data declarations
  - Constant declarations
  - Define declarations
  - Type declarations
  - Variable declarations
- Executable statements
- Module declarations
  - Tasks
  - Functions
  - Subroutines
  - Events
  - Startup module
  - Shutdown module

Note that the `Startup` and `Shutdown` modules are placed out of the way as the very end of the program. This may seem counter-intuitive, but once written these modules are rarely changed and so they need not clutter the other code. Also, wherever they are placed amongst the other module declarations, the statements in the `Startup` module will always be executed first and the statements in the `Shutdown` module will always be executed last.

The above formatting guidelines can be applied to an existing program by clicking Program > Format Source Code in Mint WorkBench. See *Source code reformatting* on page 13-12.

3.8.2 Comments

Comments are an important part of any program, because however carefully variables, subroutines, etc. are named, there is often a need to explain a statement's purpose. Comments allow you to state clearly how a piece of code functions, aiding readability. It is good practice to use comments liberally in a program, but many statements often do not need further explanation. A comment is initiated using a single quote character, which causes all text up to the end of the line to be a comment. Below is an example of a comment.

```
'Calculate the pivot value, avoiding divisions by zero
```

The presence of comments does not have any impact on the performance of a program, though they increase the size of the source file, which, if stored on the controller will consume more memory.

3.8.3 White space

The term white space covers the use of spaces and blank lines, and its use can considerably improve the readability of a program. It is recommended that spaces be placed after commas and either side of arithmetic and relational operators at the very least, and anywhere else where readability can be improved by its use.

Indentation is a form of white space that makes a program easier to read by aligning the start of each line of code in a manner that reflects the nesting of the block-structured elements in the program. These block-structured statements have yet to be discussed, but to generalize a block-structured statement can be viewed as a statement container delimited by a pair of keywords of the form `Block`.. `End Block`. These delimiters should be aligned in the same column as the code that precedes it, and the statements contained should be indented further (two spaces is recommended).

The following example shows a poorly formatted example of a number guessing program:

```
Dim value As Integer
Dim guess As Integer
Dim attempts As Integer
Loop
value=1+Int(Rnd*100)
attempts=0
'Iterate until a correct guess is made
Repeat
'Read the guess and increment the number of attempts
Input "Enter guess: ",guess
attempts=attempts+1
'See if the guess was correct, prompting as required
If guess<value Then
Print "Higher"
ElseIf guess>value Then
Print "Lower"
End If
Until guess=value
'Notify the user of success and the number of attempts used
Print "Correct in ",attempts," attempts"
End Loop
```

Shown below is the same program, but with careful use of indentation and white space:

```
Dim value As Integer
Dim guess As Integer
Dim attempts As Integer

Loop
  value = 1 + Int(Rnd * 100)
  attempts = 0

  'Iterate until a correct guess is made
  Repeat
    'Read the guess and increment the number of attempts
    Input "Enter guess: ", guess
    attempts = attempts + 1

    'See if the guess was correct, prompting as required
    If guess < value Then
      Print "Higher"
    ElseIf guess > value Then
      Print "Lower"
    End If
  Until guess = value

  'Notify the user of success and the number of attempts used
  Print "Correct in ", attempts, " attempts"
End Loop
```

The careful use of white space is a powerful aid to readability so should always be used. All the code samples in this document are formatted carefully, and it is recommended that this technique is acquired early on and maintained. Since the subject of formatting is such a subjective matter, it is ultimately up to each individual to decide what style to adopt.

3.8.4 Compilation Errors

These occur when the program text contains something that does not conform to the language specification. This might be due to a badly formed number, a missing `Then` after the condition in an `If` statement, multiple declarations that share the same name, etc. These are all detected by the compiler and listed in the Build tab of the Mint WorkBench Output window. A program that contains errors will not execute until they are all resolved. The compiler also detects code that may be incorrect and issues a warning for each occurrence, and it is recommended that these are all resolved.

3.8.5 Execution

In general, execution starts at the first statement of the parent task and terminates when the last statement has been executed. However, there are some exceptions to this, listed below

- The presence of a `Startup` module will cause execution to start at its first statement, and when its last statement has been executed, execution will continue at the first statement of the parent task.
- The presence of a `Shutdown` module will cause execution to continue at its first statement when the program terminates for any reason. Usually this is due to the last statement of the parent task having been executed, but can also be the result of an unhandled error.
- Errors in the program will cause execution to terminate unless an error handler is present, in which case, execution will temporarily be directed to the error handler's statements. When the error handler's last statement has been executed, execution will continue from the point at which the error occurred. Note that errors in the `Startup` and `Shutdown` modules always cause immediate termination, whether an error handler is present or not,. Errors in the error handler are fatal unless `Option ErrorFatal` (see page 7-8) is set to 0 (zero).
- Executing the `End` statement will cause immediate termination.

When the parent task terminates, all child tasks also terminate. Execution is initiated either by the run-time system receiving a run command or automatically on boot-up if the program has been instructed to do so by using an `Auto` statement.

3.8.5.1 Run-time Errors

These occur when a program has compiled successfully and is executing. These fall into two categories, the synchronous error and asynchronous error.

- Synchronous errors are those that happen as a direct result of code in the program, such as a division by zero, a parameter being out of range, etc. If an error handler is present then it will be called, otherwise execution will terminate immediately. However, some synchronous errors are so serious that execution will terminate without executing the error handler, though the `Shutdown` module will still be executed.
- Asynchronous errors are those that either happen indirectly from code in the program or which are caused by an external interaction. For example, motion may be initiated under program control (or triggered externally via a host application or the command line), but

this motion might fail due to a following error. This will generate an error condition that will be handled by the error handler. However, an important difference from synchronous errors is that asynchronous errors do not cause the program to terminate if an error handler is not present.

Errors that occur in a `Critical` block (see page 8-21) that masks out the error handler are dealt with as soon as the error handler is no longer masked out. If multiple errors occur, then they will be handled in sequence on *e*100 products. On non-*e*100 products only the last error will be handled.

4.1 Introduction

An expression is used to calculate a result using a mixture of operators and operands. Expressions are used in many circumstances, such as in an assignment, calling a subroutine or a function, in conditional statements, etc. The simplest form of expression is simply a literal or an identifier, for example:

```
123
x
```

Fortunately, expressions can be more complicated than this, for example:

```
x + 1
x <= y
x(i) > 2 * (y + z)
x = y + z
```

Note that the last expression looks very much like an assignment (see *Assignment* on page 6-1 for details), because the same symbol is used to represent assignment and equality. However, the context of the '=' can be used to determine what it represents, but this can lead to somewhat confusing looking statements like:

```
equal = i = j
```

This evaluates the expression 'i = j', which will be either true (1) or false (0), and then assigns the result to variable 'equal'. This might be more clearly expressed as:

```
equal = (i = j)
```

However, the above expressions are much more efficiently evaluated than using an `If` statement, for example:

```
If i = j Then equal = _true Else equal = _false
```

A fuller description of assignment is covered in *Assignment* on page 6-1.

4.2 Operators

An operator is something that performs an operation on its operand data. Operators that take a single operand are called unary operators and operators that take two operands are called binary operators. An example of a unary operator is the unary minus, which uses the '-' character, so the expression '-j' will read the value from variable 'j' and negate the result. An example of a binary operator is subtraction, which also uses the '-' character, so in the expression 'i - j', 'i' and 'j' are the operands, and the result is the value of 'i' minus 'j'.

Binary operators, in general, do not evaluate their operands in a prescribed order. Consequently, the expression x() + y() may evaluate the function call x() first followed by y(), but it may do it the other way round. Consequently, no assumptions should be made with regard to operand evaluation order. The only exception to this is for the operators `AndAlso` and `OrElse`, which guarantee left to right evaluation.

4.2.1 Arithmetic operators

The arithmetic operators +, -, *, /, \, % and ^ are used to evaluate addition, subtraction, multiplication, division, integer division, modulus and exponentiation respectively. All these operators take numeric operands, which may be of mixed type, and each returns a numeric result of an appropriate type. The term 'appropriate type' means that if either operand is a float, then the result is of type float. If both operands are integer, then the result is of type integer. There are two exceptions to this:

- The division operator (/), which always returns a float.
- The integer division operator (\), which rounds any float operands to integer and returns an integer.

For example, the expression 3.142 + 123 will yield the floating-point result 126.142, and 128.7 \ 12.4 will be treated as 129 \ 12, yielding 10.

The modulus operator (`Mod` or `%`) performs a division, but returns the amount left over if the numerator is not an exact multiple of the denominator. It is normal to use this with integer operands, but it can also be used with floating-point operands. The sign of the result is the same as that of the numerator.

4.2.2 Relational operators

The relational operators =, <>, <, <=, >, >= are used to evaluate equality, inequality, less than, less than or equal to, greater than, and greater than or equal respectively. All these operators take operands that are either both numeric or both string, and all return an integer value, either 1 (one) if the condition is true, or 0 (zero) if it is false. When used with string operands, a lexical (character by character) comparison is performed, for example:

```
"abc" < "ABC"       'Evaluates to false (0)
"abc" < "abcdef"     'Evaluates to true (1)
```

While it is perfectly valid to use expressions like `inPosition = _true`, `inPosition <> _true` and `inPosition = _false`, these are considered poor style and are better expressed as `inPosition`, `Not(inPosition)` and `Not(inPosition)` respectively.

4.2.3 Logical operators

The term 'logical' relates to the two states 'true and 'false'. In Mint Basic anything non-zero is considered to be true and only exactly zero is considered false (note that this applies to floating-point values too, so even the smallest non-zero value is considered to be true).

There are four logical operators in Mint Basic, the unary operators `Not` and `Bool`, which are used to perform logical negation and logical affirmation respectively, and the binary operators `AndAlso` and `OrElse`, which are used to perform logical conjunction and inclusive disjunction respectively. All these operators take numeric operands and return an integer result (with the value `0` or `1`). The following truth table shows the results of applying the operators `Not` and `Bool`:

Operand	**Not**	**Bool**
-12846002	0	1
-1	0	1
-0.01	0	1
0	1	0
0.01	0	1
1	0	1
8366271	0	1

The `Not` operator has the symbolic equivalent `!`, and is used to reverse a logical state. The `Bool` operator has no symbolic equivalent (though using `!!` achieves the same result), and is used to convert a numeric value to a Boolean value, which can be useful when performing logical operations with the `And` operator. Since `Not` and `Bool` are operators, it is not necessary to use brackets around the operand, but it is considered good practice to do so.

The `AndAlso` and `OrElse` operators are similar to their bitwise counterparts `And` and `Or`, but differ in that they deal exclusively with the two logical states 'true and 'false'. They also differ in that they employ a technique called 'short-circuit evaluation', which means that operands are only evaluated if they need to be. For example when evaluating `i OrElse j`, if `i` is true (non-zero) then the whole expression is true and so there is no point in evaluating the rest (which could be something much more complex than the variable `j`). Likewise, when evaluating `i AndAlso j`, if `i` is false, then the whole expression is false, irrespective of the value of `j`. The following truth table shows the results of applying the operators `AndAlso` and `OrElse`.

Operand 1	**Operand 2**	**AndAlso**	**OrElse**
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

Remember that while only the operand values `0` (false) and `1` (true) are shown in the above table, any non-zero operand is considered to be true.

The AndAlso and OrElse operators guarantee left to right evaluation, so for example x() OrElse 1 will call function x, even though the whole expression is known to be true. Likewise, x() AndAlso 0 will call function x, even though the whole expression is known to be false.

To take most advantage of the short-circuiting of these operators, if it is known that an operand predominantly evaluates to false, then use it as the first operand of AndAlso. If it is known that an operand predominantly evaluates to true, then use it as the first operand to OrElse. These two tricks will increase the possibility that the second operand can be safely discounted, hence saving time.

Note that the Bool, AndAlso and OrElse operators are only available when Option MintV5.5Keywords (see page 7-4) is enabled. Furthermore, Bool is only for target formats 11 and above, and the AndAlso and OrElse operators are only available in target formats 13 and above.

4.2.4 Bitwise operators

The term bitwise relates to the bits that make up an integer value, which can be imagined as being a stream of true and false values stored in a single integer value (try displaying integer values using Print Bin to see the bit pattern). Unary operators work on a single set of bits and binary operators work on the corresponding bits from two sets.

The bitwise operators And, Or, Xor and ~ are used to evaluate bitwise conjunction, bitwise inclusive disjunction, bitwise exclusive disjunction and ones-complement respectively. The operators And and Or have symbolic equivalents & and | respectively. All these operators take numeric operands and return an integer result.

These operators are especially useful to turn on/off bits, as shown below.

```
'Turn on bits 3 and 5
bits = bits Or 2#101000

'Turn off bits 1 and 4
bits = bits And ~2#10010
```

Note the use of the one's complement operator (~) to invert the bits specified prior to using the bitwise conjunction operator (And) to turn off only the bits specified.

Special care should be taken when using the bitwise operators in a logical sense, as they will not necessarily give the result expected unless the operands are only 0 (zero) or 1 (one).
A typical mistake with these operators would be to assume that anything non-zero was true; in the following example, this means the program would not enter the If statement:

```
i = 2#10
j = 2#01
If i And j Then
   ...
End If
```

Of course, this is not an error, as the result of the conditional expression is zero since there are no corresponding bits that are both set. This type of problem can be avoided by using the logical operators AndAlso and OrElse, which give the added benefit of performing a short-circuit evaluation (see page 4-3). If logical operation is required but short-circuiting should be avoided, then i <> 0 And j <> 0 or Bool(i) And Bool(j) can be used, but remember that forcing each operand to be zero or one by using <> or `Bool` is only necessary when the operands may have values other than zero or one.

4.2.5 String concatenation operator

The + operator can be used with string operands to concatenate (join together) its string operands. For example, to append a semi-colon character onto the end of string ‘s’, the following code is used.

```
s = s + ";"
```

4.2.6 Immediate If operator

The ‘immediate if’ operator, `IIf`, gives the ability to make a choice within an expression. It takes three operands enclosed within brackets and evaluates to either the second or third operand depending on the condition expressed in the first operand. If the condition is true, then the second operand is returned, otherwise the third operand is returned. The second and third operands must be compatible, either numeric or exactly the same type. Only one operand is ever evaluated according to the condition. For example, to avoid a division by zero the following code can be used:

```
y = IIf(x <> 0, Sin(x) / x, 1)
```

The following two statements are equivalent:

```
s = Mid(IIf(i < 10, t, u), 4, 2)
If i < 10 Then s = Mid(t, 4, 2) Else s = Mid(u, 4, 2)
```

and the following two statements are also equivalent:

```
mySub(w, x, IIf(y < 0, 0, y), z)
If y < 0 Then mySub(w, x, 0, z) Else mySub(w, x, y, z)
```

Since the `IIf` operator is used like a function, its precedence is largely irrelevant. However, while it looks like a function, it is not executed like one, as that would entail the evaluation of all its operands before being called, which would severely negate its usefulness.

Note that `IIf` is only available when `Option MintV5.5Keywords` (see page 7-4) is enabled, and only for target formats 13 and above.

4.2.7 Is operator

The `Is` operator simply accesses the `Select` statement’s expression to allow it to be used in a `Case` statement. It is similar to `Case Is` in Visual Basic, but is slightly more flexible in that it may be used multiple times in one `Case` statement and does not have to immediately follow the `Case` keyword. For example:

```
Select Case i
  Case 12 To 16: DoStuff()
  Case Is < 0: RangeError()
End Select
```

would have to be coded:

```
Select Case i
  Case 12 To 16: DoStuff()
  Case _minInt To -1: RangeError()
End Select
```

Alternatively, an `If` statement could be used:

```
If i < 0 Then
  RangeError()
Else
  Select Case i
    Case 12 To 16: DoStuff()
  End Select
End If
```

Note that it is better to specify a range using a construct like `Case 12 To 16` rather than `Case Is >= 12 AndAlso Is <= 16`, as the compiler can generate significantly better code. Note that this operator is only available when the MintV5.5Keywords option is enabled, and only for target formats 11 or above.

4.2.8 Miscellaneous operators

The following operators require further knowledge of Mint Basic:

- The scope override operator (see *Overriding scope* on page 8-20).
- The member access operator (see *Structures* on page 5-6).
- The array-subscripting operator (see *Arrays* on page 5-4).
- The function-calling operator (see *Functions* on page 8-7).
- The compound parameter operator (see *Advanced parameter passing* on page 11-2).
- The redirection operator (see *Redirection* on page 11-4).

These operators will be explained when necessary.

4.3 Order of evaluation

An expression is evaluated by applying the operators to the operands in a prescribed sequence, in general taking operators from left to right. The sequence could be specified as being a simple left to right evaluation, but it is conventional for arithmetic operators to be evaluated in an order that is a function of the operators present. This sequence is defined by the precedence of each operator, which is described in detail below. However, it is possible to override this evaluation order by using brackets, and this is also discussed below.

4.3.1 Operator precedence

An expression is evaluated in a particular order, dependent on the precedence of the operators used in the expression. A common method uses the phrase BODMAS as a reminder, which stands for Brackets, Order (exponentiation), Division, Multiplication, Addition and Subtraction. This method evaluates terms in brackets first, followed by order (exponentiation or raising to a power), followed by division, followed by multiplication, followed by addition and subtraction. Although BODMAS specifies operator precedence, it is quite limited since it only applies to arithmetic operators. Note that brackets, as described by the 'B' in BODMAS, are not considered to be an operator. This is because they are only used to order the evaluation and so do not actually evaluate anything.

The following table shows the operator precedence for all Mint Basic operators, some of which have yet to be explained. This table groups operators of the same precedence, with the operators in one group having a higher precedence than operators in lower groups:

Operator	Description	Usage
`::`	Scope override	*module_name*::*local_name*
`.`	Member access	*structure_name*::*member_name*
`()`	Array subscripting	*array_name*(*exprs*)
`()`	Parameter passing	*module_name*(*exprs*)
`[]`	Compound parameter	*call_name*([*exprs*], *exprs*)
`->`	Redirection	*redirect_name->call*
`Not`, `!`	Logical not	Not(*expr*), !*expr*
`Bool`	Logical affirmation	Bool(*expr*)
`~`	Bitwise complement	~*expr*
`-`	Unary minus	-*expr*
`+`	Unary plus	+*expr*
`^`	Exponentiation	*expr* ^ *expr*
`*`	Multiply	*expr* * *expr*
`/`	Divide	*expr* / *expr*
`\`	Integer divide	*expr* \ *expr*
`Mod`, `%`	Modulus	*expr* % *expr*, *expr* Mod *expr*
`+`	Plus	*expr* + *expr*
`+`	String concatenation	*expr* + *expr*
`-`	Minus	*expr* - *expr*
`<`	Less than	*expr* < *expr*
`<=`	Less than or equal	*expr* <= *expr*
`>`	Greater than	*expr* > *expr*
`>=`	Greater than or equal	*expr* >= *expr*

Operator	Description	Usage

`=`
`<>`
	Equal
	Not equal
		expr = *expr*
		expr <> *expr*

And, &	Bitwise AND	*expr* And *expr*, *expr* & *expr*
`Or`, | `Xor`	Bitwise inclusive OR Bitwise exclusive OR	*expr* Or *expr*, *expr* | *expr* *expr* Xor *expr*
`AndAlso`	Logical conjunction	*expr* AndAlso *expr*
`OrElse`	Logical inclusive disjunction	*expr* OrElse *expr*
`IIf`	Immediate if	IIf(*expr*, *expr*, *expr*)
Is	`Select` expression	Is

Note that the term *expr* is used to represent an expression and *exprs* is used to represent a comma-separated list of expressions. For calls, *exprs* may validly represent no expressions at all, and in the case of a compound parameter, the comma between it and any following parameters present is only required if there are other parameters.

When an expression is composed of operators of the same precedence, binary operators are evaluated from left to right, and unary operators from right to left. For example `2*i\j` is evaluated as `(2*i)\j`, and `2^3^4` is evaluated as `(2^3)^4`, giving the answer 4096. Similarly, `-~-i` is evaluated as `-(~(-i))`, which is intuitive as a unary operator can only be applied to an operand that has been evaluated.

4.3.2 Use of brackets

Brackets are used to force the order of evaluation to whatever is required. For example, with the operator precedence rules of Mint Basic, the expression:

```
2 * i + 1
```

will be evaluated as

```
(2 * i) + 1
```

If this is what is required, then there is no need to use brackets, as most people are familiar with the correct evaluation order for arithmetic operators. However, brackets would have to be used if it was required for the expression to be evaluated as:

```
2 * (i + 1)
```

Mint Basic, unlike some languages, always honors brackets, even when used with operators of the same precedence. So, when evaluating `8*(i\4)`, the term `i\4` will be determined before multiplying it by 8. Usually this makes no difference, which is why some languages do not dictate that brackets should force the evaluation order for operators of the same precedence. However, in cases where the result would overflow (such as would be the case if `i` were greater than one-eighth of `_maxInt`), then it can be significant.

Once an expression becomes complicated, perhaps using many different operator types (arithmetic, relational, bitwise etc.), then it makes increasing sense to use brackets to make it clear how it will be evaluated, even if they are not required. Below are three examples of identical expressions, the first heavily bracketed, the second lightly bracketed and the last not bracketed at all:

```
(a < b) OrElse (c AndAlso (d = e))
a < b OrElse (c AndAlso d = e)
a < b OrElse c AndAlso d = e
```

The number of brackets used depends on personal preference. Too many brackets can clutter an expression, making it difficult to read, and too few can introduce uncertainty about how the expression will be evaluated without a detailed understanding of the precedence rules.

4.4 Functions

A function is a piece of code that returns a result, and as such its only use is in expressions. There are two types of function, intrinsic and user-defined. Intrinsic functions are those that are an integral part of the language, like `Sqrt` and `Log`, while user-defined functions are those that are declared by the user in their program. Both types of function are called in the same way, by using the function's name and appending any parameters that it requires enclosed within brackets. Below are examples of expressions that use function calls:

```
Rnd
Rnd()
Sqrt(x)
1 + Sqrt(x ^ 2 + y ^ 2)
(Exp(x) – Exp(-x)) / 2
calcConveyerSpeed(itemsPerSecond, itemSeparation)
```

Note that the brackets used to enclose the parameters are optional for functions that take no parameters, as is shown above with the `Rnd` function.

The declaration of user-defined functions is described in *Functions* on page 8-7, and tables of intrinsic functions are given in section 9, *Conditional Compilation*.

4.5 Type casting

Type casting is when data of one type is converted into another type. Often the user does this explicitly, so this is called an explicit cast. Below are some examples of explicit casts used in assignment statements (see *Assignment* on page 6-1):

```
i = Int(x)
x = Float(i)
```

Another type of cast is one that is automatically inserted by the compiler to ensure that data is compatible with its usage, and this is called an implicit cast. Below are examples of implicit casts:

```
i = x
x = i
```

The reason that this may be important is due to the internal representations of the different data types being such that one data type cannot necessarily be represented exactly using another type, which can cause a loss of precision. Examples of precision loss are shown below.

```
Dim x As Float, i As Integer
i = 2147483647
x = i
Print Int(x) - i
```

The above program will print the value -127, indicating a loss of precision. This occurs because it is not possible to store an integer that uses more than 24 of its 32 bits in a floating-point variable, which only has 24 bits of precision. The compiler detects these instances and issues a warning, allowing them to be examined to ensure their correctness. Assuming they are correct, the warning can be avoided by using an explicit cast.

More information on the type casting function available can be found in *Type conversion* on page 10-10.

4.6 Floating-point limitations

Special care has to be taken when using floating-point data. While it appears to have a high precision combined with a massive range (allowing it to express both large and small values), it cannot be relied on to store data exactly.

4.6.1 General Properties

Floating-point values are encoded in 32-bit binary, using 24 bits for the value and 8 bits for the scale factor, which gives an equivalent decimal precision of 7.22 digits. This encoding is broadly similar to that used by integer data, so the resultant value is the sum of the bit values that are set, where each bit value is a power of 2. The finite number of bits available coupled with the differing number base results in floating-point operations being inherently inexact. In the following table, the decimal value 10 is encoded in binary as 1010_2, along with some other examples:

	Bit				
	4	**3**	**2**	**1**	**0**
	2^{Bit}				
Decimal	**16**	**8**	**4**	**2**	**1**
0	0	0	0	0	0
1	0	0	0	0	1
2	0	0	0	1	0
3	0	0	0	1	1
10	0	1	0	1	0

As can be seen, whole numbers can be represented exactly. This method is extended in the following table, to show how fractional values are represented (the binary point is placed between bits 0 and -1 for clarity):

	Bit												
	3	**2**	**1**	**0**	**.**	**-1**	**-2**	**-3**	**-4**	**-5**	**-6**	**-7**	**-8**
	2^{Bit}												
Decimal	**8**	**4**	**2**	**1**	**.**	**1/2**	**1/4**	**1/8**	**1/16**	**1/32**	**1/64**	**1/128**	**1/256**
0	0	0	0	0	**.**	0	0	0	0	0	0	0	0
1	0	0	0	1	**.**	0	0	0	0	0	0	0	0
10	1	0	1	0	**.**	0	0	0	0	0	0	0	0
0.5	0	0	0	0	**.**	1	0	0	0	0	0	0	0
0.25	0	0	0	0	**.**	0	1	0	0	0	0	0	0
0.5625	0	0	0	0	**.**	1	0	0	1	0	0	0	0
0.1	0	0	0	0	**.**	0	0	0	1	1	0	0	1

As can be seen, the decimal value 0.1 is represented by the binary value 0.00011001_2, but adding these bit values 1/16 + 1/32 + 1/256 gives 0.09765625, which is not exactly 0.1. This clearly illustrates how some values cannot be represented precisely in binary due to the number of digits required exceeding that available[3]. Moreover, in the case of 0.1, the binary encoding repeats indefinitely, so it would be imprecise how ever many bits were available. This is analogous to the inability to represent 1/3 in decimal, but in base 3 it would be represented exactly by 0.1_3. The IEEE 754 single precision standard uses 24 bits for the significand[4] and 8 bits for the exponent, so the most precise encoding for 0.1 is $110011001100110011001100_2 \times 2^{-4}$.

Due to the finite precision of floating-point data the result of a floating-point operation often has to be rounded, and this rounding has an impact on the precision of the final result. Ideally, all products would use IEEE 754 arithmetic, as this would ensure that they all calculated to the same standard. However, due to the hardware used in some controllers, alternative methods are used. For example, the DSP used in the NextMove range of controllers uses a proprietary format built into the hardware of the DSP, and while it has properties in common with IEEE 754, it also has significant differences. One of the main reasons for not using IEEE 754 arithmetic is speed, as the implementation of such things as gradual underflow, unbiased rounding, etc. (which all serve a useful purpose in terms of accuracy), reduce execution speed.

Examples:
The following code illustrates the relative scale problem.

```
Dim x As Float

x = 1000000
Print x = x + 0.5

x = 10000000
Print x = x + 0.5

x = 100000000
Print x = x + 0.5
```

On a NextMove controller, the values 0, 1, and 1 will be displayed, but on a MintDrive*II*, the values 0, 0 and 1 are displayed, indicating that IEEE 754 can discern the difference for longer. Problems can occur when rounding errors produce a result with a limited precision, but which is very close to being precise. This provides results that are apparently correct when displayed, such as 4.0000, 0.2500, etc., but which only appear this way due to the rounding that takes place when outputting to a fixed number of decimals. This is not usually too much of a problem if the results stay in the floating-point domain, but when casting to integer a value that appears to be whole (such as 4.0000) may not be. This sort of problem is heavily implementation dependent (remember that some systems use formats other than IEEE 754), and so a problem that occurs on one system may not occur on another. The following program illustrates this; when executed on a controller from the NextMove family it displays 4.0000 and 3, but when executed on a MintDrive*II* it displays 4.0000 and 4:

```
Dim x As Float = 200
x = x / 50
Print x; Int(x)
```

- -

3. For brevity and to illustrate a point, this table has only 12 bits of precision, half that of the IEEE 754 single precision standard.
4. It actually uses a sign bit with 23 bits for the significand (this term has generally replaced the term “mantissa”) and 8 bits for the exponent. However because the significand is normalized, the first bit is always known to be 1, so there is no point wasting space by storing it, so this hidden bit gives effectively 24 bits of precision.

A related problem can arise due to the propagation of rounding errors, which causes the small errors inherent in the representation of a floating-point value to become magnified by repeating an operation many times. The following program illustrates this, which when executed on a controller from the NextMove family displays the values `pi = 3.1356` and `pi = 3.1416`:

```
Const n = 50000

Dim i As Integer
Dim s As Float

s = 0
For i = 1 To n - 1 Step 2
  s = s + 1 / (2 * i - 1) - 1 / (2 * i + 1)
Next i

Print "pi = ", s * 4

s = 0
For i = n - 1 To 1 Step -2
  s = s + 1 / (2 * i - 1) - 1 / (2 * i + 1)
Next i

Print "pi = ", s * 4
```

Clearly, the second loop calculates pi more accurately, because it calculates the small terms (i.e. the ones with the largest divisor) first, hence allowing them to be treated as significant before the terms get so large that they become swamped.

Note that the above program highlights a deficiency in the DSP used in the NextMove range of controllers, because it does not employ unbiased rounding or allow it to be used, instead always rounding down (i.e. towards -∞). However, if an IEEE 754 emulation library is used the results are both “pi = 3.1416”, but if the rounding mode is set to round down then the values “pi = 3.1356” and “pi = 3.1416” are displayed. This shows that the rounding mode causes the inaccuracy.

These are just a few of the problems inherent in dealing with floating-point arithmetic, which is an integral part of programming, and so the programmer needs to exercise diligence in dealing with such matters.

5.1 Introduction

Mint Basic provides a number of simple declaration types to allow such things as variables and constants to be created for later use. The declaration of subroutines, functions, etc. are also permitted in Mint Basic. However, since these are quite complicated, they are described later in *Modular Programming* starting on page 8-1.

5.2 Constants

Often a program uses a particular value in many places, so it may be beneficial to give this value a name which can be used instead.

Mint Basic includes some pre-defined constants, like `_pi`, and so no formal declaration is required in this case. However, in cases where the constant is more specific, you can provide a name and value within your program. This is done using the `Const` keyword:

```
Const identifier = expression
```

The expression must be composed only of literals or other named constants to enable it to be evaluated by the compiler. Using named constants helps to make a program more readable since it replaces a number with a meaningful name. This avoids the confusion of unexplained 'magic numbers' when returning to the program at a later date. Another benefit is that they also centralize a value to its constant declaration, making it easier to change without having to search through the whole program to replace occurrences of the value. Both of these factors will help to improve the maintainability of the program. It is common practice, though not strictly required, to prefix constant names with a leading underscore, as illustrated by the predefined constant `_pi`. For example, the calculation of the circumference will become:

```
circumference = 2 * _pi * radius
```

Another example where a string constant is required is:

```
Const _escape = "\1b"
Print #_term3, _escape, 'B'
```

The type of a constant is derived from the expression assigned to it, and so it is generally not needed to specify the type in the declaration of the constant. However, it can be specified using the syntax:

```
Const identifier As type = expression
```

The specified type must be compatible with the expression.

5.3 Defines

A Define statement provides the means of substituting something in the place of an identifier that is used in the program. A define can be similar to a constant if it defines a name as being equivalent to a literal, but it is more flexible than this and can allow quite complex substitutions to be made. Try to avoid using defines to represent values, as constants are better suited for this purpose.

A define is created with the Define statement, which is followed by an equals sign and then the text that is to be substituted. Below is an example of one that references an element in the Comms array:

```
Define boxesPerSecond = Comms(12)
```

Using the above declaration is a simple matter of using the identifier boxesPerSecond in a location where Comms(12) would be valid, as is shown below:

```
If boxesPerSecond < 1 Then boxesPerSecond = 1
```

A define may contain references to other defines in its declaration. The following example illustrates this:

```
Define initBoxesPerSecond = boxesPerSecond = 5
```

The above define can confusingly be used in two ways, as an assignment or as an equality test:

```
initBoxesPerSecond
If initBoxesPerSecond Then
  ...
End If
```

A define can be useful for specifying a range of values, for example axes:

```
Define xyAxes = 0, 1
...
VectorA(xyAxes) = 0;
```

Defines must be declared before they are used and are processed in the order in which they are encountered, irrespective of their scope.

5.4 Variables

A variable is a named area of memory where data can be stored. Unlike constants, the value can be changed during execution of the program. The general form of the `Dim` statement is:

```
Dim identifier As type
```

It is good practice to put all the `Dim` statements together at the top of a module after any `Const` and `Define` statements. Avoid using `Dim` in other places, like after non-declaration statements or inside block constructs like `Loop`, etc.

```
Dim counter As Integer   'A good place for a variable declaration

Loop
  Dim found As Integer   'A poor place for a variable declaration
  ...
End Loop
```

The area of memory that is allocated to a variable depends on the class of module in which it is declared, and it can be either allocated a static or a dynamic address. This is an advanced topic, and further details on the modules available can be found in section 8, *Modular Programming*. However, if it is required that a variable that would normally be dynamic has a static address, it should be declared with the `Static` keyword:

```
Static counter As Integer
```

This will not alter the scope of the variable, but will make it behave like a global variable.

5.4.1 Simple

While the data type of literal data is implied in the formatting of the literal (for example 12575 is an integer, 12.0 is a float and “Enter speed” is a string), for variables, the type of storage needs to be specified when the variable is declared, as shown below:

```
Dim x As Float
```

The variable ‘x’ has now been allocated the memory it requires and can now be used to store floating-point values and be used in expressions.

String variables are declared in a similar manner, but due to them being able to contain a variable number of characters, a maximum size can optionally be specified using an asterisk followed by the size, as shown below:

```
Dim s1 As String
Dim s2 As String * 4
```

Variable ‘s1’ may contain up to the default number of characters (usually 64), while variable ‘s2’ may contain no more than four characters.

Variables can be initialized in their declaration by assigning them a value, as shown below:

```
Dim x As Float = 12.25
Dim s1 As String = "Hello world"
Dim s2 As String * 4 = "-1.6"
```

Multiple declarations can be made using one `Dim` statement:

```
Dim i As Integer, x As Float, s As String
```

While this is a useful shorthand notation, care should be taken to give each declaration a data type, avoiding declarations like the one below:

```
Dim i, j, k As Integer
```

The above declaration will create two floating-point variables called ‘i’ and ‘j’, and one integer called ‘k’. This happens because the default data type, if none is specified, is `Float`.

5.4.2 Arrays

If a number of items of data must be stored, then it usually makes sense to store these in a single variable that has multiple elements within it (note that structures can also be used for this purpose, which is discussed in *Structures* on page 5-6). This is called an array, and uses the following type of declaration:

```
Dim dataPoints(1000) As Float
```

The above variable ‘dataPoints’ contains 1000 floating-point values, indexed from 1 to 1000 inclusively. Arrays, like simple scalar variables, can be initialized in their declaration, but since they are an aggregate of values, the initialization should be enclosed in braces `{ }`. The above declaration would then become:

```
Dim dataPoints(1000) As Float = {-0.2, -0.05, 0.6, 0.3, 0.01, 0;}
```

Note how the last initialization value has a semi-colon placed after it, which causes all remaining elements to be set to that value (zero in this case).

To access an individual element of an array, the array variable must be indexed. The examples below show element 16 being assigned and element 750 being read:

```
dataPoints(16) = 100.5
Print dataPoints(750)
```

Note that if an index is used that is outside the range specified in the declaration of the array, an index out of range run-time error (code 3103) will be generated.

Once an array has been indexed, the result behaves exactly as a simple variable of that type and can be used in the same way as a simple variable. For example, an array of floats that has been indexed can be used in an `Input` statement, be passed to a subroutine, etc. An array of structures that has been indexed can have its members accessed, or be assigned another structure, etc.

Arrays can be multi-dimensional, with no practical limit on the number of dimensions allowed. Below are example declarations of a two-dimensional and a three-dimensional array:

```
Dim zRefPoints(2, 64) As Float
Dim lookupTable(16, 2, 8) As Integer
```

The index range, by default, starts from 1 (one), but this can be changed using the `Base` option or by setting the default for this option using the Compiler Options dialog in Mint WorkBench. Using this technique, the only valid bases are zero or one. If zero is used, the array is indexed from zero up to the size specified, inclusively.

```
Option Base 0    'Index arrays from zero
```

Note that this is different to the C/C++ programming language, which specifies the number of elements, but which indexes them from zero, i.e. the last valid index is one less than the size. This can lead to misleading variable declarations like:

```
Dim zRefPoints(1, 63) As Float  'Really a 2x64 array
```

A much clearer and more flexible way of specifying the index range is to use the To keyword in the declaration. The following declares an array that may be indexed from -5 to 5 in the first dimension, and from 0 to 9 in the second dimension:

```
Dim xValues(-5 To 5, 0 To 9) As Float
```

Arrays are stored in a column major format, so the following declaration:

```
Dim n(2, 2) As Integer = {1, 2, 3, 4}
```

will contain the values n(1, 1) = 1, n(2, 1) = 2, n(1, 2) = 3 and n(2, 2) = 4. This can be altered using the `RowMajor` option (see page 7-5), which by default is zero (giving the above behavior), but by setting this to one, the values would be n(1, 1) = 1; n(1, 2) = 2; n(2, 1) = 3; n(2, 2) = 4. Most people find initializing row major arrays more intuitive, as the data is entered row by row rather than column by column.

Note that an array cannot be dynamically resized at run-time.

5.4.3 Memory usage

For safety, arrays are stored in a manner that allows the array bounds to be validated during execution, and so determining the amount of memory used by an array is not a simple matter of multiplying the number of elements by the size of a single element. To achieve this, an array is stored as a header followed by the element data. The header is a sequence of 32-bit words. The first of these contains the number of dimensions, which is followed by a lower and upper bound for each dimension. The following table shows the amount of memory used by different declarations:

Declaration	**Size (bytes)**
`Dim a As Float`	4
`Dim a(10) As Float`	52
`Dim a(2,5) As Float`	60
`Dim a As String`	68
`Dim a(10) As String * 1`	92
`Dim a(10) As String`	692

In general, multiply the number of dimensions by two and add one, then multiply this result by four (because each of the parameters defining the number of dimensions and the lower and upper bound for each is four bytes in size). Add to this the number of elements in the array multiplied by the size of an element in bytes.

5.5 Structures

Structures provide a means of grouping data together, possibly 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.

Structures are declared using the `Structure` keyword:

```
Structure TComplex
  real As Float
  imag As Float
End Structure
```

Here, a structure has been declared called ‘TComplex’ (note the use of a leading 'T' to denote that it is a data type) that contains two members called 'real' and 'imag', both of type `Float`. This declaration does not reserve any memory, as it purely defines a template of the structure’s contents. To create a variable of this type, the `Dim` statement is used, which allocates the required amount of memory:

```
Dim x0 As TComplex, x1 As TComplex
```

In general, structures can be used in most cases where an intrinsic type, like `Float`, would be permitted. For example they can be used in the declaration of arrays, assigned to, and used as parameters. However, structures cannot be returned by a function and they cannot be passed by value.

A structure may be initialized in its declaration by assigning a value to it. Since a structure is an aggregate, like an array, the initialization values must be enclosed within braces `{ }` and there must be a value for each member of the structure. An example of the previous declaration, but this time initialized, is shown below:

```
Dim x0 As TComplex = {10.5, 0.1}, x1 As TComplex = {-2.09, -1.937}
```

To access the contents of a structure variable, the member access operator (a period) must be used. For example, to zero variable ‘x0’, the following code is used:

```
x0.real = 0
x0.imag = 0
```

As previously mentioned, it is possible to use user-defined types in most circumstances, including assignment:

```
x1 = x0
```

Structures may also be nested, for example:

```
Structure T1
  a As Float
  b As Float
End Structure

Structure T2
  c As T1
  d As Integer
End Structure

Dim x As T2
```

```
x.d = 0
x.c.a = 12.6
x.c.b = -0.0625
```

Initialization within the declaration for variable ‘x’ would look like this:

```
Dim x As t2 = {{1.0, 2.0}, 123}
```

Note how the initialization of the nested structure appears within its own set of braces.

Structures may include members that are arrays, and be declared as arrays too:

```
Structure t3
  a As Float
  b As Float
End Structure

Structure t4
  c(2) As t1
  d As Integer
End Structure

Dim y(3) As t4, i As Integer, j As Integer
...

y(i).d = 0
y(i).c(j).a = 12.6
y(i).c(j).b = -0.0625
```

Initialization within the declaration for variable ‘x’ would look like this:

```
Dim y(3) As t4 = {{{{1.0, 2.0}, {3.0, 4.0}}, 123}, _
                  {{{5.0, 6.0}, {7.0, 8.0}}, 456}, _
                   {{{9.0, 10.0}, {11.0, 12.0}}, 789}}
```

Note again how each nested initialization appears within its own braces, and how arrays are treated in a similar manner by enclosing them in braces. As with the initialization of simple arrays, a semi-colon can be used to cause the last value to be repeated for all remaining array elements. To illustrate this, the previous declaration could then become something like this:

```
Dim y(3) As t4 = {{{{0.0, 0.0};}, 123};}
```

The above declaration causes all elements of the array member ‘c’ of ‘t4’ to be set to zero and member ‘d’ set to 123, and this is repeated for all three elements of variable ‘y’. Evidently, nested structures can become complicated, so this type of usage should be limited to cases where it is strictly necessary.

Note that structures are only available when `Option MintV5.5Keywords` (see page 7-4) is enabled, and only for target formats 12 and above.

5.6 Bitfields

Bitfields provide a means of partitioning a sequence of bits such that each partition has a prescribed bit-range and a unique name to allow it to be accessed. Bitfields are declared using the `Bitfield` keyword:

```
Bitfield TCommsData
  parity As 8 To 8
  data As 0 To 7
End Bitfield
```

Here, a bitfield has been declared called ‘TCommsData’ (note the use of a leading ‘T’ to signify that it is a data-type) that contains two members called ‘parity’ and ‘data’. Note that these members do not have a conventional data-type, but instead have their bit range specified using constant values with the `To` keyword. If a member requires only a single bit, then the `To` keyword can be omitted (i.e. `parity As 8` would be valid). Members of a bitfield are of type integer and, with the exception of a member that extends from bit 0 to bit 31, are all unsigned.

The declaration of a bitfield does not reserve any memory, it simply defines a template of its contents. To create a variable of this type, the `Dim` statement is used, which allocates the required amount of memory.

```
Dim x As TCommsData
```

In general, bitfields can be used in most cases where an intrinsic type, like integer, would be permitted. For example they can be used in the declaration of arrays, assigned to, used as parameters, returned by a function and passed by reference or value.

A bitfield may be initialized in its declaration by assigning a value to it. Since a bitfield is just a 32-bit integer, it may be initialized using a single integer value. However, since some structure has been incorporated via members, these may be individually initialized by enclosing their values in braces `{ }` and there must be a value for each member. In the latter case, the initialization is performed in declaration order of the members, which is only significant if members overlap. An example of the above declaration, but initialized, is shown below.

```
Dim x0 As TCommsData = 0
Dim x1 As TCommsData = {0, 0}
```

The members of a bitfield are accessed using the member’s name, which has the semantics of an integer variable, but with a range limited by the bit-range specified in the bitfield’s declaration.

```
If x.parity Then
  ...
End If
```

When writing to a bitfield member, any bits outside the range allowed are masked out, as illustrated below:

```
x.parity = 1   'Writes 1
x.parity = 2   'Writes 0
x.parity = 3   'Writes 1
```

While a bitfield variable has an address, its members define data within that address, and so are not individually addressable. Because of this, passing a member as a reference parameter will require a temporary variable to be used (see *Issues Relating to Reference Parameters* on page 8-3) and so the contents of the bitfield member passed will not be changed. However, a whole bitfield may be passed as a reference parameter, and any changes made will be reflected in the contents of the passed parameter.

Note that bitfields are only available when `Option MintV5.5Keywords` (see page 7-4) is enabled, and only for target formats 14 and above.

5.7 Labels

Labels have two uses:

- To define a location in a program that execution can be directed to using the `GoTo` statement.
- To give a block of code a qualification that can be used by the `Exit` and `Continue` statements.

A label declaration is initiated with the hash (#) character, followed by the name of the label:

```
#myLabel
```

See *GoTo statement* on page 6-14 and *Exit and Continue statements* on page 6-10 for more details on how labels are used.

6.1 Introduction

Mint Basic provides a number of non-declaration statement types, which are termed ‘action statements’. These vary from simple assignment to conditional and repetitive execution. The action statements used for controlling tasks and calling subroutines are discussed in *Modular Programming* starting on page 8-1.

6.2 Assignment

It is a very common task to evaluate an expression and to store the result. This is achieved using the assignment operator, where the expression to the right of the equals sign is evaluated and stored in the variable to the left of the equals sign:

```
variable = expression
```

Typically, the variable is a simple identifier:

```
y = 81.05
y = x
y = a * x ^ 2 + b * x + c
```

The variable assigned can itself be an expression, as is the case when a specific array element is indexed, a structure member accessed, or any combination of these:

```
y(i) = a * x(i) ^ 2 + b * x(i) + c
x.a = (y + z) / 2
z(i).a = 0
w.c(i) = 0
```

If the expression is not the same type as the variable, then it will be automatically cast, if possible (see *Type casting* on page 4-11). The following example shows two assignments, the first where the integer expression is cast to float, and the second where the floating-point expression is cast to integer:

```
Dim x As Float, i As Integer

x = i
i = x
```

Note that casting a float to an integer is performed by truncation, not rounding. The two assignments above are the same as:

```
x = Float(i)
i = Int(x)
```

Data of any type can be used in an assignment statement, so long as it is a compatible type. This includes strings and arrays, examples of which are shown below:

```
Dim s1 As String * 10, s2 As String * 20
s1 = s2
```

Note that in the case of the string assignment, the types are the same but the sizes are different. This is only a problem if the string being assigned contains more characters than the variable can contain, which will result in a ‘string overflow’ run-time error (3109).

Arrays may be assigned in two ways, firstly by assigning a sequence of values, and secondly by assigning another array variable. An example of each of these is shown below:

```
Dim x(10) As Float, y(2, 5) As Float

x = 1, 2, 3, 4, 5, 6, 7;
y = x
```

Note that the assignment of a sequence of values is not enclosed in braces, which are only required if the assignment is in the variable’s declaration. Note also that in the assignment of array ‘x’ to array ‘y’ that the types are the same, but the structure is different. This is not a problem, as the internal storage is just a sequence of values. While the number of elements is the same in this example, if they were different, no error would result; only as many elements as will fit into the destination are copied.

6.3 Commands

A command is a statement that causes a named routine to be executed. This named routine can be a built in command or a user defined command. Built in commands, also called intrinsic commands, include `Print`, `Run`, `Shift`, etc. User defined commands are created by declaring a subroutine and calling it using its name. This is an advanced topic and is discussed in *Subroutines* on page 8-1.

A command is executed by using its name as a statement, as shown below:

```
Run
```

Many commands can accept parameters, which are supplied to the command by enclosing them in brackets and separating each with a comma, as shown below.

```
Run(conveyerTask, glueGunTask)
```

Similarly, for subroutines, the name of the subroutine is used in the same way. An exception to this parameter passing mechanism is that for I/O commands the parameters are not enclosed in brackets:

```
Print #_TERM1, "Products/second = ", 1000 * nProducts / nTime
```

This is discussed in more depth in *Input and output* on page 10-2.

6.4 Control flow

Statements are executed sequentially, and while this is very useful, only the simplest of programs can be written relying on this behavior alone. To do anything truly useful, statements that control the flow of execution must be used. For this purpose, Mint Basic supports a range of common block-structured constructs to allow the order of execution to be precisely controlled. These may be based on conditions being met, or may be unconditional, and may be a once only branch or a form of repetition.

6.4.1 Conditional execution

Conditional execution is used to decide whether to execute one section of code or another, and is always based on one or more conditions.

6.4.1.1 If statement

The `If` statement, in its simplest form, can be read as "if a condition is met, then execute these instructions, otherwise continue execution after the `If` statement". For example:

```
If x < y Then
  x = y
  lessThan = _true
End If
```

This can also be expressed more concisely using a single-line `If` statement:

```
If x < y Then x = y: lessThan = _true
```

It is a common requirement to do something when a condition is true, but otherwise do something else. This is achieved by using the `Else` statement, for example:

```
If x < y Then
  x = y
  lessThan = _true
Else
  lessThan = _false
End If
```

Again, this could be expressed more concisely using a single-line `If` statement, but as the number of statements starts to grow, it makes more sense to use the block notation.

The final form allows multiple conditions to be tested in sequence using the `ElseIf` statement, until one condition is met. For example:

```
If x < y Then
  x = y
ElseIf x <= 1.05 * y Then
  x = (x + y) / 2
Else
  outOfRange = _true
End If
```

The `ElseIf` keyword used above can be replaced with the `Else If` keyword, which is provided for compatibility with older versions of Mint Basic.

The `Select` statement, described below, may be more appropriate for the case of matching a single expression against a number of possible values.

6.4.1.2 Select statement

The `Select` statement is typically used to test an expression for equality against a range of values. The expression and the `Case` values must be compatible types (numeric and string types cannot be mixed) and may be any type that allows relational comparison, i.e. any numeric type (including character) and string[5] are permitted. For example:

```
Select Case LastKey
  Case 'A': x = x + 1
  Case 'B': x = x - 1
End Select
```

This example checks the last key pressed, and if it was ‘A’ it increments variable ‘x’, otherwise if it was ‘B’ it decrements ‘x’. If neither case is matched, nothing is done. Clearly, the same effect could be achieved using an `If` statement, but use of the `Select` statement allows the compiler to generate more efficient code when the tests are against integral values whose range is limited and which do not contain big gaps. Even in non-optimal conditions some improvements are possible (and the worst case is guaranteed to be no worse than using an `If` statement) so it is recommended to use the `Select` statement whenever testing a range of integral values.Note that execution always exits the `Select` statement when a `Case` statement is encountered after the one that matched the `Select` expression. This means that there is no fall-through like in C or C++, which means that the following code executes nothing when `LastKey` is "B":

If it is required to do something if another key was pressed, then a `Case Else` statement can be added:

```
Select Case LastKey
  Case 'A' = x + 1
  Case 'B'
  Case 'C': x = x – 1
  Case Else: Print "Bad selection"
End Select
```

If the intention was to decrement “x" when `LastKey` was “B" or “C", then the following code should be used:

```
Select Case LastKey
  Case 'A': x = x + 1
  Case 'B', 'C': x = x - 1
End Select
```

Note also that the `Case` keyword immediately after the `Select` keyword may be omitted, but should be retained if compatibility with previous versions of Mint Basic is required.

If it is required to do something when any other key was pressed, then a `Case Else` statement can be added:

```
Select Case LastKey
  Case 'A' = x + 1
  Case 'B': x = x - 1
  Case Else: Print "Bad selection"
End Select
```

As the following example illustrates, a range of values can be tested, incrementing ‘x’ if ‘mode’ is 0, and decrementing ‘x’ if ‘mode’ is 1, 3, 4, or 5:

5.String comparison within the `Select` statement is only available in target formats 11 and above.

```
Select Case mode
  Case 0: x = x + 1
  Case 1, 3 To 5: x = x – 1
  Case 2, 6: x = x + 10
  Case 7: x = x – 10
  Case Else: Print "Bad mode"
End Select
```

When the number of statements in each `Case` section increases, it may be convenient to put them each on their own line. In the following example, the previous example has been expanded to increment ‘y’ by 1 when ‘x’ is incremented by 10, and to decrement ‘y’ by 1 when ‘x’ is decremented by 10:

```
Select Case mode
  Case 0
    x = x + 1
  Case 1, 3 To 5
    x = x – 1
  Case 2, 6
    x = x + 10
    y = y + 1
  Case 7
    x = x – 10
    y = y - 1
  Case Else
    Print "Bad mode"
End Select
```

This formatting of the `Select` statement is preferred and will be used in all further examples.

Furthermore, a condition can be tested by using the `Is` operator (see page 4-5). The following example shows how to increment variable ‘x’ when the `Select` expression is 0 to 10, decrement ‘x’ when the `Select` expression is greater than 50, or otherwise zero ‘x’:

```
Select Case (x + y) / 2
  Case 0 To 10
    x = x + 1
  Case Is > 50
    x = x - 1
  Case Else
    x = 0
End Select
```

Finally, it is possible to use non-constant expressions in a `Case` statement:

```
Select Case (x + y) / 2
  Case a To b + 1
    x = x + 1
  Case Is > c
    x = x - 1
  Case Else
    x = 0
End Select
```

Where ‘a’, ‘b’ and ‘c’ are variables or function calls. Note that for a `Case` containing multiple elements (e.g. `Case a, b, c To d, e`), then these are compared with the `Select` expression from left to right until a match is found, any remaining elements being ignored. This is a form of ‘short-circuit’ evaluation, and is not normally a problem unless one of the elements is a function call that causes side effects (see *Side effects* on page 8-7 for more information on this).

Note that the Is operator is only available when Option MintV5.5Keywords (see page 7-4) is enabled, and only for target formats 11 and above.

6.4.2 Repetitive execution

The statements in this category are used to cause statements to be repeatedly executed, either conditionally or unconditionally. It is common in computing to use the term ‘iteration’ to describe repetition.

6.4.2.1 Loop statement

This statement allows unconditional looping, executing repeatedly all the instructions contained within it:

```
Loop
  ...
End Loop
```

Looping continues indefinitely, unless an Exit statement is encountered.

6.4.2.2 Repeat statement

This statement allows conditional looping, executing all the instructions contained within it until the specified condition is met:

```
Repeat
  ...
Until expression
```

Note that one iteration of the loop is always performed, as the condition is at the end of the loop. Looping continues until the condition becomes true (non-zero), unless an Exit statement is encountered. Below is an example of a Repeat statement being used to evaluate π/4:

```
Const n = 50000

Dim i As Integer, s As Float = 0.0

i = n – 1
Repeat
  s = s + 1 / (2 * i - 1) – 1 / (2 * i + 1)
  i = i - 2
Until i < 1
```

6.4.2.3 While statement

This statement allows conditional looping, executing the instructions contained within it while the condition remains true:

```
While expression
  ...
End While
```

Note that it is possible that no iterations will be executed, as the condition is at the top of the loop. Looping continues while the condition remains true (non-zero), unless an Exit statement is encountered. Below is an example of a While statement being used to evaluate π/4:

```
Const n = 50000

Dim i As Integer, s As Float = 0.0

i = n – 1
While i >= 1
  s = s + 1 / (2 * i - 1) – 1 / (2 * i + 1)
  i = i - 2
End While
```

6.4.2.4 For statement

This statement allows a prescribed number of iterations to be performed by specifying a starting value, a finishing value and a step:

```
For counter = start To end Step step
  ...
Next counter
```

Below is an example of a `For` statement being used to evaluate π/4.

```
Const n = 50000
Dim i As Integer, s As Float = 0.0

For i = n - 1 To 1 Step -2
  s = s + 1 / (2 * i - 1) – 1 / (2 * i + 1)
Next i
```

The `For` loop obeys the following rules:

- The counter may be any numeric variable, and while this is usually a simple variable, it can also be an array element, a structure member, or any combination of these.
- It is illegal to jump into a `For` loop, as this would skip its initialization and cause unpredictable behavior. Because of this, an error will be reported by the compiler.
- The counter is initialized to the start value and increments by the specified step prior to starting the next iteration.
- The `Step` specification is optional, and if omitted will default to 1 (one).
- Looping continues until the final value is exceeded, unless an `Exit` statement is encountered or `GoTo` is used to jump outside the loop.
- Once iteration has started, the end value and the step are frozen (i.e. even if they were specified using variables, changing the value of these variables will have no effect on the iterations performed).
- No iterations will be performed if the start value is greater than the end value with a positive step or the start value is less than the end value with a negative step.
- After the loop terminates, the final value of the counter will be equal to the last value used in the body of the loop, i.e. such that adding the step would take it beyond the upper limit.

The `For` loop is handled in one of two ways depending on the firmware revision. In older builds (MVM Library Build 45 and prior), the number of iterations required is calculated ahead of time and only this number of iterations is performed. This strategy results in slightly odd behavior when the loop counter is modified within the loop's body. For this reason, newer firmware (MVM Library Build 46 and above) adopts a strategy that iterates until the final value is exceeded, but leaving the loop counter equal to the last value used in the loop (this final point is a deviation from Visual Basic to aid compatibility with previous versions of Mint Basic).

The following program shows how each strategy functions:

```
Dim i As Integer, f As Float, fnew As Float

'Mint Basic For loop run-time semantics (old)
i = (1000 - 0) / 0.1
f = 0
While i > 0
  f = f + 0.1
  i = i - 1
End While
Print "Mint Basic old"; f

'Mint Basic For loop run-time semantics (new)
f = 0
Loop
  fnew = f + 0.1
  If fnew > 1000 Then Exit
  f = fnew
End Loop
Print "Mint Basic new"; f

'Visual Basic For loop run-time semantics
f = 0
While f < 1000
  f = f + 0.1
End While
Print "Visual Basic"; f
```

The above program produces the following output (on a controller from the NextMove family).

```
Mint Basic old  999.8049
Mint Basic new  999.9048
Visual Basic    1000.0050
```

While the shortfall is usually small, as shown in the above example, the worst case shortfall can be as large as the step size, and the program below illustrates this (when executed on a controller from the NextMove family):

```
Const _fStep = 2 / 3

Dim f As Float, fUpper As Float, i As Integer

For i = 1 To 5
  fUpper = i * _fStep
  For f = 0 To fUpper Step _fStep
  Next f
  Print i; f; fUpper; IIf(Abs(f - fUpper) < 0.01, "Good", "Bad!")
Next i
```

Using MVM Library Build 45 and prior:

```
1       0.6667  0.6667  Good
2       1.3333  1.3333  Good
3       2.0000  2.0000  Good
4       2.6667  2.6667  Good
5       2.6667  3.3333  Bad!
```

Using MVM Library Build 46 and above:

```
1       0.6667  0.6667  Good
2       1.3333  1.3333  Good
3       1.3333  2.0000  Bad!
4       2.6667  2.6667  Good
5       3.3333  3.3333  Good
```

Clearly, neither the old nor the new strategy copes with floating-point data in all cases, and the reasons for this are discussed in the next section.

6.4.2.5 Error propagation

Whenever floating-point arithmetic is performed, there exists the possibility that arithmetic operations can accumulate errors. This is perfectly normal, and is not usually a problem unless the calculation is performed within a loop in such a manner that these small errors accumulate to a significant level, possibly rendering the final result useless. Such errors can occur in statements like:

```
For x = 0 to 1000 Step 0.1
Next x
Print x
```

Because of the differences discussed in the previous section, the above code fragment will either display 999.8029 or 999.9048 on a controller from the NextMove family. The first value occurs because the number of iterations is correctly calculated as being 10000, but due to floating-point limitations the final value falls short. The second value occurs because 999.9048 is within 0.1 of 1000, and so when incremented was beyond the upper limit, hence terminating the loop. In both cases the value is not 1000.0 because floating-point arithmetic is inherently inaccurate, leading to the accumulation of many small errors due to repeatedly adding 0.1 to variable ‘x’.

If an accurate series of 10000 equally spaced values were required in the range 0 to 1000, it would be more accurate to use the code below:

```
For i = 0 To 10000
  x = i / 10
Next i
```

These examples show the simplest forms of error associated with using floating-point arithmetic, and this subject is discussed in more depth in *Floating-point limitations* on page 4-12.

6.4.3 Overriding the natural flow of execution

The conditional and repetitive constructs present in Mint Basic provide a means of specifying the natural flow of execution in a program. Most of the time these constructs are entirely adequate by themselves, but there are occasions when the natural flow of a program needs to be broken. The commands in this section allow this to occur, although they should be used sparingly as their use generally makes a program less readable.

6.4.3.1 Exit and Continue statements

The `Exit` statement is used to immediately jump out of the closest surrounding loop, thus terminating it. The `Continue` statement is used to jump immediately to the end of the closest surrounding loop, thus causing its next iteration to start.

```
Repeat
  ...
```

```
    If expression Then Exit       'Jump to the statement after Until
    If expression Then Continue   'Jump to the Until
    ...
  Until expression
```

The `Continue` statement is often used when reversing the condition and indenting a further level could cause such deep nesting that the program might become harder to read:

```
For i = 1 to n
  'Skip negative elements
  If a(i) < 0 Then Continue

  'Process positive elements
  ...
Next i
```

Compare this with equivalent code that uses a block `If` statement rather than a `Continue` statement.

```
For i = 1 to n
  'Skip negative elements
  If a(i) >= 0 Then
    'Process positive elements
    ...
  End If
Next i
```

The example below illustrates how only the closest surrounding loop is operated on, and will cause execution to jump to the statement immediately after the `End Loop` when the condition in the `If` statement is met:

```
Repeat
  ...
  Loop
    ...
    If expression Then Exit
    ...
  End Loop
  ...
Until expression
```

Methods of getting around this restriction are discussed in the following sections.

6.4.4 Keyword qualification

As was shown in the previous section, it is only possible to exit or continue the closest surrounding loop. This limitation can be avoided by qualifying the `Exit` or `Continue` statement with the keyword of the required loop. This is illustrated by modifying the previous example to make it exit the `Repeat` loop instead:

```
 Repeat
  ...
  Loop
    ...
    If expression Then Exit Repeat
    ...
  End Loop
  ...
Until expression
```

When qualified, the `Exit` statement may also be used to exit a `Select` statement, as shown below:

```
Select Case i
  Case 1:
    ...
  Case 2:
    ...
    If expression Then Exit Select 'Cannot use just Exit
    ...
  Case Else
    ...
End Select
```

As with loops, only the closest surrounding `Select` statement can be exited. The problem of only operating on the closest surrounding block of the specified type is illustrated below, where it is required to exit the outer `Loop` statement, but with no obvious way of achieving this:

```
Loop
  ...
  Loop
    ...
    If expression Then Exit Loop 'Exits inner loop only!
    ...
  End Loop
  ...
End Loop
```

A means of getting around this restriction is discussed in the next section.

6.4.5 Labeled qualification

As shown in the previous section, it is only possible to exit or continue the closest surrounding block matching the specified keyword qualification. This limitation can be avoided by qualifying the required block with a label and using this label in the `Exit` or `Continue` statement (see *Labels* on page 5-10 for details of their syntax). Note that the label used to qualify the block is local to the block and is only visible to the `Exit` and `Continue` statements enclosed in that block.

This is illustrated by modifying the last example from the previous section to make it exit the outer `Loop`:

```
Loop#outer
  ...
  Loop
    ...
    If expression Then Exit Loop outer
    ...
  End Loop
  ...
End Loop
```

Since the loop is uniquely qualified by the label in the above example, the keyword qualification may be dispensed with:

```
Loop#outer
  ...
  Loop
    ...
    If expression Then Exit outer
    ...
  End Loop
  ...
End Loop
```

Labeled qualification may also be used to exit a `Select` statement, but in this case it must be used in conjunction with keyword qualification, otherwise it would try to locate a loop with a matching label:

```
Select Case#i i
  Case 1:
    ...

  Case 2:
    ...
    Select Case j
      Case -1:
        ...

      Case 1:
        ...
        If expression Then Exit Select i 'Cannot use just Exit i
        ...

      Case Else
        ...
    End Select
    ...

  Case Else
    ...
End Select
```

In the example above, note how the outer `Select` statement is labeled with the name 'i', which is the same name as the variable used as the select expression. This is allowed because the block label is not a label declaration (and so will not clash with any other declarations in the program), it is just a name tagged to a block that is visible only to the `Continue` and `Exit` commands.

Note that labeled blocks are only available when `Option MintV5.5Keywords` (see page 7-4) is enabled, and only for target formats 13 and above.

6.4.5.1 Summary

The following points summarize how `Exit` and `Continue` operate.

- `Exit` and `Continue` statements with no qualification operate only on loop blocks and search for the closest surrounding loop block.
- `Exit` and `Continue` statements qualified with only a keyword search for the closest surrounding block with a matching keyword.
- `Exit` and `Continue` statements qualified with only a label operate only on loop blocks and search for the closest surrounding loop block with a matching label.
- `Exit` and `Continue` statements qualified with both a keyword and a label search for the closest surrounding block where both qualifications match.
- The block label is specific to the block it is qualifying and is only visible to the `Exit` and `Continue` statements nested within this block.
- Nested blocks may use the same block label, the ambiguity of doing so being resolved by the above rules.
- Non-nested blocks within the same scope may use the same block label without ambiguity, since they can only be accessed from inside the block.

6.4.6 GoTo statement

The GoTo statement is used to direct execution to another point in the program, namely the location of a label (see *Labels* on page 5-10 for details of their syntax). Below is an example of how they can be used:

```
If Abs(pivot) < assumedZero Then GoTo zeroPivot
...
#zeroPivot
```

The label must be in the same scope as the GoTo statement, hence disallowing the jumping into or out of modules. Additionally, it is illegal to jump into the body of a For loop or a Semaphore block, since they will not yet be initialized for correct operation. The compiler will issue an error if it detects such a misuse.

The GoTo statement should be reserved for unusual circumstances where the natural structure of an algorithm has to be broken. Code that uses GoTo can always be rewritten to avoid its use, though sometimes at the cost of an extra variable or repeated conditional tests. If these additional costs are significant, or the natural clarity of the algorithm is disrupted, then its use might be justified.

6.4.7 Delaying execution

Mint Basic has two statements that delay execution, but without executing any statements. These statements provide a shorthand notation to aid readability and also enable multi-tasking programs to run efficiently.

6.4.7.1 Pause statement

The Pause statement only allows execution to pass through it when the specified condition is met. An example is shown below, which will wait for a key to be pressed on terminal 1.

```
Pause(InKey(_TERM1))
```

It might not be obvious why this statement is useful, as it could have been implemented using one of the repetitive statements, for example:

```
Repeat: Until InKey(_TERM1)
```

However, while this will work, it is not elegant and will cause a multi-tasking program to execute more slowly. The inefficiency arises because, in the time-frame of processing instructions, if the condition is false, it is unlikely to become true during execution of the next instruction, or indeed the next ‘n’ instructions (assuming ‘n’ is small). In this example, it is waiting for a key to be pressed, and since it takes time for the user to notice this and act on it, it makes little sense to continue testing this condition repeatedly. The Pause statement is designed to avoid this situation by handing execution to other tasks if the condition is false, which leads to much improved efficiency in a multi-tasking program.

Note that the condition must be a function of something external to the task that contains the Pause statement or an infinite loop will occur. This is because the Pause statement does not execute any statements, and so cannot itself cause the state of the condition to change. For example, consider a task waiting for a variable to become non-zero, as shown below:

```
Dim counter As Integer = 0

Pause(counter > 0)
Print "Counter greater than zero"
```

This code would loop forever, since variable ‘counter’ will never be greater than zero. However, if another task were added that could alter the state of ‘counter’, then this condition has at least a chance of succeeding:

```
Dim counter As Integer = 0

Run(handleCounter)
Pause(counter > 0)
Print "Counter greater than zero"


Task handleCounter
  Dim i As Integer = 0

  Loop
    If i > 1000 Then
      counter = counter + 1
      i = 0
    End If
    i = i + 1
  End Loop
End Task
```

In the case of `Pause(InKey)`, the condition is a function of a process external even to the Mint Basic program.

6.4.7.2 Wait statement

The `Wait` statement only allows execution to pass through it when the specified number of milliseconds has elapsed, for example:

```
Wait(50)
```

As with the `Pause` statement, you could express this using a repetitive statement, but this would incur the same type of multi-tasking inefficiency, since even in a single millisecond a great many instructions can be executed. Furthermore, it would also require a variable to monitor the passage of time. These are both good reasons to use `Wait`:

```
Time = 0    'Hope nothing else is using this!
Repeat: Until Time >= 50
```

Or, using a temporary variable to avoid modifying `Time`:

```
Dim t0 As Integer
...
t0 = Time
Repeat: Until Time – t0 >= 50
```

Or, using the `Time` data type:

```
Dim t0 As Time
...
t0 = 0
Repeat: Until t0 >= 50
```

Evidently, these are all lengthy and inefficient so should be avoided. If the `Pause` statement is used, then multi-tasking efficiency will be close to optimal, as shown below:

```
Time = 0
Pause(Time >= 50)
```

However, this is still not as concise as the `Wait` statement.

7.1 Introduction

The Mint Basic compiler can be directed to behave in a variety of ways with regard to keyword support, error reporting, code generation and run-time behavior, using the keywords in this section.

The default behavior of the compiler is stored in the registry, and these settings can be adjusted using the Tools, Options…, Advanced Options dialog in Mint WorkBench. Note that these settings are stored for each controller, so changing an option when connected to a controller will only change the settings for that specific controller.

In some of the following sections, options are listed using the format '*Description* (`Option Name`)', where *Description* indicates the text shown in the Tools, Options…, Advanced Options dialog in Mint WorkBench, and (`Option Name`) shows the equivalent Mint program statement (if available). Directive statements in a program always take priority over the defaults set in Mint WorkBench.

7.2 Auto

This statement indicates that the program should start executing automatically on power-up, providing that there are no initialization errors or warnings.

```
Auto
```

The location of the `Auto` statement is unimportant, so long as it appears where a statement is valid, though it will typically be placed in the `Startup` module. Note that the same functionality can be obtained using code 3101 statement:

```
Option Auto 1
```

The benefit of this is that it is possible to be explicit about whether auto-running is required or not rather than having to search the source program for an `Auto` statement. Also, because it is a compiler option, it is possible to make the default setting whatever is required.

7.3 Option

This statement allows an internal option of a given name to be set to a specific value, thus modifying the compiler's behavior. The following example shows how the `InKey` command can be made to return -1 instead of 0 when the input buffer is empty:

```
Option InKeyMode 0
```

`Option` statements must be placed at the outer level, and it is usual to place them at the head of the program.

Information on all code 3101s available is detailed in the following sections.

7.3.1 Compatibility options

The compatibility mode controls an aggregate of options to enable the easy specification of the required compatibility.

Compatibility Mode	Description
0	No compatibility required, i.e. all options will be applied and not be overridden.
5000	Compatibility with MintMT, i.e. all new features disabled.
5400	Compatibility with MintMT, but with new features like structures enabled.
5500	Compatibility with Mint Basic used on *e*100 products.

The compatibility mode can be set in a Mint Basic program using `Option CompatibilityMode`, which can be set to any of the above values. The effect of setting this option is shown in the following table:

	CompatibilityMode		
Option	**5000**	**5400**	**5500**
`Abbreviations`	1	1	1
`BraceUsage`	0	0	1
`CFormatting`	0	1	1
`CharCase`	0	0	0
`ChrReturnsString`	0	0	1
`ErrorRegs`	2	2	1
`LegacyKeywords`	1	1	0
`LegacyParameter`	0	0	1
`MintV5.5Keywords`	0	1	1
`ModuleNesting`	0	1	1
`OptionalParameter`	0	0	1
`PromoteCharsToString`	1	1	0
`ZeroPad`	1	1	0

Because this option controls an aggregate of options, setting one of code 3101s in the above table individually will be overridden if followed by setting the compatibility mode. For example, the following statement sets the compatibility mode to 5400 with the exception of disallowing legacy keywords:

```
Option CompatibilityMode = 5400
Option LegacyKeywords 0
```

However, in the following example, because legacy keywords are disabled prior to setting the compatibility mode, it will be reset, thus allowing their use:

```
Option LegacyKeywords 0
Option CompatibilityMode = 5400
```

Note that under compatibility modes 5000 and 5400, error handling is performed using `Err`, `Erl`, etc., which are automatically primed. This behavior is designed to be compatible with previous versions of Mint Basic, so ERRORREADNEXT, etc., if available, should not be used in either of these compatibility modes.

7.3.2 Keyword support options

The keywords supported by Mint Basic are controlled with these options.

Allow abbreviated keywords (`Option Abbreviations`) controls whether the abbreviated names of MML functions are permitted. Use of abbreviations no longer improves execution speed as it did in version 4 of Mint Basic and before, and since they make programs difficult to read, their use should be avoided.

Legacy keywords (`Option LegacyKeywords`) controls whether short-hand notation used before MintMT is recognized or are treated as valid identifiers. The short-hand notations affected by this option are shown in the following table:

Keyword	**Description**
`a`***n***, `adc`***n***	`ADC(`***n***`)`
`Dint, EInt`	Disable/enable digital input events
`ik`	`InKey`
`i`***n***, `in`***n***	`INX(`***n***`)`
`o`***n***, `out`***n***	`OUTX(`***n***`)`
`rk`	`ReadKey`
`tm`	`Terminal`
`te`	`Time`
`Tron, Troff`	`Trace on/off`
`wt`	`Wait`

Mint v5.5 keywords (`Option MintV5.5Keywords`) controls whether features new to Mint Basic are recognized or are instead treated as valid identifiers.

Keyword	Description
`AndAlso`	Logical conjunction (short-circuit)
`Bitfield..End Bitfield`	User defined bitfield type
`Bool`	Logical affirmation
`CvtIeee2Flt`	Conversion functions for re-mapping the internal representation of numeric types
`CvtInt2Flt`	
`CvtFlt2Ieee`	
`CvtFlt2Int`	
`Echo`	Input echo on/off
`EventPriority`	Allows control of event priorities
`IIf`	Immediate if
`Is`	Case expression relational operator
`IsAlnum`	Is alphanumeric (a-z, A-Z, 0-9)
`IsAlpha`	Is alphabetic (a-z, A-Z)
`IsAscii`	Is ASCII code (0-127)
`IsCntrl`	Is control code (0-31, 127)
`IsDigit`	Is decimal digit (0-9)
`IsLower`	Is lower case (a-z)
`IsUpper`	Is upper case (A-Z)
`IsXDigit`	Is hexadecimal digit (0-9, a-f, A-F)
`OrElse`	Logical inclusive disjunction (short-circuit)
`Shutdown..End Shutdown`	Shutdown module declaration
`Structure..End Structure`	User defined structure type
`Wrap`	Wrap a value to within limits
`WrapOffset`	Shortest offset to a target value within the wrap limits

7.3.3 Code generation options

Array base (`Option Base`) controls whether the lowest index of arrays is 0 (zero) or 1 (one) if it is not specified explicitly. The default is 1 (one).

Allow C format strings (`Option CFormatting`)controls whether the format strings used in the C language are permitted in a `Using` clause.

Chr returns string (`Option ChrReturnsString`) controls whether the `Chr` function returns a string or an integer (flagged for display as a character). The default is 1 (one), which makes it return a string.

Compound parameters (`Option CompoundParameters`) controls the extent to which compound parameters can be used. A setting of 0 (zero) prohibits their use. The default setting of 1 (one) allows their use, but only in the first parameter. A setting of 2 allows their use in any parameter.

Line tracking mode (`Option LineMappings`) controls the method used to keep track of the currently executing line number in a Mint Basic program. A setting of 0 (zero) uses the line instruction. A setting of 1 (one) uses a line mapping table (the default). A setting of 2 provides no line tracking information at all.

Allow dynamic modules inside any static module (`Option ModuleNesting`) controls whether subroutines and functions are allowed to be declared inside all static modules or not. Previous versions of Mint Basic only allowed subroutines and functions to be declared within tasks. The default is to allow dynamic module declarations within all static module types.

Optimization level (`Option OptLevel`) controls the level of optimization used by the compiler. Higher values indicate more optimization. The valid range of values is 0 to 4, with the default being 2. Each value has the following meaning.

0. No optimization.
1. Dead code removal.
2. Structure preserving optimizations.
3. Structure changing optimizations (may give odd behavior when single-stepping).
4. Iteration of all optimizations until no changes are made.

Use ByRef as default parameter passing mechanism (`Option ParametersByRef`) controls whether parameters are passed by reference or by value when it is not specified explicitly. The default is 1 (one), which passes parameters by reference.

Promote characters to string (`Option PromoteCharsToString`) controls whether character data (that enclosed in single quotes) is automatically promoted to a string where appropriate. The default is 0 (zero), which does not promote them.

Static initialisation method (`Option StaticInitialisation`) controls when static variables are initialized. A setting of 0 (zero) only initializes on program download. A setting of 1 (one) initializes on program download and also initializes all static variables each time the parent task is executed. A setting of 2 initializes on program download and also initializes the static variables declared in a module each time the module is executed, except for static variables declared in a dynamic module, which are initialized each time the parent task is executed. A setting of 3 is the same as setting 2, but without the exception for static variables declared in a dynamic module. The default setting is 2. These settings can be viewed in terms of the amount of initialization being performed being at its lowest with a setting of 0 (zero), and gradually increasing with each setting increment.

Enable watch window support (`Option WatchWindowSupport`) controls whether information used to monitor the contents of static variables is included in the executable. The default is 1 (one), which includes the required information.

Default string size in characters (`Option StringSize`) controls the maximum number of characters allowed in a string when it has not been specified explicitly. The default is 64.

Vector table maximum hole size (`Option VectorMaxHoleSize`) controls how large the holes in a vector table can become before they are terminated and a new table is started. When tables become segmented, which can be due to holes or non-constant `Case` expressions, the vector tables are processed in sequence until a match is found. The default value is 2147483647 (i.e. there is no practical limit).

Vector table maximum size (`Option VectorMaxSize`) controls how large the vector table used for Select statement can become. When a vector table exceeds this size, it results in multiple segments, each of which is processed in sequence until a match is found. The default value is 256.

Automatically initialize local variables (`Option ZeroLocals`) controls whether local variables in subroutines and functions (i.e. dynamic variables) are automatically initialized to zero/null on entry. The default is 0 (zero), which leaves them uninitialized.

7.3.4 Error and warning options

Warning levels are `0`, `1` or `2`, with `0` signifying that the warning is ignored, `1` signifying a warning, and 2 promotes a warning to be an error. The global warning level, set with `Option WarningLevel`, only applies to individual warnings that have a level of `1`, i.e. if a warning is disabled (set to 0), then it will not be affected by the global warning level, and if an individual warning is promoted to be an error (set to 2), then it will not be affected if the global warning level is set to 0 or 1.

Brace usage warning level (`Option BraceUsage`) controls whether omitting braces around array initialization data is ignored, warned or flagged as an error. The default is 1 (one).

Case already exists warning level (`Option CaseExists`) controls whether duplicating a `Case` expression is ignored, warned or flagged as in error. The default is 1 (one).

Declaration hidden warning level (`Option DeclarationHidden`) controls whether declarations that hide others are ignored, warned or flagged as an error. The default is 1 (one).

Declaration hides predefined warning level (`Option DeclarationHidesPredefined`) controls whether declarations in a program that share a name with an item in the symbol table is ignored, warned or flagged as an error. The default is 2.

Declaration unused warning level (`Option DeclarationUnused`) controls whether declarations that are unused are ignored, warned or flagged as an error. The default is 1 (one).

Errors per statement (`Option ErrorsPerStatement`) controls the number of errors reported for each statement, and is intended to limit the number of errors produced. The default is 1 (one).

Floating-point equality test warning level (`Option FloatComparison`) controls whether testing floating-point values for equality is ignored, warned or flagged as an error. The default is 1 (one).

Function return variable not set warning level (`Option FunctionReturn`) controls whether not setting the function return variable for all paths through a function is ignored, warned or flagged as an error. The default is 1 (one).

Legacy feature warning level (`Option LegacyFeatures`) controls whether legacy features (like MML functions that have a new name, labeled events, `GoSub`, `Return` and all keywords enabled with the `LegacyKeywords` option, see page 7-4) are allowed and ignored, allowed but warned or flagged as an error. The default is 1 (one).

Legacy parameter warning level (`Option LegacyParameter`) controls whether using dot or square bracket parameters is ignored, warned or flagged as an error. The default is 1 (one).

LHS/RHS parameter mismatch warning level (`Option LhsRhsParameterMismatch`) controls whether mismatches between the left-hand side and right-hand side parameters are ignored, warned or flagged as an error. The default is 1 (one).

Maximum number of errors (`Option MaxErrors`) controls the maximum number of errors that will be reported before compilation is aborted with a “Too many errors” error. The default is 100.

Optional parameter warning level (`Option OptionalParameter`) controls whether omitting an optional parameter is ignored, warned or flagged as an error. The default is 1 (one).

Precision loss warning level (`Option PrecisionLoss`) controls whether operations that might lose precision are ignored, warned or flagged as an error. The default is 1 (one).

Temporary used warning level (`Option  TempUsedInCall`) controls whether operations that require the use of a temporary variable are ignored, warned or flagged as an error. The default is 1 (one).

Global warning level (`Option WarningLevel`) controls the global warning level, which if 0 (zero) causes warnings to be ignored, if 1 (one) keeps warnings as warnings, and if 2 promotes warnings to be errors. The default is 1 (one).

7.3.5 Run-time options

Auto run program on power-up (`Option Auto`) controls whether programs should auto-run on power-up or not. The default is 0 (zero), which means they should not auto-run by default.

Character case (`Option CharCase`) controls whether character data is converted to upper-case, lower-case or left unaltered. The default is 0 (zero).

Error in error event is fatal (`Option ErrorFatal`) controls whether errors that occur in the `ONERROR` event are fatal. The default is 1 (one), which means that errors are fatal in `ONERROR`.

Error registers (`Option ErrorRegs`) controls whether the error registers (`Err`, `Erl`, etc.) are automatically primed prior to entering the `ONERROR` event. The default is 1 (one), which allows their use for pre-*e*100 products only. Setting this to 2 allows their use for all products and a setting of 0 (zero) disables their use for all products.

InKey mode (`Option InKeyMode`) controls whether the `InKey` function returns -1 or 0 (zero) when the input buffer is empty. The default is 1 (one), which makes `InKey` return 0 (zero) when the input buffer is empty.

Maximum concurrent MML calls (`Option MaxMmlCalls`) controls how many MML calls may be made in parallel on products that support this (currently only *e*100 products). Any value between 1 and 15 is allowed, although system stability may be compromised with settings above 2. The default value is 2.

Angles measured in radians (`Option Radians`) controls whether angles are represented in degrees or radians. The default is 0 (zero), which is degrees. Setting this to 1 (one) will cause angles to be measured in radians. Previous versions of Mint Basic only allowed the use of degrees. This option may be useful if compatibility with Visual Basic is required, or radians are more convenient for the application.

Row major arrays (`Option RowMajor`) controls whether arrays are stored as row major or column major. Previous versions of Mint Basic only used column major, which remains the default setting. Note that this option only affects the initialization of arrays and has no impact on how they are indexed in a program.

Scheduler (`Option Scheduler`) controls which scheduler is used to execute programs. The schedulers are:

- 0 (Default) This uses an ageing process that factors the priority by the waiting time for each waiting task (excluding blocked tasks) to decide which to execute next. Blocked tasks wait until an unblocking event occurs, at which point the blocked tasks are processed in turn to remove the blockage, after which normal scheduling continues. This scheduler speed is inversely proportional to the number of executing tasks (excluding blocked tasks).
- 1 This uses probability theory to execute tasks. Each task is allocated a weighting equivalent to the task's priority (set using TaskPriority). The scheduler then randomly picks the next task, which means that tasks with greater priority (weighting) have a greater chance of being chosen. On average, this results in extremely rapid execution for all tasks, while increasing the chance that high priority tasks will be serviced frequently. However, it does not guarantee the frequency at which a particular task will be chosen. This scheduler operates in constant time.

Shutdown behavior (`Option Shutdown`) controls the behavior of the Shutdown module. The default value is 0 (zero) and the setting of each bit has the following effect:

- 0. User break does not run the shutdown module.
- 1. User break and error conditions encountered in the shutdown module do not cause the Mint Break Type to be enforced.
- 2. User break is not allowed when executing the shutdown module.

Zero pad numeric output (`Option ZeroPad`) controls whether numeric output is padded with leading zeroes or spaces. The default is 0 (zero), which pads with spaces.

7.3.6 Configuration options

These options are only adjustable via the Compiler Options dialog, since they are used to configure the compiler rather than to have any specific effect on the program being compiled.

Convert MML functions to uppercase is used to allow the editor to either convert all MML function names to upper case (the default) or to leave them unaltered as they are typed.

Hash table initial size is used to control the initial size of the hash tables used by the compiler. The default is 5.

Hash table maximum size is used to control how large the hash tables used by the compiler are allowed to become. The default is 0, which implies they are unlimited.

Hash table resize threshold is used to specify the average number of items per bucket that if exceeded will cause the hash tables used by the compiler to be resized. The default is 3.

Show compilation statistics is used to control whether a brief summary of the compilation is shown as a diagnostic in the build tab. The default is 0 (disabled).

7.3.7 Listing generation options

These options are only adjustable via the Compiler Options dialog, since they are used to configure the compiler's listing generation facility rather than to have any specific effect during a compilation.

Apply conversions is used to specify whether certain conversions should be applied to the source file. The default is 1 (one). These conversions include:

- Conversion of identifiers to match the character case of their declaration.
- Conversion of legacy keywords to the current standard (e.g. `ik` becomes `InKey`, `a0` becomes `ADC(0)`).
- Conversion of literals to the current standard (e.g. `01101` becomes `2#1101`).
- Conversion of calls to use bracketed parameters, including dot parameters and commands that use unbracketed parameters (e.g `POS.0` becomes `POS(0)` and `mySub a, b, c` becomes `mySub(a, b, c)`).

Bracketing method is used to specify the level of bracketing that is to be used in expressions. The valid range is 0-5, and the default is 3. A setting of 0 will have the minimum number of brackets present that are necessary to represent the expression, higher values introducing progressively more brackets.

Case separation is used to specify the number of blank lines between successive `Case` elements of a `Select` statement. The default is 1 (one).

Comment column alignment is used to specify on what column end-of-line comments are to be aligned. The default is 32.

Convert single-line If statements to block form is used to control whether single-line `If` statements are converted to block `If` form. The default is 0 (disabled).

Indentation depth is used to specify the indentation depth for block-structured constructs. The default is 2.

Module separation is used to specify the number of blank lines between module declarations. The default is 3.

Sort declarations is used to specify whether declarations should be sorted into a prescribed order. The default is 0.

8.1 Introduction

Modular programming is based on the principle of splitting a problem into a number of manageable parts, and then dealing with each of these in isolation. This allows each part to be small enough to comprehend, hence easing its design, implementation and testing. It also allows parts to be re-used within a program, thus avoiding duplication. It is then a matter of writing some code that uses these parts in a meaningful manner, a much simpler task than writing one huge program.

Mint Basic uses the term “module” to describe these components, each of which has unique properties. These modules include the commonly available subroutines and functions, and the less common module types like tasks, events, the `Startup` module and `Shutdown` module.

8.2 Subroutines

Subroutines are used to collect code together that performs a specific purpose. They are useful in situations where the same piece of code requires execution in a number of places, though their use shouldn’t be dictated by this alone; they can still be very useful even if only called from one location.

A subroutine is declared using the `Sub` keyword:

```
Sub mySub()
   ...
End Sub
```

Note the empty brackets after the subroutine’s name, which normally contain parameters, but which must be present even if there are none. Subroutines are called by using their name as a statement:

```
mySub
```

While this is quite useful, subroutines become even more useful when they can receive input data, which is achieved by using parameters. The following code shows how this can be achieved:

```
Sub mySub(x As Float)
   ...
End Sub
```

This allows a single floating-point value to be passed to the subroutine, which it can then use for some purpose. Like subroutines that take no parameters, this subroutine is called using its name as a statement, but with the parameter following it enclosed in brackets:

```
mySub(12.875)
```

Subroutines that accept more than one parameter separate each with a comma. Below is an example of a subroutine that takes many parameters:

```
quadratic(a0, a1, a2, z1, z2)
Print "Solutions are: ", z1, " and ", z2

Sub quadratic(a As Float, b As Float, c As Float, _
              x1 As Float, x2 As Float)
  Dim d As Float

  d = Sqrt(b ^ 2 – 4 * a * c)
  x1 = (-b + d) / (2 * a)
  x2 = (-b - d) / (2 * a)
End Sub
```

The above example shows a subroutine that accepts five parameters, the first three used as input (they are only read), and the last two used as output (they are written to). This allows subroutines to return data, as is shown above, where variables ‘z1’ and ‘z2’ are set to the roots (solutions) of the quadratic.

By default, all parameters are passed by reference, which allows the contents of the passed variable to be altered from inside the subroutine, as was shown in the previous example. Conversely, parameters passed by value use a copy of the value so that any changes made to it within the subroutine will not alter data outside of the subroutine. While parameters are considered as references unless specified otherwise, it is good practice to explicitly state the passing mechanism by prefixing a parameter’s name with either `ByRef` or `ByVal`. The previous example would then become:

```
quadratic(a0, a1, a2, z1, z2)
Print "Solutions are: ", z1, " and ", z2


Sub quadratic(ByVal a As Float, ByVal b As Float, ByVal c As Float, _
              ByRef x1 As Float, ByRef x2 As Float)
  Dim d As Float

  d = Sqrt(b ^ 2 – 4 * a * c)
  x1 = (-b + d) / (2 * a)
  x2 = (-b - d) / (2 * a)
End Sub
```

Doing this allows the compiler to generate more efficient code, and also limits unwanted side effects. For example if the value of one of the parameters ‘a’, ‘b’, or ‘c’ were changed inside the subroutine as part of the computation, then this change would not be propagated back into a variable used in the call (in this case ‘a0’, ‘a1’ and ‘a2’).

Parameters may be of any valid data type, though there are limitations with strings and user-defined types since they must always be passed by reference. It is still beneficial to explicitly specify `ByRef` for these, even though it is the default. With string parameters, it serves no purpose to specify the size of the string (doing so will generate an error), as string parameters inherit their size from whatever is passed when the call is made. String parameters are simply references to another string, and the string they reference will have a size specified (either implicitly or explicitly).

Sometimes, it is necessary to pass an array as a parameter. A parameter is made to represent an array by putting empty brackets after its name. The following is an example of a subroutine that calculates the maximum and minimum of an array:

```
Sub maxMin(x() As Float, ByRef maximum As Float, ByRef minimum As Float)
  Dim i As Integer

  maximum = -1e38 : minimum = 1e38
  For i = LBound(x) To UBound(x)
```

```
    If x(i) > maximum Then maximum = x(i)
    If x(i) < minimum Then minimum = x(i)
  Next i
End Sub
```

Like string and user-defined parameters, array parameters are always passed by reference, so it not strictly necessary to specify `ByRef`, though it is still good practice to do so. As with strings, there is no need to specify the array's index bounds, as they will be inherited from the parameter supplied when the call is made. The compiler validates each call that involves array parameters to ensure that they all use arrays that have the same number of dimensions and are of the same data type.

The default parameter passing mechanism can be adjusted using the compiler option `ParametersByRef`, (see page 7-5) or by setting the default in the Compiler Options dialog in Mint WorkBench.

```
'Pass parameters by value unless specified otherwise
'(or are arrays, strings or structures)
Option ParametersByRef 0
```

Note that if the default is changed to `ByVal`, then this will not alter the passing method used for parameters that must be passed by reference (arrays, strings and structures) in the case of the mechanism not being explicitly specified.

8.2.1 Issues relating to reference parameters

Because reference parameters allow a two way communication of data both into and out of a subroutine or function, their use is slightly more complicated. This arises because to make this two-way mechanism work requires that the parameter is represented as a pointer to its data, and this requires that two conditions be met.

- The passed parameter must be of the same type as the parameter declaration. Clearly, it would make very little sense to have an integer parameter refer to a floating-point variable, as their internal representations are different.
- The passed parameter must represent something that can be assigned to, such as a simple variable, an array element or a structure member. Clearly, it is only possible to copy a value to an expression that represents a memory location.

For each parameter, the Mint Basic compiler checks whether the above conditions have been met, and if so simply passes its address. Otherwise, the only way to make the call is to automatically create a suitable variable, copy the passed expression into it, and finally pass its address. This variable is called a temporary variable, and while they enable the call to be made, the ability to return data is lost, since there is no way to interrogate the contents of a temporary (it is simply a means to an end). Because of this, the compiler generates a warning to bring to the attention of the user that a temporary has been used, as quite often their use signifies a problem with the program's implementation.

For example, assuming the declarations below:

```
Dim a As Float, i As Integer
Dim x(10) As Float

Sub incByTenPercent(ByRef x As Float)
  x = 1.1 * x
End Sub
```

The code below shows calls that do not require temporaries to be used:

```
incByTenPercent(a)
incByTenPercent(x(i))
```

The code below shows calls that will require temporaries to be used, and so in both cases the parameter supplied will not change value (not even the seemingly valid variable ‘i’):

```
incByTenPercent(a + 1)
incByTenPercent(i)
```

A further problem exists for string parameters, as these have a size which can vary depending on the declaration and how many characters are currently stored in the string. The problem is that if a string temporary is required, it assumes the default string size, and this may not be large enough. The following example demonstrates this problem:

```
doComms(_escape + Mid(output, 2))

Sub doComms(ByRef packet As String)
  ...
End Sub
```

The above code will fail with a ‘string overflow’ error if the expression on the first line contains more than the default number of characters in the `packet` string (usually 64). To avoid this happening, declare a string variable that is known to be large enough to contain the string, and use that to make the call:

```
Dim myTempStr As String * 2000
...
myTempStr = _escape + Mid(output, 2)
doComms(myTempStr)

Sub doComms(ByRef packet As String)
  ...
End Sub
```

In general, the use of temporaries should be avoided.

8.3 The concept of locality

The previous section showed how a subroutine is declared and used, and while it showed how variables can be declared inside a subroutine, it did not mention the significance of this. One of the key benefits of modular programming is the ability to write sections of code without needing to worry about how they will interact with other parts of the program. This is achieved through the use of modules, which are simply containers for code. It is then quite reasonable that if a module contains declarations, then those declarations will be local to the module and therefore only visible within it. This idea is applicable to all types of declarations made within a module, and it also applies to parameters too, which can be viewed as local variables whose contents are automatically primed when a call is made.

The term *scope* is used to define the visibility of an identifier. An identifier is 'in scope' if it is accessible at a certain point in the program, and is 'out of scope' if it is not visible at a certain point in the program. The scoping rules are such that a module can see declarations made in enclosing modules. This concept makes it possible to write code safe in the knowledge that no identifiers will conflict with those enclosed within other modules. The following code illustrates the concept of scope:

```
Dim a As String
...
a = ""                     'Modify the String 'a'

Task myTask1
  Dim a As Float
  ...
  a = a / 2                'Modify the Float 'a'

  Sub mySub1()
    Dim a As Time
    a = 0                  'Modify the Time 'a'
    ...
  End Sub

  Sub mySub2()
    a = a + 1              'Modify the Float 'a'
    ...
  End Sub

  Function myFunc(ByVal a As Integer) As Integer
    a = a % 2              'Modify the Integer 'a'
    ...
  End Function
End Task

Task myTask2
  a = "\1b["               'Modify the String 'a'
  ...
End Task
```

The global variable 'a' declared on the first line is different to that declared within 'myTask', which is different again to that declared within 'mySub1', which is different again from that declared within 'myFunc'. Note that the parameter 'a' used in function 'myFunc' is considered a local, and so the name cannot be used in a declaration inside the function (as a variable for example).

To illustrate the concept of declarations made outside a module being visible, but only in an outward direction, the variable 'a' in 'myFunc' is not visible while inside 'mySub1', 'mySub2' or 'myTask1', and the variable 'a' in 'mySub1' is not visible while inside 'myFunc' or 'myTask1'. This offers the potential to allow different programmers to write each of these modules in isolation, as all they need to know is the interface to each module, since the internals are completely private.

Note that scope is only applicable to modules and not to any other block-structured constructs like loops or critical blocks etc. Consequently, a variable declared inside a loop will have the same scope as the loop’s parent module. Such declarations should be avoided.

There is a mechanism for explicitly overriding the natural scoping rules of Mint Basic, and this is discussed later in *Overriding scope* on page 8-20.

8.4 Functions

Functions are very similar to subroutines but return a result, so they are used in expressions. As such, the same rules about local declarations and parameter passing also apply.

Functions are declared using the `Function` keyword, and below is an example of a function that returns the maximum value contained in an array:

```
Function maximum(ByRef x() As Float) As Float
  Dim i As Integer

  maximum = -1e38
  For i = LBound(x) To UBound(x)
    If x(i) > maximum Then maximum = x(i)
  Next i
End Function
```

Since functions are designed to be used in expressions, they must return a result. As with variables, the data type of the function's result must be specified, as shown in the previous example. The mechanism used to specify the result is to assign it to the function's name, which can be viewed as being equivalent to a local variable. The setting of the function's result must be made for all possible paths through the function. Functions cannot return arrays or user-defined types.

8.4.1 Side effects

A side effect is a term used to describe the situation where a variable outside the scope of a module is altered during the execution of the module. Most of the time, this behavior is both intentional and understandable, but in the case of the module being a function it can lead to confusion. Typically, this arises when a function alters a variable external to it, but the result of the function is dependent on this external variable. The following example shows this side effect:

```
Dim a As Float = 0

Print sneaky(0)
Print sneaky(0)

Function sneaky(x As Float) As Float
  sneaky = Cos(x + a)
  a = (a + 1) % 180
End Function
```

The above code will display different values for the two apparently identical calls made.

Side effects in functions should be avoided, as they add significant complication to the verification of a program. Side effects in subroutines should be minimized by passing variables that require their contents to be modified as reference parameters where possible.

8.5 Recursion

Subroutines and functions can be called whenever they are in scope, and so it is allowable that they may call themselves. This self-reactivation is called recursion, and is appropriate when an algorithm is recursively defined. For example, the Mint Basic compiler makes extensive use of recursion, as the elements that make up the language may be nested arbitrarily.

A chain of recursive calls must terminate at some point, so the chain of calls must have a conditional statement at some point that ends the recursion. In the following example that calculates terms of the Fibonacci series, the recurrence is only continued while ‘n’ is greater than 2:

```
Function fib(ByVal n As Integer) As Integer
  If n > 2 Then
    fib = fib(n - 2) + fib(n - 1)
  Else
    fib = 1
  End If
End Function
```

Recursion can be an extremely powerful tool, but there is a penalty for each reactivation. This is because when a subroutine or function is called, data must be stored on a stack to enable continuation when it terminates. In addition to this, memory needs to be allocated to store the local variables used in the function. If the recurrence is continued for too long, then this will result in an ‘out of memory’ run-time error. The example shown has only one item of local data, the integer parameter ‘n’, and so its overheads would appear to be low. However, this function can be easily re-coded to use simple iteration rather than recursion, for example:

```
Function fib(ByVal n As Integer) As Integer
  Dim nextVal As Integer, temp As Integer

  fib = 0
  nextVal = 1
  While n > 0
    temp = fib + nextVal
    fib = nextVal
    nextVal = temp
    n = n - 1
  End While
End Function
```

Although the appearance of the recursive example may look attractive, it takes 2566 ms to calculate the first 20 terms, compared to only 23 ms for the iterative formulation. This example is not particularly realistic, in fact it purposely shows a poor use of recursion, but it provides a very simple illustration of the advantages and disadvantages of recursion. The advantage is that it can represent an algorithm concisely; the disadvantage is potentially slow execution when compared to a comparable iterative solution. This shows that recursion should be reserved for cases where the alternative becomes unacceptably complicated, or when the performance penalty is bearable.

Recursion can be viewed as a means of iterating but without an explicit loop, and some programming languages (of the 'functional' variety) necessitate its use for even the simplest of tasks. For example, to evaluate the maximum value in an array, the following function could be used:

```
Function maximum(a() As Float, ByVal n As Integer) As Float
  maximum = IIf(n = 1, a(n), Max(a(n), maximum(a, n - 1)))
End Function
```

To summate the contents of an array, the following function can be used:

```
Function summate(a() As Float, ByVal n As Integer) As Float
 summate = IIf(n = 1, a(1), a(n) + summate(a, n - 1))
End Function
```

To calculate the greatest common divisor of two values, the following function can be used:

```
Function gcd(ByVal m As Integer, ByVal n As Integer) As Integer
 gcd = IIf(n = 0, m, gcd(n, m % n))
End Function
```

The examples given so far are purposely very simple, but a more useful application of recursion is the quick-sort algorithm:

```
 Sub qsort(a() As Float, ByVal m As Integer, ByVal n As Integer)
  Dim pivot As Float
  Dim i As Integer, j As Integer

  i = m
  j = n
  pivot = a((m + n) \ 2)

  Repeat
    While a(j) > pivot
      j = j - 1
    End While
    While a(i) < pivot
      i = i + 1
    End While
    If i <= j Then
      swap(a(i), a(j))
      i = i + 1
      j = j - 1
    End If
  Until j < i

  If m < j Then qsort(a, m, j)
  If i < n Then qsort(a, i, n)
End Sub
```

It is possible to code all of the examples to use iteration rather than recursion, some of which will be more efficient, but all of which will be more complicated.

8.6 Tasks

Tasks are used to define a section of code that can be executed in parallel with other tasks. This allows distinct processes to be isolated, hence assisting maintainability.

8.6.1 The Parent task

All Mint Basic programs have at least one task, called the Parent task, which is composed of all the statements defined outside of any module type:

```
Dim x As Float

Loop
  Input "x = ", x
  Print "Sqrt(", x, ") = ", Sqrt(x)
End Loop
```

These statements constitute an implicit form of task, almost as if there were a 'Task ParentTask..End Task' surrounding the entire program, but this is not required explicitly, nor is it allowed. An important property of the Parent task is that when it terminates, any other tasks currently executing also terminate. The declaration and manipulation of these other tasks is described in the following sections.

8.6.2 Declaring tasks

A task is declared using the `Task` keyword, for example:

```
Task menu
  ...
End Task
```

As well as being able to have local constants, variables, etc., a task may also have local subroutines and functions. For example:

```
Task menu
  ...
  Sub display()
    ...
  End Sub
  Sub keypress()
    ...
  End Sub
End Task
```

A task terminates either when it runs out of statements to execute, when the `Exit Task` keyword is encountered, when its name is used in an `End task-name` command, or when the parent task terminates. Therefore, if it is required that a task runs continuously then it must use a loop of some sort to continue processing, for example:

```
Task menu
  Loop
    'Handle menu processing in this loop
    ...
  End Loop

  Sub display()
    ...
  End Sub

  Sub keypress()
    ...
  End Sub
End Task
```

Tasks are global, and so must be declared at the outer lever (i.e. they cannot be nested inside other modules).

8.6.3 Starting tasks

A task that has been declared does not execute until told to do so, which is achieved using the Run command:

```
Run(menu)
```

Multiple tasks can be started using one Run command, which is more concise than having a Run command for each task, for example:

```
Run(menu, checkInputs, controlMotion)
```

Note that the Run command initiates execution from the first statement, irrespective of whether the task was terminated, suspended or already running.

8.6.4 Ending tasks

A task normally continues executing until it runs out of statements, but it can be forced to terminate by another task using the End command, for example:

```
End(checkInputs)
```

Like the Run command, multiple tasks can be ended in one statement:

```
End(checkInputs, controlMotion)
```

Note that the End command causes termination irrespective of whether the task was running or suspended. If the task was already terminated End has no effect. Another means of ending a task is by using the Exit Task command:

```
Task checkInputs
  ...
  If ... Then Exit Task
  ...
End Task
```

Clearly, this can only be used within the task being terminated and is equivalent to jumping to the End Task statement, causing the task to terminate.

8.6.5 Suspending tasks

A task's execution can be halted without stopping it completely by using the TaskSuspend command, for example:

```
TaskSuspend(controlMotion)
```

Like the Run command, multiple tasks can be suspended in one statement:

```
TaskSuspend(checkInputs, controlMotion)
```

When a task has been suspended, it remains in a state of limbo until it is told to resume execution by a `TaskResume` command. The `TaskSuspend` command has no effect on a task that is already suspended or terminated.

8.6.6 Resuming tasks

A task that has been suspended can be resumed by using the `TaskResume` command, for example:

```
TaskResume(controlMotion)
```

Like the `Run` command, multiple tasks can be resumed in one statement:

```
TaskResume(checkInputs, controlMotion)
```

Note that the `TaskResume` command is only useful when the task on which it operates is in a suspended state, otherwise it will not do anything.

8.6.7 Testing the status of a task

A task's status can be queried using the `TaskStatus` function. A task can be in one of three states.

- Terminated
- Running
- Suspended

The predefined constants `_tskTerminated`, `_tskRunning` and `_tskSuspended` can be used in the testing process. An example of testing the status of a task is shown below:

```
If TaskStatus(checkInputs) = _tskRunning Then Run(controlMotion)
```

A typical use for this function is to ensure that the parent task does not terminate prematurely, which would also terminate all child tasks:

```
Pause(TaskStatus(menu) = _tskTerminated)
```

8.6.8 Task scheduling

While tasks appear to execute in parallel, in reality they do this by executing a few instructions in one task before moving on to do the same in another task, etc. This is handled by a scheduler, and the scheduler uses two parameters to control this behavior, one to determine when to run a task and the second to determine how long to run it for. These parameters are called the *priority* and the *quantum size*.

Each task has a priority, which determines the frequency at which it is executed. For example, a task with priority 10 will be executed twice as often as a task with priority 5. The precise means by which this is achieved is controlled by `Option Scheduler` (see *Run-time options* on page 7-8). The default priority for all tasks is 10, though this can be adjusted using the TaskPriority command.

```
TaskPriority(ParentTask, 2)
```

Note that the priority specifies a relative priority with respect to other tasks, so in a program with two tasks, setting the priorities to 50 and 100 is exactly the same as setting them to 5 and 10.

In addition to the priority, each task also has a quantum size, which specifies how many instructions to process before task switching. Note that instructions do not equate to Mint Basic statements, but rather to the ‘machine code’ instructions generated by the Mint Basic compiler and executed by the Mint Virtual Machine. The default quantum size for all tasks is 10, though this can be adjusted using the `TaskQuantum` command.

```
TaskQuantum(ParentTask, 1)
```

Note that having a small quantum size results in smoother execution, but the cost of this will be slow execution (because relatively more time is spent task switching). Conversely, a large quantum size will result in jerky execution, causing tasks to get no CPU time for relatively long periods, but overall execution will be fast. The default quantum size provides a good balance between smoothness and speed of execution, and should only be changed when there is a specific need.

8.7 Events

Events are used to handle situations that can occur at any point during program execution. These situations each have a name, and if an event handler is present in the program with a matching name, then it will be automatically called whenever that particular situation occurs.

Events are declared using the `Event` keyword.

```
Event ONERROR
  ...
End Event

Event In0
  ...
End Event
```

Events are global, and so must be declared at the outer lever (i.e. they cannot be nested inside other modules). Like tasks, events may have local subroutine and function declarations, which is desirable if they only have meaning within the event they are declared in.

More information of the events that are supported on a given product is given in the help file, and examples of common event types are presented in *Event handling* on page 10-18.

8.8 Startup module

The `Startup` module declares a section of code that will executed before executing any code in the parent task, and is typically used for configuration purposes.

The `Startup` module is declared as shown below.

```
Startup
   ...
End Startup
```

There can be only one `Startup` module in a Mint Basic program, which is why it does not have a name. The `Startup` module must be declared at the outer level (i.e. it cannot be nested inside other modules).

Note that this is a module declaration rather than an executable statement, so execution does not flow through it like other block-structured constructs such as `Loop..End Loop`, etc. Because of this, it must be placed with the other module declarations, and it is recommended that it is placed out of the way at the end of the program.

Note that no events will be handled and no tasks will execute while the `Startup` module is being processed, and any errors encountered will be fatal (i.e. will terminate execution).

Any child tasks initiated with `Run` inside the `Startup` module will effectively be ignored, as execution of the parent task always starts out with all child tasks in a terminated state. The presence of the `Run` command without parameters in the `Startup` module will cause execution to immediately jump to the first statement of the parent task, this being equivalent to issuing the `Exit Startup` command.

8.9 Shutdown module

The Shutdown module declares a section of code that will be executed whenever the program terminates, and its intended use is to put a machine into a safe state. The Shutdown module is declared as shown below:

```
Shutdown
  ...
End Shutdown
```

There can be only one Shutdown module in a Mint Basic program, which is why it does not have a name. The Shutdown module must be declared at the outer level (i.e. it cannot be nested inside other modules).

Note that this is a module declaration rather than an executable statement, so execution does not flow through it like with other block-structured constructs like `Loop..End Loop`, etc. Because of this, it must be placed with the other module declarations and it is recommended that it is placed out of the way at the end of the program.

Note that no events will be handled and no tasks will execute while the Shutdown module is being processed, and any errors encountered will be fatal (i.e. will terminate execution, but without causing the Shutdown module to be executed again).The behavior of the shutdown module can be controlled using `Option Shutdown`, which is described in *Run-time options* on page 7-8.

Note that the Shutdown module is only available when `Option MintV5.5Keywords` (see page 7-4) is enabled, and only for target formats 14 and above.

8.10 Exiting modules

Modules normally exit when they complete execution of the last statement in their body, at which point `End Sub`, `End Task`, etc. will be encountered. However, it is sometimes desirable to force early termination of a module, and this is achieved using the `Exit` statement, in much the same way as it is used to terminate a block (like a `Repeat` loop). To exit a module, the `Exit` statement must be qualified with the module type, so to exit a task the `Exit Task` command would be used:

```
Task myTask
  ...
  Loop
    ...
    If condition Then Exit Task
    ...
  End Loop
End Task
```

Note that there is one important difference between exiting modules and blocks; it is not possible to specify a module type different to the module enclosing the statement. This means that if you are in a subroutine called from within a task, it is not possible to use `Exit Task` from inside the subroutine. The example below illustrates this illegal usage:

```
Task myTask
  mySub

  Sub mySub()
    ...
    If condition Then Exit Task   'Not allowed!
    ...
  End Sub
End Task
```

The reason this is not allowed is because it is not easy to determine how to safely exit the subroutine before exiting the task. For example, the subroutine may have been called from another subroutine or function, or called from outside the task in which it was declared (using the scope override operator), or may have been called recursively. These are all conditions that are difficult to handle in a safe manner.

The `Exit` command can still be used within a module to exit a block it contains, either unqualified or qualified with the required block type.

8.11 Static and dynamic modules

There are two fundamentally different module types in Mint Basic; the static module and the dynamic module.

8.11.1 Static modules

Static modules are used to represent a section of code that can have only a single instance active at any given time. Static modules must be declared at the outer level, making it illegal to nest them inside any other module type. The static module types supported by Mint Basic are events, tasks, the `Startup` module and the `Shutdown` module, which may all have dynamic modules declared within them.

In static modules, local variables are allocated fixed locations in memory, making them like global variables, but with limited scope. Due to them having fixed locations, the scope override operator may be used to access static variables in another module.

Local variables in a static module are automatically initialized to zero if no explicit initialization is present in its declaration, but this will only occur once when the program is downloaded. A static variable that is initialized will take the specified value on downloading the program and will also be reassigned this value each time execution passes through its declaration.

8.11.2 Dynamic modules

Dynamic modules are used to represent a section of code that can have multiple instances active at any given time. Subroutines and functions are dynamic modules, since they can both be called from different tasks at the same time and can even call themselves. Dynamic modules can be declared inside any static module, including the parent task, but dynamic modules cannot contain dynamic module declarations (i.e. it is illegal to declare subroutines and functions within a subroutine or function).

In dynamic modules, local variables are allocated when required and later freed when no longer needed. As a consequence of this, local variables do not have a fixed memory location, and so the scope override operator cannot be used to access them.

Local variables in dynamic modules are not initialized to zero if no explicit initializer is present in its declaration. This is because there can be a significant performance penalty in doing this, especially with arrays. Typically, the user will initialize local variables as required, and so this duplication of effort is usually a waste.

8.11.3 Lifetime

The lifetime of a variable is the period from the moment it comes into existence to the moment it ceases to exist. This depends on whether it is allocated a static or dynamic address and, as mentioned in the previous sections, this is usually determined by the class of module in which it is declared. However, this can be overridden by declaring the variable using the keyword `Static` rather than `Dim`.

A static variable has a lifetime that is the same as the lifetime of the program. In other words, as soon as a program is downloaded to a controller or drive, a static variable exists and can be used, and only when the program is deleted does its lifetime come to an end.

A dynamic variable has a lifetime equal to the lifetime of the module in which it is declared. In other words, when a dynamic module is activated its dynamic variables are created and can be used, and when the module terminates the lifetime of its local variables comes to an end.

Lifetime should not be confused with scope, because a variable being “live" does not mean that it will be visible (i.e. in scope), though the converse is always true. An example of this is a variable declared in a static module, as this will not be in scope in any other module, though it is “live" and can therefore be accessed from another module using the scope-override operator (`::`). However, not all variables that are “live" can be accessed this way, such as a variable in a subroutine or function (unless it is declared using `Static` rather than `Dim`), as it is in another stack frame (making it hard to locate) and is transient (i.e. its lifetime may expire at any time).

8.12 Overriding scope

The scoping rules of Mint Basic are designed to facilitate modular programming techniques, as discussed in *The concept of locality* on page 8-5. However, it is sometimes desirable to be able to access something directly that is not in the current scope. For example, it may be neater to access a variable local to a task to monitor its state rather than having to make that variable global.

This can be achieved by using the scope override operator, which has the symbol `::` and can be used to access constants, variables, subroutines and functions declared in a static module, or constants and static variables declared in a dynamic module.

```
Conveyor::productsPerSecond = 45

Task Conveyor
  Dim productsPerSecond As Integer
  ...
End Task
```

The above example shows how a variable local to a task can be manipulated with the scope override operator.

It is sometimes required to gain access to the global scope, for example to access a variable whose name has been reused in the current scope. This can be achieved in two ways, so to access global variable 'x', either of the following could be used:

```
ParentTask::x
::x
```

The benefit of the second method, apart from less typing, is that it goes directly to the global scope without the possibility of being blocked by an intermediate declaration. Though being blocked is usually unlikely to occur, the example below shows how it can occur unexpectedly:

```
Task spooler
  Dim counts As Integer
  ...
End Task

Task picker
  Dim spooler As Integer

  If spooler::counts = 0 Then    'Error: spooler is an integer!
  ...
End Task
```

To avoid this, the expression `spooler::counts = 0` needs to be replaced with either `ParentTask::spooler::counts = 0` or `::spooler::counts = 0`. The only time that the latter expression must be used is in the highly unlikely case that a declaration called `ParentTask` exists, thus blocking access to it.

8.13 Task synchronization

Mint Basic provides two mechanisms for synchronizing tasks, the `Critical` block and the `Semaphore` block. The `Critical` block provides a means of protecting a section of code by simply stopping the scheduler from task switching, while the `Semaphore` block provides a means of protecting a section of code by only allowing it to execute if a resource can be acquired.

The section of code being protected by either mechanism is commonly called a *critical section*, and this should not be assumed to be synonymous with a `Critical` block. This potential source of confusion is not confined to Mint Basic, as other programming environments support mechanisms to protect a critical section of code (like the mutex, semaphore and monitor), one of which is also called a critical section.

The `Critical` block is supported by all versions of Mint Basic, while the `Semaphore` block is a relatively new addition. In general, a `Semaphore` block should be used to protect a resource and a `Critical` block should be used to ensure that a section of code is executed speedily. However, because the `Semaphore` block is a relatively new addition, the `Critical` block is often used to protect a resource, and in this case it is advisable to change existing `Critical` blocks to `Semaphore` blocks wherever possible to allow programs to execute more efficiently.

8.13.1 Critical block

Mint Basic allows sections of code to be executed concurrently and for events to interrupt execution (see *Tasks* on page 8-10, and *Events* on page 8-14), but it is sometimes necessary to execute a section of code without allowing it to be interrupted. This is achieved by placing the code within a `Critical` block. Typically, this is used to stop tasks from interfering with each other, but this situation is better handled using semaphores. The `Critical` block is better suited to making a section of code execute as rapidly as possible by inhibiting other tasks (and possibly events) from using processor time. This situation may be the due to a calculation being performed in a dynamic system where higher speed calculation increases the integrity of the result. However, for historical reasons, the examples in this section show the former use.

For example, if the factoring of a variable requires protection from the effects of multi-tasking, then this can be achieved as shown below:

```
Critical
  product = product * 2
End Critical
```

This will ensure that the operation will complete without being interrupted by another task, which may be important if other tasks also modify this variable.

The following example shows how multi-tasking can interfere with the correct operation of a program:

```
Dim product As Integer = 1

Run(factor1, factor2)
Pause(TaskStatus(factor1) = _tskTerminated AndAlso _
      TaskStatus(factor2) = _tskTerminated)
Print "Product = ", product

Task factor1
```

```
    Dim i As Integer

    For i = 1 To 10
      product = 2 * product
    Next i
  End Task

  Task factor2
    Dim i As Integer

    For i = 1 To 10
      product = 2 * product
    Next i
  End Task
```

The above program displays the incorrect value of 16384 rather than 1048576. This happens because the global variable ‘product’ has been read in one task but before it can be factored and the result stored, the other task has written its own result to ‘product’. This will again be overwritten when the first task completes the writing of its result. The code below shows how this can be avoided by enclosing each statement that modifies the value of ‘product’ inside a `Critical` block:

```
Dim product As Integer = 1

Run(factor1, factor2)
Pause(TaskStatus(factor1) = _tskTerminated AndAlso _
      TaskStatus(factor2) = _tskTerminated)
Print "Product = ", product

Task factor1
  Dim i As Integer

  For i = 1 To 10
    Critical
      product = 2 * product
    End Critical
  Next i
End Task

Task factor2
  Dim i As Integer

  For i = 1 To 10
    Critical
      product = 2 * product
    End Critical
  Next i
End Task
```

Sometimes it is necessary to not only inhibit multi-tasking while a statement sequence is being executed, but to also inhibit the processing of events. This is achieved by supplying a bit-pattern of events that are allowed to occur inside the `Critical` block, which is called an ‘event inclusion mask’. Not specifying a value will allow all events to be processed, while a value of zero will stop all events from being processed. For example:

```
Critical
  'All events will get processed here
  ...
End Critical

Critical(0)
  'No events will get processed here
  ...
End Critical
```

```
Critical(_evONERROR Or _evTIMER)
  'Only error and timer events will get processed here
  ...
End Critical
```

Critical blocks may be nested and each may have its own event inclusion mask. The event inclusion mask of a nested Critical block may allow events not allowed in the enclosing Critical blocks. In the following example, the first Critical statement disallows all events and then the program waits for 5000 ms. The program then enters another critical section that allows all events and then waits for 5000 ms. Since events are allowed during this second 5000 ms period, the TIMER event prints the time each second for five seconds:

```
Dim t As Time = 0

Critical(0)      'Disallow both multi-tasking and events
  Wait(5000)
  Critical       'Disallow multi-tasking, but allow events
    Wait(5000)
  End Critical
  Wait(5000)
End Critical

Print "Terminating"

Event TIMER
  Print "Time = ", t
End Event

Startup
  TIMEREVENT = 1000
End Startup
```

When a Critical block exits, either by falling out of the bottom or via a command like Exit, Exit Sub, GoTo, etc., any event inclusion mask will be reset to the correct setting for the new point of execution.

If an error occurs whilst in a Critical block that has masked out the ONERROR event, then one of two things can happen. Firstly, if there is an ONERROR event handler declared in the program, then any errors encountered will be processed when ONERROR becomes active (i.e. when the Critical block that has masked it out exits, or a nested Critical block that allows it is entered). Secondly, if there is no ONERROR event handler present, then execution will terminate immediately.

8.13.2 Semaphore block

The Critical block, discussed in section 8.13.1, provides a means of allowing a section of code to execute without any multi-tasking and optionally without any event processing. This is a simple mechanism that has significant limitations, mainly because it does not allow only specific tasks to be frozen.

A better way of synchronizing tasks is to use a semaphore, which can be considered as a license that must be acquired before a task can proceed. When the task has finished, the semaphore is released, allowing other tasks that require its use to acquire it. When a semaphore cannot be acquired (because it has already been acquired, possibly by the same task that is now trying to acquire it), then the task attempting its acquisition stalls until it becomes free, a state called ‘blocking’. Crucially, while this semaphore acquisition and release is happening, multi-tasking continues for all tasks except those that cannot acquire a semaphore. This makes semaphores a much more efficient means of controlling a multi-tasking program than the `Critical` block, which stalls all tasks whether or not they need to be stalled.

A semaphore is represented in Mint Basic by a variable of type `Semaphore`, which must be declared in the parent task to allow all tasks access to it. The acquisition and release is achieved using a semaphore block, initiated using `Semaphore` and terminated using `End Semaphore`, with the semaphore variable specified as a parameter:

```
Dim s As Semaphore

Semaphore(s)
  ...
End Semaphore
```

A semaphore variable has a size associated with it that represents the number of tasks that can concurrently acquire it. The size is specified using an asterisk followed by an integer constant, which if omitted defaults to one. A size of one is called a binary semaphore (because it has two states, acquired and not acquired) and is very similar to a mutex (an acronym for ‘mutual exclusion’). Examples of semaphore declarations are shown below.

```
Dim a As Semaphore        'Allow one task access at a time
Dim b As Semaphore * 2    'Allow two tasks access at a time
Dim c As Semaphore * 1    'Allow one task access at a time
```

A more complete example follows to show a semaphore in action, with some performance figures to compare it with a `Critical` block and unprotected code:

```
Const _n = 1000

Dim semProduct As Semaphore
Dim product As Integer = 0
Dim t As Time, t0 As Integer, i As Integer = 100

t = 0
Run(task1, task2, counter)
Pause TaskStatus(counter) = _tskTERMINATED
t0 = t
Print "Product = ", product, " in ", t0, "ms"
Print "Counter = ", counter::i, " = ", counter::i / t0, "/ms"

Print


Sub criticalSection()
  i = i + i \ 2
  product = i + product - 1
  i = i + i \ 2
  product = i + product - 1
  i = i + i \ 2
  product = i + product - 1
  i = i + i \ 2
  product = i + product - 1
End Sub


Task task1
  Dim i As Integer

  For i = 1 To _n
    Semaphore(semProduct)
      criticalSection
    End Semaphore
  Next i
```

```
End Task


Task task2
  Dim i As Integer

  For i = 1 To _n
    Semaphore(semProduct)
      criticalSection
    End Semaphore
  Next i
End Task


Task counter
  Dim i As Integer

  i = 0
  Repeat
    i = i + 1
  Until TaskStatus(task1) = _tskTERMINATED AndAlso _
        TaskStatus(task2) = _tskTERMINATED
End Task
```

The above program was executed on a NextMove PCI, and the following data was recorded:

	Result	**Counts**	**Time (ms)**	**Counts/ms**
Unprotected	1126639334	6127	1174	5.22
Critical	754490945	502	599	0.84
Semaphore	754490945	11252	1867	6.03

The unprotected code produces an answer different to protected code, as would be expected, but this is only included to compare execution speed. The `Critical` block is clearly the quickest, but at the expense of slowing down the counter task by over a factor of 6. The `Semaphore` block is the slowest, which is to be expected, as there is an overhead in acquiring and releasing semaphores, but the important point is that it does not slow down other tasks.

When a `Semaphore` block terminates, either by falling out of the bottom of the block or due to an `Exit`, `Continue` or `GoTo` statement that directs execution out of a `Semaphore` block, the semaphore will be released. This is even the case if a task is terminated by another task using `End task-name`, or by Mint WorkBench or a host application.

Note that a `Critical` block can still be used for sections of code that need to execute rapidly, as multi-tasking slows execution. Note that this technique can even be used within a `Semaphore` block, as the strength of a `Semaphore` block is that it allows multi-tasking to continue. However, something that should never be done is the acquisition of a semaphore within a `Critical` block, as this will cause deadlock if the semaphore cannot be acquired.

There are times when it is necessary to perform some other action if a semaphore cannot be acquired, and this can be achieved using an `Else` section within the `Semaphore` block:

```
Semaphore(s)
  ...
Else
  ...
End Semaphore
```

Here, if the semaphore cannot be acquired, then the code contained in the `Else` section is executed. When an `Else` section is present, the critical section of code is between the `Semaphore` and the `Else`, as the statements in the `Else` section are not subject to the semaphore being acquired. If the semaphore must be acquired after executing the code in the `Else` section, then the `Semaphore` block must be placed inside a loop:

```
Loop
  Semaphore(s)
    ...
    Exit Loop
  Else
    ...
  End Semaphore
End Loop
```

There are also times when the acquisition of a semaphore needs to time out, and this can be achieved by specifying the maximum acquisition time in milliseconds:

```
Semaphore(s, 100)
  ...
Else
  ...
End Semaphore
```

This will cause the statements in the `Else` section to be executed if the semaphore 's' cannot be acquired after trying for 100 ms. If no `Else` section is present, then execution continues after the semaphore block.

It is illegal to jump into a semaphore block, as this will skip the acquisition of the semaphore, and because of this an error will be reported by the compiler. Note that the semaphore block is only available when `Option MintV5.5Keywords` (see page 7-4) is enabled, and only for target formats 14 and above.

8.13.3 Deadlock

When two tasks require access to the same resource, but neither can acquire it, the program is said to be in a state of deadlock. While it is possible using a `Critical` block to create a situation where the program freezes, it is much more likely to occur when using the `Semaphore` block. For example, the following program will wait indefinitely at the `Pause` statement in task 'b' because it is in a `Critical` block and the expression is a function of another task, which is not allowed to continue because of the `Critical` block:

```
Dim i As Integer

Run(a, b)
Pause(TaskStatus(a) = _tskTERMINATED AndAlso_
      TaskStatus(b) = _tskTERMINATED)
Print "Finished"


Task a
  For i = 1 to 10000
  Next i
End Task


Task b
  Critical
    Pause(i > 9000)
  End Critical
End Task
```

While it is possible that variable 'i' may be altered by a host application, this is not probable.

The `Semaphore` block is much more likely to cause deadlock problems, which can occur due to the following reasons:

- Trying to acquire a semaphore while in a `Critical` block or an event. This is bad because multi-tasking is frozen while in a `Critical` block or an event, and so if the acquisition fails it will fail for ever (unless a timeout or an `Else` section is used, or a task holding a semaphore is terminated externally, i.e. by a host application).
- The task trying to acquire the semaphore has already acquired it. This is bad because a semaphore is not a re-entrant/recursive lock, so once acquired they will block even if the task trying to acquire it has already previously acquired it.
- Different tasks attempting to cyclically acquire multiple semaphores. This has the potential to be bad if the semaphores are acquired in a particular order, a simple case of which is shown below:

```
Dim x As Semaphore, y As Semaphore, z As Semaphore

Task a
  Semaphore(x)
    Semaphore(y)
      'Execute code
    End Semaphore
  End Semaphore
End Task


Task b
  Semaphore(y)
    Semaphore(z)
      'Execute code
    End Semaphore
  End Semaphore
End Task


Task c
  Semaphore(z)
    Semaphore(x)
      'Execute code
    End Semaphore
  End Semaphore
End Task
```

The above tasks may or may not enter a state of deadlock, and this depends entirely on how the scheduler switches tasks. For example:

- If task 'a' acquires 'x' and 'y' and starts to execute code, then task 'b' cannot acquire 'y'.
- Then task 'c' acquires 'z' but cannot acquire 'x' (already taken by task 'a').
- Then task 'a' releases 'x' and 'y'. This allows task 'b' to acquire 'y', but it still cannot acquire 'z' (already taken by task c).
- Then task 'c' acquires 'x', executes its code, then releases 'x' and 'z'. This allows task 'b' to acquire 'z' and start executing code.
- Finally task 'b' releases 'y' and 'z'.

However, if the scheduling happened differently, then for example:

- Task 'a' acquires 'x', task 'b' acquires 'y', and task 'c' acquires 'z'.

- Task 'a' cannot then acquire 'y', task 'b' cannot acquire 'z' and task 'c' cannot acquire 'x', etc.

Evidently, deadlock may or may not occur depending entirely on the order of scheduling, which is a function of the task priorities, the task quantum sizes and the code in each task. Changes to any of these may cause deadlock, so in situations like this it is safest to assume the worst. There are a few strategies to avoid deadlock:

1. Acquire all resources at once and do not proceed until this is achieved.
2. Release acquired resources on failing to acquire another and attempt to request them later.
3. Enforce that resources are acquired in a particular order.

Since Mint Basic does not support the acquisition of multiple resources in one attempt, option 1 cannot be implemented directly. However, it can be simulated manually using option 2, as shown below:

```
Loop
  Semaphore(a)
    Semaphore(b)
      'Eureka, acquired both semaphores, so execute critical section
      Exit Loop     'This releases semaphores 'a' and 'b'
    Else
      Continue Loop 'This releases semaphore 'a'
    End Semaphore
  End Semaphore
End Loop
```

In the above example, the acquisition of 'a' will either succeed or fail. On acquiring 'a', the acquisition of 'b' will either succeed (allowing the critical section to be entered) or immediately jump to the `Else` section, causing the loop to be continued and implicitly freeing 'a' (remember that jumping out of a `Semaphore` block releases its semaphore).

Option 3 can either be implemented by the programmer ensuring that resource acquisition occurs in a particular order, or, if the algorithm allows it, a `Semaphore` array coupled with a means of acquiring semaphores from it in index order can be used. When a critical section exits, either by falling out of the bottom or via a command like `Exit`, `Exit Sub`, `GoTo` etc., any event inclusion mask will be reset to the correct setting for the new point of execution.

9.1 Introduction

It is sometimes desirable to omit sections of code, perhaps to target one source program to different machine configurations. To achieve this, the keywords `#If`, `#ElseIf`, `#Else` and `#End If` are used to control what code is compiled and what code is ignored, and they behave in a manner very similar to the `If`, `ElseIf`, `Else` and `End If` keywords, but with a few important differences.

- The expression used in a `#If` or `#ElseIf` condition must evaluate to a constant value, and so must be composed solely of literals, constants, or defines that represent a literal or constant[6].
- While the `If` statement can be provided with an expression that can be evaluated at compile time, each section of the `If` statement is carefully examined irrespective of whether it will be executed or not. This means that, for example, you cannot use an unsupported keyword, even in a section that will never be executed. The `#If` statement lifts this restriction, the only limitation being that all sections are syntactically correct.
- The `If` statement cannot be used to enclose alternate declarations with the same name, whereas the `#If` statement can distinguish which should be retained and which should be discarded.
- The `If` statement cannot be used to enclose module declarations, whereas the `#If` statement can.

The terms used in conditional compilation expressions can be declared in the program, but this can be limiting since the program must be changed to target another machine configuration. To overcome this, terms can be specified in Mint WorkBench by clicking Program > Conditional target settings. This allows a number of named configurations to be created, each containing the definitions required to target the program without having to alter it. See the Mint help file.

6.The `#Const` keyword used by Visual Basic to declare pre-processor values is not supported as it is not required due to the implementation used by the Mint Basic compiler.

9.2 Usage

The #If keyword is used to initiate a conditional compilation block. It is followed by an expression that must be composed of values known at compile time, which in turn is followed by the keyword Then and a new line. If the expression is true (non-zero) then its section of code is active and any other sections present are inactive. The #If block is terminated using the #End If keyword, as shown below.

```
#If 1 Then
  'Some code
#End If
```

The previous example used an integer literal, but it is possible to use a constant declared externally in a conditional target, or declared in the program, as shown below:

```
Const _mode = 1
  ...
#If _mode Then
  'Some code
#End If
```

The expression may use operators as required, provided it evaluates to a constant, as shown below.

```
#If _mode = 1 OrElse _mode = 3 Then
  'Some code
#End If
```

It is common to allow an alternative section of code to be compiled when the condition is false, which is achieved using the #Else keyword:

```
#If _mode = 1 OrElse _mode = 3 Then
  'Some code
#Else
  'Some alternative code
#End If
```

It may be required to have a number of alternative sections, which is achieved using the #ElseIf keyword.

```
#If _mode = 1 Then
  'Some code
#ElseIf _mode = 2 Then
  'Some alternative code
#ElseIf _mode = 3 Then
  'Some alternative code
#Else
  'Some alternative code
#End If
```

If it is required to conditionally allow the use of a keyword, the Defined function can be used:

```
#If Defined(KEYWORD) Then
  'Coding using the specified keyword
#Else
  'Alternative coding using other keywords
#End If
```

Note that the `Defined` keyword may only be used in a `#If` or `#ElseIf` condition.

All the above constructs may be nested, and while the examples use constants local to the program, constants defined in a conditional target may be used in exactly the same way.

Two additional constructs can be used to cause warnings and errors to be generated during compilation, which can be useful as reminders and for catching invalid conditional compilation paths. These are called `#Warning` and `#Error` respectively, for example:

```
#If _mode = 1 Then
  ...
  #Warning "To do: This code does not handle failure cases well"
#Else
  #Error "Incorrect mode used"
#End If
```

A number of predefined constants are provided to make it easier to target a program to a specific controller/drive. These include `_platform`, `_NextMovePCI`, etc. (see Reserved words staring on page 13-34). For example:

```
#If _platform = _NextMoveE100 Then
  'Some code
#ElseIf _platform = _MotiFlexE100MintCard Then
  'Some alternative code
#ElseIf _platform = _MotiFlexE100 Then
  'Some alternative code
#Else
  'Some alternative code
#End If
```

Conditional compilation keywords are only available in target formats 13 and above.

9.3 Limitations

When compared with other languages, there are a few limitations with conditional compilation in Mint Basic. These are described in the following sections.

9.3.1 Syntactic correctness

The intention of conditional compilation is to allow a program to be easily targeted using internally or externally declared constants, and the implementation reflects this simple requirement. Additionally, the implementation is designed to not interfere with the automatic listing generation facility. Because of this, the implementation is not based on that of a traditional pre-processor, instead being handled by the syntax analyzer, much like in C#. A consequence of this is that certain forms of conditional compilation constructs are not allowed. Principally, as was mentioned in Usage, this implementation requires that all sections of a conditional compilation block be syntactically correct, and it is this that prohibits certain 'traditional' constructs. For example, the following code is illegal:

```
 #If 0 Then
   Add some code here to implement homing of the gantry axis
#End If
```

But this can easily be recoded to make it syntactically correct:

```
#If 0 Then
  'Add some code here to implement homing of the gantry axis
#End If
```

For example, the following is illegal:

```
#If 1 Then
Function fac(ByVal n As Integer) As Integer
#Else
Function fac(n As Integer) As Integer
#End If
  fac = IIf(n = 0, 1, n * fac(n – 1))
End Function
```

...but can be easily recoded to make it legal:

```
#If 1 Then
  Function fac(ByVal n As Integer) As Integer
    fac = IIf(n = 0, 1, n * fac(n – 1))
  End Function
#Else
  Function fac(n As Integer) As Integer
    fac = IIf(n = 0, 1, n * fac(n – 1))
  End Function
#End If
```

Instances of the illegal constructs shown above should be rare and are easily avoided, as demonstrated.

9.3.2 Numeric conditions

Currently, the condition must be composed of numeric terms, which means that string comparisons, such as the one shown below, are not allowed:

```
 #If _s = "MODE 1" Then
  ...
#End If
```

This restriction should not prove limiting and may be removed in a future version.

9.3.3 Allowed operators

While it is perfectly valid to use expressions in a `#If` condition, the expression may only contain operators, i.e. no function calls are allowed, either intrinsic (e.g. `Log`, `Sin`, etc.) or user-defined (those declared in a program). Certain operators are not allowed, like the redirection operator, `->`, and all terms must be numeric. Brackets may be freely used to order evaluation.

```
#If Log10(_n) = 2 Then   'Illegal, functions not allowed!
```

This restriction should not prove limiting and may be removed in a future version.

9.3.4 Sequencing

The terms used in a conditional expression must be known (which is also a requirement when a pre-processor is used), so the following example is illegal:

```
 #If _n = 1 Then
  ...
#End If

Const _n = 1
```

What is less obvious is that when a constant's value is a function of another constant, then they also need to be declared such that one knows about the other. For example, the following code is illegal:

```
Const _n = _m + 1
Const _m = 0

#If _n = 1 Then
```

This is illegal because the syntax analyzer has tried to evaluate `_m + 1`, but could not because it is a function of `_m`, which is as yet undeclared. This is only a problem for conditional compilation, as the semantic anal yser can determine the value of _n, as by that stage in the compilation process all the terms have been declared and so the evaluator may look them up. However, the syntax anal yser is not so fortunate, as it can only see what has gone before, and so will generate an error if it encounters a term that it does not understand.

10.1 Introduction

Mint Basic has a variety of built-in commands and functions. Since these are an intrinsic part of the language, they are commonly called intrinsic commands and functions, and so are available on all products.

This section categorizes these and gives a description on how each is used.

10.2 Input and output

Mint Basic has a number of commands and functions that facilitate I/O. These commands differ from the other intrinsic commands in the way that they are called:

- A terminal may be specified as the first parameter, which is prefixed with a '#' character. For compatibility with older versions of Mint Basic, the terminal parameter does not need to be separated from following parameters with a comma, though it is good practice to do so.
- Modifiers may be used ahead of parameters to alter the number base. They do not need to be separated from the items around them using a comma.
- The `Using` clause may be used with the `Input`, `Print` and `Line` commands. This clause has parameters itself.

The terminal parameter, or if omitted the `Terminal` keyword, allows a number of terminals to be specified using a bit pattern to represent each terminal. This enables output to be sent to multiple terminals in one statement, and is achieved by combining the `_TERMx` constants using the `Or` operator. For example, to print the message "Error" on terminals 1 and 3, the following command is used:

```
Print #_TERM1 Or _TERM3, "Error"
```

While all the commands / functions below accept a terminal parameter, only the ones that are functions, or can be assigned to, do not use the # suffix to denote the terminal, instead using bracketed notation. The other commands all use the # suffix, as there may be some ambiguity as to whether the first parameter is a terminal or not.

10.2.1 Beep

This command causes the specified terminal to beep.

```
Beep #_TERM2
```

If no terminal is specified, then the `Terminal` bitmap will be used.

10.2.2 Bol

This command causes the cursor to move to the beginning of the line for the specified terminal.

```
Bol #_TERM1
```

If no terminal is specified, then the `Terminal` bitmap will be used.

10.2.3 Echo

This is both a command and a function. As a command, it allows input echoing to be enabled or disabled for a given terminal:

```
Echo(_TERM1) = _false
```

The default setting for each terminal is `_true`, which causes the characters received to be echoed. However, if the input device handles its own echoing, then echoing should be disabled to avoid each character appearing twice.

As a function, Echo allows the input echo setting for the given terminal to be queried:

```
    If Echo(_TERM3) Then
```

Note that this is only available when Option MintV5.5Keywords (see page 7-4) is enabled, and only for target formats 12 and above.

10.2.4 InKey

This function returns a character from the specified terminal.

```
    c = InKey(_TERM2)
```

If no terminal is specified, then the Terminal bitmap will be used.

Note that by default, this function will return zero if the input buffer is empty. If the input is such that it is valid for a null character to be received, then the behavior of this function can be adjusted using Option InKeyMode (see page 7-5) to make it return -1 if the input buffer is empty. Using InKey primes LastKey.

10.2.5 Input

This command allows a variable's contents to be set by entering text from an input device like a keyboard or CAN keypad on the specified terminal. This is achieved by specifying the variable that requires setting after the terminal name, optionally preceded by a prompt string, as shown below:

```
    Input #_TERM2, x
    Input #_TERM2, x, y
    Input #_TERM2, "x = ", x
    Input #_TERM2, "x = ", x, "y = ", y
    Input #_TERM1, camData(i)
```

Output modifiers cannot be used with the Input statement.

For floating-point input decimal is always used, but for integer input decimal is assumed unless a base prefix supported by Mint Basic is used. For example, for integer input, if “101” was entered then the resulting value read would be 101, but if “0101” was entered, then the resulting value read would be 5.

The Using clause can be used, which causes input to be constrained to the specified parameters. For floating-point input, the two parameters represent the number of whole digits and the number of decimal digits. For integer input, only the number of whole digits can be specified. A negative value for the number of whole digits causes the sign (+/-) to be placed before the first digit.

```
    Input #_TERM1, position Using(4, 2)
```

When a Using clause is used, input is always in decimal so any leading zeroes present will not cause the value to be treated as binary. If no terminal is specified, then the Terminal bitmap will be used.

10.2.6 LastKey

This function returns the last key read using either `InKey` or `ReadKey`.

```
Select Case LastKey
```

Note that each task has its own copy of the last key that was read, hence making it possible to use `ReadKey` in one task and `InKey` in another with `LastKey` working correctly in each case.

10.2.7 Line

This command is used output to the specified line, which is cleared prior to printing.

```
Line #_TERM1, 4, "x = ", x, " i = ", Hex i Using(4)
```

The `Line` statement allows all the capabilities of the `Print` statement to be used (see `Print #` below for details).

10.2.8 Locate

This command allows the cursor to be positioned on the specified terminal. The cursor location is specified by two comma separated expressions that specify the column and row to move the cursor to:

```
Locate #_TERM1, 10, 4
Locate #_TERM2, i + j, 3 * (k – 2)
```

If no terminal is specified, then the `Terminal` bitmap will be used.

10.2.9 Print

This command takes a terminal parameter followed by a series of comma or semi-colon separated expressions to print. When the semi-colon is used to separate expressions, a tab is printed between them. The final expression may have a comma placed after it to suppress printing of the carriage-return and line-feed. Below are examples of simple print statements:

```
Print #_TERM1
Print #_TERM2, x
Print #_TERM2, "x = ", x
Print #_TERM2, "x = ", x; " y = ", y
Print #_TERM1, "x = ", x,
```

Note that the first statement will just print a carriage return and line feed to terminal 1 and the last statement will suppress the carriage return and line feed. If no terminal is specified, then the `Terminal` bitmap will be used.

An expression may be preceded by an output modifier to control the output format to use. The allowed modifiers are `Bin`, `Dec`, `Hex` and `Sci`, which respectively output in binary, decimal (the default), hexadecimal and scientific notation.

```
Print #_TERM1, "x = ", Sci x
Print #_TERM1, "Axes = ", Bin Axes
```

An expression may have a `Using` clause after it to specify the number of characters to display in front of and behind the decimal point:

```
Print #_TERM2, "Value = ", x Using(4, 6)
Print #_TERM1, "Value = ", Sci x + y Using(6, 3),
```

The Using clause may also use a format string in place of one or two numeric expressions. This allows all the format strings permitted in the C language to be used in Mint Basic:

```
Print #_TERM2, "Value = ", i Using("%12u")
Print #_TERM1, "Value = ", x + y Using("%12.6f"),
```

The first line of the example shows the only way that an integer can be displayed as unsigned. The second line shows a more convenient way of specifying a field-width and number of decimals for floating-point output. The ability to use C format strings is controlled by the CFormatting option (see page 7-5), which must be set to 1 (one) to enable it. See *C Format Strings* on page 13-13 for details.

10.2.10 ReadKey

This function returns the currently depressed key on a CAN keypad node.

```
i = ReadKey(_TERM3)
```

If no terminal is specified, then the Terminal bitmap will be used. Note that this differs from InKey in that keystrokes are not buffered. Using ReadKey primes LastKey.

10.3 Mathematical functions

Mint Basic has a range of mathematical functions to assist in the calculation of complex expressions.

10.3.1 Abs

This function returns the absolute value of the expression (i.e. removes the sign).

```
If Abs(x – y) < 0.5e-6 Then Exit Sub
```

This function is useful for testing the magnitude of a value when the sign of it is unimportant.

10.3.2 Acos

This function returns the arc-cosine of the expression.

```
 x = Acos(z)
```

The expression must be in the range -1 to 1 otherwise an ‘invalid argument’ run-time error (code 3101) will be generated. The result is in the range 0 to 180 degrees.

10.3.3 Asin

This function returns the arc-sine of the expression.

```
x = Asin(z)
```

The expression must be in the range -1 to 1 otherwise an ‘invalid argument’ run-time error (code 3101) will be generated. The result is in the range -90 to 90 degrees.

10.3.4 Atan

This function returns the arc-tangent of the expression.

```
x = Atan(y / x)
```

The result is in the range -90 to 90 degrees.

10.3.5 Atan2

This function returns the arc-tangent in the correct quadrant, by using two numeric arguments.

```
x = Atan2(y, x)
```

The result is in the range -180 to 180 degrees.

10.3.6 Cos

This function returns the cosine of the expression (in degrees).

```
x = Cos((y – z) / 2)
```

The result will be in the range -1 to 1.

10.3.7 Exp

This function returns the exponential of the numeric argument.

```
x = Exp(y)
```

This function is the anti-logarithm of the `Log` function.

10.3.8 Frac

This function returns the fractional part of the numeric argument (i.e. removes the integer part from the floating-point expression).

```
f = Frac(x)
```

10.3.9 Log

This function returns the natural logarithm (base *e*) of the numeric argument.

```
x = Log(y)
```

The argument must be greater than zero, otherwise an ‘invalid argument’ run-time error (code 3101) will be generated.

10.3.10 Log10

This function returns the common logarithm (base 10) of the numeric argument.

```
x = Log10(y)
```

The argument must be greater than zero, otherwise an ‘invalid argument’ run-time error (code 3101) will be generated.

10.3.11 Max

This function returns the largest of the numeric arguments.

```
x = Max(x, y)
```

It is permissible to use arrays as arguments, in which case the largest element of the array will be returned.

```
x = Max(myArray)
```

It is also permissible to supply multiple arrays as parameters.

```
x = Max(myArray1, myArray2)
```

The mixing of array and scalar parameters is also allowed.

```
x = Max(x, y, myArray1, myArray2, 0)
```

There is no practical limit to the number of arguments, and the result type is float if any arguments are of type float, otherwise integer. Note that this is only available when `Option MintV5.5Keywords` (see page 7-4) is enabled, and only for target formats 14 and above.

10.3.12 Min

This function returns the smallest of the numeric arguments.

```
x = Min(x, y)
```

It is permissible to use arrays as arguments, in which case the smallest element of the array will be returned.

```
x = Min(myArray)
```

It is also permissible to supply multiple arrays as parameters.

```
x = Min(myArray1, myArray2)
```

The mixing of array and scalar parameters is also allowed.

```
x = Min(x, y, myArray1, myArray2, 0)
```

There is no practical limit to the number of arguments, and the result type is float if any arguments are of type float, otherwise integer. Note that this is only available when `Option MintV5.5Keywords` (see page 7-4) is enabled, and only for target formats 14 and above.

10.3.13 Pow

This function returns the result of raising the first numeric argument to the power of the second numeric argument.

```
x = Pow(y, z)
```

Anything to the power of 0 (zero) gives an answer of 1 (one).

Any value, positive or negative, can be raised to the power of a whole number, either positive or negative, and the result will correctly reflect the number of multiplications performed. For example, `Pow(-2,3)` will give the result -8, as it is equivalent to -2 * -2 * -2, and `Pow(-2,-3)` will give the result -0.125, which is 1 / -8).

If the first parameter is 0 (zero) and the second parameter is negative, then an invalid argument run-time error (code 3101) will be generated, as this is essentially a division by zero.

The first argument must be greater than or equal to 0 (zero) if the second argument is not a whole number, otherwise an ‘invalid argument’ run-time error (code 3101) will be generated. In the case of an error, the value returned is zero.

10.3.14 Rnd

This function returns a pseudo-random number greater than zero but less than 1.

```
x = Rnd()
```

Each time that the `Rnd` function is called, a new value will be returned. The same sequence will be returned each time the controller is power cycled.

10.3.15 Round

This function returns the result of rounding the numeric argument to the nearest integer.

```
i = Round(x)
```

Note that if the argument is outside of the range of an integer, then an ‘invalid argument’ run-time error (code 3101) will be generated. This function can also return a floating-point value rounded to a given number of decimal places. For example, if variable ‘x’ contains 18.383539 then the statement below will assign 18.384 to variable ‘f’.

```
f = Round(x, 3)
```

If a number of decimal places of zero is specified, then it will round to the nearest whole value. If -1 is used it will round to the nearest 10, and if -2 is used it will round to the nearest 100, etc. Note that the ability to round to a given number of decimal places is only available when `Option MintV5.5Keywords` (see page 7-4) is enabled, and only for target formats 14 and above.

10.3.16 Sgn

This function returns the sign of the supplied numeric argument, which is -1 if negative, 0 if zero, and +1 if positive.

```
i = Sgn(x)
```

10.3.17 Sin

This function returns the sine of the expression (in degrees).

```
x = Sin((y – z) / 2)
```

The result will be in the range -1 to 1.

10.3.18 Sqrt

This function returns the square root of the numeric argument.

```
x = Sqrt((x – 1) / y)
```

The argument must not be negative or an ‘invalid argument” run-time error (code 3101) will be generated.

10.3.19 Tan

This function returns the tangent of the expression (in degrees).

```
x = Tan((y – z) / 2)
```

The result will be in the range $-\infty$ to ∞.

10.4 Type conversion

Mint Basic includes a number of functions to allow the conversion from one numeric data-type to another, and these functions fall into one of two categories:

- The value preserving transformation.
- The non-value preserving transformation.

The value preserving transformation is most commonly used, and converts the value subject to the precision limitations of the destination type, so integer 185 would become float 185.0, and float 12.875 would become integer 12. The `Int` and `Float` functions fall into this category.

The non-value preserving transformations have a very narrow field of use and are rarely needed. There are two types of non-value preserving transformation:

- Those that simply reinterpret the internal bit-pattern without altering it, such as the `CvtInt2Flt` and `CvtFlt2Int` functions.
- Those that apply a transformation, such as the `CvtIeee2Flt` and `CvtFlt2Ieee` functions (assuming that the native float format is not IEEE 754 compliant).

10.4.1 CvtIeee2Flt

This function is used to convert an integer value encoded in the IEEE 754 single precision format to the native floating-point format used by the controller.

```
f = CvtIeee2Flt(i)
```

This function is useful when receiving floating-point data from an external source that uses IEEE format floats, or when the bits that make up an IEEE float that has been manipulated locally needs to be converted back into a native float. Note that this is only available when `Option MintV5.5Keywords` (see page 7-4) is enabled, and only for target formats 13 and above.

10.4.2 CvtInt2Flt

This function is used to convert the internal bit representation of an integer value into a float.

```
f = CvtInt2Flt(i)
```

This function should not be confused with the `Float` function, which converts the value of an integer expression to its closest representation in the floating-point domain. The `CvtInt2Flt` function simply dupes the compiler into using the bit-pattern of an integer as if it were a float, and is typically used when a bit-pattern that represents a native float needs to be converted back into a float. Note that this is only available when `Option MintV5.5Keywords` (see page 7-4) is enabled, and only for target formats 13 and above.

10.4.3 CvtFlt2Ieee

This function is used to convert a floating-point value encoded in the format used by the controller to an integer value encoded in the IEEE 754 single precision format.

```
f = CvtFlt2Ieee(i)
```

This function is useful when transmitting floating-point data to an external source that uses IEEE format floats, or when the bits that make up an IEEE float need to be manipulated directly. Note that this is only available when `Option MintV5.5Keywords` (see page 7-4) is enabled, and only for target formats 13 and above.

10.4.4 CvtFlt2Int

This function is used to convert the internal bit representation of a floating-point value into an integer.

```
f = CvtFlt2Int(i)
```

This function should not be confused with the `Int` function, which converts the value of a floating-point expression to the integer domain truncated towards zero. The `CvtFlt2Int` function simply dupes the compiler into using the bit-pattern of a float as if it were an integer, and is typically used when the bits that make up a native float need to be manipulated directly. Note that this is only available when `Option MintV5.5Keywords` (see page 7-4) is enabled, and only for target formats 13 and above.

10.4.5 Int

This function returns the integer component of the numeric argument by truncating the fractional part.

```
i = Int(x)
```

Note that if the argument is outside of the range of an integer, an ‘invalid argument’ error (code 3101) will be generated. If it is required to round to the nearest integer, the `Round` function can be used (see page 10-9). This explicit cast should be used in place of implicit casts automatically inserted by the compiler.

10.4.6 Float

This function returns the floating-point equivalent of the numeric argument.

```
x = Float(i)
```

Since the range of a `Float` exceeds that if an `Integer`, no range errors will be generated. However, the precision of a float may result in the value not being as precise as it was in integer domain. This explicit cast should be used in place of implicit casts automatically inserted by the compiler.

10.5 String manipulation

Mint Basic has a range of functions to allow strings to be manipulated.

10.5.1 Asc

This function returns the ASCII code of the first character in the string argument.

```
code = Asc(s)
```

If the string is empty, an invalid argument run-time error (code 3101) will be generated and 0 (zero) will be returned.

10.5.2 Chr

This function returns a single character string of the numeric argument, which is an ASCII code.

```
s = s + Chr(i)
```

Note that if the argument is outside of the range of a character (0-255), an invalid argument error (code 3101) will be generated.

The behavior of this function now conforms with most other forms of the Basic programming language. However, previous versions of Mint Basic were such that the `Chr` function could only be of use in a `Print` statement (to allow an integer to be displayed as a character). If this more limited functionality is required, for example for compatibility reasons, then `Option ChrReturnsString` (see page 7-5) can be set to 0 (zero).

10.5.3 Eval

This function returns the floating-point evaluation of the string argument. This function differs from the `Val` function in that it can evaluate expressions involving variables declared in the program, whereas the `Val` function only converts numeric values.

```
x = Eval(s)
```

It can be made to return an integer value, without loss of precision, using the construct.

```
x = Int(Eval(s))
```

Similarly, it can be made to return a string value, either in floating-point or integer format (the latter without loss of precision) depending on the string contents, using the following construct.

```
x = Str(Eval(s))
```

Like the `Val` function, when returning an integer result and the value encoded in the string exceeds the range of a signed integer, an integer out of range error will be generated (code 3104) and the result will be truncated at either -2^{32} or $2^{31}-1$ depending on its sign. If the string contents cannot be decoded, then an evaluation error (code 3111) will be generated and the error string will indicate the problem.

Note that this is only available when `Option MintV5.5Keywords` (see page 7-4) is enabled, and only for target formats 14 and above.

10.5.4 InStr

This function returns the character location of the first occurrence of one string in another. The first character is at location 1 (one).

```
i = InStr(source, search)
i = InStr(start, source, search)
```

The search starts at the first character if ‘start’ is omitted or at the character specified in ‘start’ if present. The search is case sensitive. Zero is returned if the string is not found.
If ‘start’ is beyond the end of the ‘source’ then 0 (zero) is returned.
If ‘source’ is empty, then 0 (zero) is returned.
If ‘search’ is empty, then ‘start’ is returned or 1 (one) if ‘start’ is omitted.
If ‘start’ is less than or equal to zero then an invalid argument run-time error (code 3101) will be generated and 0 (zero) will be returned.

10.5.5 Left

This function returns the left ‘length’ characters of the string argument.

```
s = Left(s, length)
```

If the string is empty or ‘length’ is 0 (zero), then an empty string will be returned.
If ‘length’ is greater than the length of the string, then the entire string will be returned.
If ‘length’ is negative, then an invalid argument run-time error (code 3101) will be generated and an empty string returned.

10.5.6 Len

This function returns the length of the string argument (i.e. the number of characters it contains).

```
i = Len(s)
```

10.5.7 Mid

This is both a command and a function. As a command, it allows the contents of a string variable to be modified, starting at location ‘start’, and optionally capping the number of characters changed to ‘length’.

```
Mid(s, start) = t
Mid(s, start, length) = t
```

The string ‘s’ being modified can neither be lengthened nor shortened, and the characters written will automatically be capped to enforce this. If ‘length’ is zero, or the string ‘t’ is empty, then the string ‘s’ will remain unaltered. If ‘start’ lies outside the range of characters in ‘s’ (i.e. is less than 1 (one) or greater than `Len(s)`), or ‘length’ is negative, then an ‘invalid argument’ run-time error (code 3101) will be generated and the string ‘s’ is unmodified.

As a function, it returns the middle portion of the string argument, starting at ‘start’ and optionally containing up to ‘length’ characters.

```
s = Mid(s, start)
s = Mid(s, start, length)
```

If ‘start’ is beyond the end of the string or ‘length’ is zero, an empty string will be returned. If ‘start’ is less that 1 (one) or ‘length’ is less than 0 (zero), then an ‘invalid argument’ run-time error (code 3101) will be generated and an empty string returned. Note that the `Mid` *command* is only available when `Option MintV5.5Keywords` (see page 7-4) is enabled, and only for target formats 14 and above.

10.5.8 Right

This function returns the right ‘length’ characters of a string argument.

```
s = Right(s, length)
```

If the string is empty or ‘length’ is 0 (zero), an empty string will be returned. If ‘length’ is greater than the length of the string, then the entire string will be returned. If ‘length’ is negative, then an invalid argument run-time error (code 3101) will be generated and an empty string returned.

10.5.9 Str

This function returns the string equivalent of the numeric argument. If the argument is a float, then a floating-point conversion is performed, otherwise an integer conversion is performed.

```
s = Str(x)
```

This function may take a second parameter to specify the base to use in the conversion. When a base is specified, the numeric argument is converted to a string representing an unsigned integer.

```
s = Str(-14, 16)  's = "4294967282"
```

A negative base may be used, in which case the numeric argument is converted to a string representing a signed integer.

```
s = Str(-14, -16)  's = "-E"
s = Str(-14, -10)  's = "-14"
s = Str(-14, -8)   's = "-16"
s = Str(-14, -2)   's = "-1110"
```

The base may be any value between 2 and 36, or -2 and -36 for signed conversion. An out of range base will cause an invalid argument run-time error (code 3101) to be generated.

Note that the ability to specify the base is only available when `Option MintV5.5Keywords` is enabled, and only for target formats 14 and above.

10.5.10 Val

This function returns the floating-point equivalent of the string argument. This function differs from the `Eval` function in that it can only convert numeric values, whereas the `Eval` function evaluates expressions involving variables declared in the program.

```
x = Val(s)
```

It can be made to return an integer value, without loss of precision, using the construct.

```
x = Int(Val(s))
```

All numeric formats supported by Mint Basic may be converted using this function, as the example below shows.

```
s = "1.234e2" : Print Val(s)       'Displays 123.4000
s = "16#FF" : Print Val(s)         'Displays 255.0000
s = "0xff" : Print Val(s)          'Displays 255.0000
s = "1.234e2" : Print Int(Val(s))  'Displays 123
s = "16#FF" : Print Int(Val(s))    'Displays 255
s = "0xff" : Print Int(Val(s))     'Displays 255
```

This function may take a second parameter specifying the base to use in the conversion, in which case an unsigned conversion will be performed and the result is an integer. Note that when a base is specified, the entire digit sequence is assumed to be in that base and conversion will halt at the first character not in the specified base, as the following example illustrates:

```
s = "332F4BD0" : Print Val(s, 16)    'Displays 858737616
s = "16#332F4BD0" : Print Val(s, 16) 'Displays 22 (number ends at #)
s = "0x332F4BD0" : Print Val(s, 16)  'Displays 0 (number ends at x)
```

The base may be left open by specifying a base of zero, in which case it will behave identically to `Int(Val(s))` in performing a translation based on the contents of the string.

The base may be any value between 2 and 36, or 0 for an open conversion. An out of range base will cause an invalid argument run-time error (code 3101) to be generated.

Like the `Eval` function, when returning an integer result and the value encoded in the string exceeds the range of a signed integer, an integer out of range error will be generated (code 3104) and the result will be truncated at either -2^{32} or $2^{31}-1$ depending on its sign. However, when the base parameter is used (unlike the `Eval` function, which does not have this facility) and the value encoded in the string exceeds the range of an unsigned integer, the result is capped at $2^{32}-1$ (remember that this unsigned value will, by necessity, be stored as a signed value of -1). If the string contents cannot be decoded, then an evaluation error (code 3111) will be generated and the error string will indicate the problem.

Note that the ability to specify the base is only available when `Option MintV5.5Keywords` is enabled, and only for target formats 14 and above.

10.6 Task manipulation

Mint Basic has commands to control the operation of tasks.

10.6.1 End

This command allows task execution to be stopped. When used with no parameters, the `End` command terminates execution of the parent task, which stops the entire program (including all child tasks). When used with specific tasks as parameters, only those tasks will be terminated:

```
End(productCounter)
End(keyMonitor, productCounter)
```

If used on a task that is not running, then no action will result.

10.6.2 Run

This command causes tasks to start execution at their very first statement. When used with no parameters, the `Run` command starts execution of the parent task, but without executing the `Startup` module. To execute the `Startup` module prior to executing the parent task, the `Run Startup` command must be used. When used with a specific task or tasks as parameters, those tasks will be executed.

```
Run(menuSystem)
Run(menuSystem, doorGuard)
```

If used on a task that is already running or suspended, then execution will be stopped prior to immediately restarting it at the first statement in the task. In the case of the parent task (e.g. via `Run`, `Run(Startup)` or `Run(ParentTask)`, all child tasks will be terminated prior to restarting execution.

10.6.3 TaskPriority

This command sets the priority of the specified task

```
TaskPriority(productMonitor, 15)
```

The default priority for all tasks is 10. Valid values are any positive integer value. See *Task Scheduling* on page 8-12 for a description of how the priority influences execution.

10.6.4 TaskQuantum

This command sets the quantum size of the specified task

```
TaskQuantum(productMonitor, 5)
```

The default quantum size for all tasks is 10. Valid values are any positive integer value. See *Task Scheduling* on page 8-12 for a description of how the quantum influences execution.

10.6.5 TaskResume

This command resumes execution of the specified tasks.

```
TaskResume(packageCounter)
```

Resuming a task makes it continue execution from the point it was suspended with the `TaskSuspend` command. If used on a task that is either running but not suspended, or not running, then no action will occur.

10.6.6 TaskStatus

This function returns the status of the task specified in the argument.

```
Pause(TaskStatus(glueGun) = _tskTERMINATED)
```

A task status may be in the following states:

- `_tskTERMINATED` (the task is terminated)
- `_tskRUNNING` (the task is running)
- `_tskSUSPENDED` (the task is suspended)

10.6.7 TaskSuspend

This command suspends execution of the specified tasks.

```
TaskSuspend(packageCounter)
```

Suspending a task halts it immediately. Its status becomes `_tskSUSPENDED` until it is resumed with a `TaskResume` command or restarted with the `Run` command, at which point its status becomes `_tskRUNNING`. Note that suspending the parent task does not cause child tasks to be suspended. If used on a task that is not running or is already suspended, then no action will result.

10.7 Event handling

Mint Basic includes a number of commands and functions to allow events to be controlled.

10.7.1 DInt

This command will disable all digital input events (IN*x*, FASTIN and FASTIN*x*). However, while disabled they will still be pended, and so will be handled as soon as they are re-enabled using EInt.

```
DInt
```

Note that this is provided for compatibility with older versions of Mint Basic, and that the MML function EVENTDISABLE should be used instead.

10.7.2 EInt

This command will enable all digital input events (IN*x*, FASTIN and FASTIN*x*) that were disabled using DInt.

```
EInt
```

Note that this is provided for compatibility with older versions of Mint Basic, and that the MML function EVENTDISABLE should be used instead.

10.7.3 EventPriority

This command sets the order of priority used for processing events.

```
EventPriority(_evONERROR, _evTIMER, _evIN)
```

If an event type is excluded from this command then it will not get processed, even if a handler is present in the program. Note that this is only available when Option MintV5.5Keywords (see page 7-4) is enabled, and only for target formats 13 and above

10.7.4 IPend

This is both a command and a function. As a command it allows the digital inputs of a specified bank to be either pended or unpended.

```
IPend(2) = 2#1110110111
```

As a function, it allows the pending status of a given bank to be read:

```
If IPend(1) And 2#1000 Then Exit Function
```

For compatibility with older versions of Mint Basic, IPend can be used with no parameters, in which case it will use the current value of Bank.

10.7.5 DprEventCode

This function returns the DPR event code used to generate the DPR event, and is most commonly used inside the DPR event.

```
i = DprEventCode
```

Note that this command is not limited to products that have Dual Port RAM, as it can be triggered from a host application using the `DoDPREvent` Mint ActiveX function.

10.8 Error handling

Mint Basic includes functions that allow the error status to be queried. Typically, these are used in the `ONERROR` event. Note that on *e*100 products, these functions cannot be used unless `Option ErrorRegs` is set to 2 (see page 7-8). See the Mint help file for details on how to handle errors if this setting is not used.

10.8.1 Erl

This function returns the line on which the last error occurred.

10.8.2 Err

This function returns the error code of the last error.

10.8.3 ErrAxis

This function returns the axis number of the last error.

10.8.4 ErrStr

This function returns the error string of the last error.

10.9 General purpose

Mint Basic includes a number of additional general purpose commands and functions that do not fit into any specific category.

10.9.1 IsAlnum

This function is used to determine if the parameter is an alphanumeric character, returning 1 (one) if it is and 0 (zero) if it is not.

```
If IsAlnum(x) Then
```

A character is alphanumeric if it is one of ‘a’ to ‘z’, ‘A’ to ‘Z’ or ‘0’ to ‘9’.

Note that this is only available when `Option MintV5.5Keywords` (see page 7-4) is enabled, and only for target formats 14 and above.

10.9.2 IsAlpha

This function is used to determine if the parameter is an alphabetic character, returning 1 (one) if it is and 0 (zero) if it is not.

```
If IsAlpha(x) Then
```

A character is alphabetic if it is one of ‘a’ to ‘z’ or ‘A’ to ‘Z’.

Note that this is only available when `Option MintV5.5Keywords` (see page 7-4) is enabled, and only for target formats 14 and above.

10.9.3 IsAscii

This function is used to determine if the parameter is an ASCII character, returning 1 (one) if it is and 0 (zero) if it is not.

```
If IsAscii(x) Then
```

A character is ASCII if its code is in the range 0 to 127.

Note that this is only available when `Option MintV5.5Keywords` (see page 7-4) is enabled, and only for target formats 14 and above.

10.9.4 IsCntrl

This function is used to determine if the parameter is a control character, returning 1 (one) if it is and 0 (zero) if it is not.

```
If IsCntrl(x) Then
```

A character is a control character if its code is 0 to 31 or 127.

Note that this is only available when `Option MintV5.5Keywords` (see page 7-4) is enabled, and only for target formats 14 and above.

10.9.5 IsDigit

This function is used to determine if the parameter is a decimal digit, returning 1 (one) if it is and 0 (zero) if it is not.

```
If IsDigit(x) Then
```

A character is a digit if it is one of ‘0’ to ‘9’.

Note that this is only available when `Option MintV5.5Keywords` (see page 7-4) is enabled, and only for target formats 14 and above.

10.9.6 IsLower

This function is used to determine if the parameter is a lowercase character, returning 1 (one) if it is and 0 (zero) if it is not.

```
If IsLower(x) Then
```

A character is lowercase if it is one of “a” to “z”.

Note that this is only available when `Option MintV5.5Keywords` (see page 7-4) is enabled, and only for target formats 14 and above.

10.9.7 IsUpper

This function is used to determine if the parameter is an uppercase character, returning 1 (one) if it is and 0 (zero) if it is not.

```
If IsUpper(x) Then
```

A character is uppercase if it is one of ‘A’ to ‘Z’.

Note that this is only available when `Option MintV5.5Keywords` (see page 7-4) is enabled, and only for target formats 14 and above.

10.9.8 IsXDigit

This function is used to determine if the parameter is a hexadecimal digit, returning 1 (one) if it is and 0 (zero) if it is not.

```
If IsXDigit(x) Then
```

A character is a hexadecimal digits if it is one of ‘0’ to ’9’, ‘A’ to ‘F’ or ‘a’ to ‘f’.

Note that this is only available when `Option MintV5.5Keywords` (see page 7-4) is enabled, and only for target formats 14 and above.

10.9.9 LBound

This function returns the lower index bound of the array argument.

```
i = LBound(a)
```

If the array is multi-dimensional, then the required dimension can be specified using an optional second numeric argument.

```
i = LBound(a, 2)
```

This is typically used inside a subroutine or function to enable it to operate on arrays of varying size.

10.9.10 Nop

This command simply consumes a very small amount of time, and is generally used when a sub-millisecond delay is required for timing purposes.

```
Nop
```

10.9.11 Pause

This command halts execution of the task in which it is located until the specified condition becomes true.

```
Pause(COMMS(14) < 10)
```

Note that this has been designed specifically to ensure that multi-tasking performance is kept high by not delaying the execution of any other task while the condition is not met.

10.9.12 Rotate

This is both a command and a function and allows the first parameter to be rotated by the number of bits specified in the second parameter. Rotating to the left is performed with a negative value and rotating to the right is performed with a positive value. The rotation is performed as if the data were unsigned (i.e. the sign bit is considered as part of the data).

As a command, it allows the contents of an integer variable to be rotated the specified number of bits:

```
Rotate(i, 16)
```

As a function, it allows the first numeric argument to be rotated by the number of bits specified in the second numeric argument:

```
i = Rotate(i, 1)
j = Rotate(j, -1)
```

Note that when using the function form, if the argument being rotated is a float, then it will be implicitly cast to an integer, which may result in an integer out of range error (code 3104) if its value lies outside that of an integer.

10.9.13 Shift

This is both a command and a function and allows the first parameter to be shifted by the number of places specified in the second parameter. Shifting to the left is performed with a negative value and shifting to the right is performed with a positive value. The shift is performed as if the data were unsigned (i.e. the sign bit is considered as part of the data).

As a command, it allows the contents of an integer variable to be shifted the specified number of bits:

```
Shift(i, 16)
```

As a function, it allows the first numeric argument to be shifted by the number of bits specified in the second numeric argument.

```
i = Shift(i, 1)
j = Shift(j, -1)
```

Note that when using the function form, if the argument being shifted is a float, then it will be implicitly cast to an integer, and in so doing may result in an integer out of range error (code 3104) if its value lies outside that of an integer.

10.9.14 Time

This is both a command and a function. As a command, it allows the millisecond timer to be set to a given value (usually zero).

```
Time = 0
```

Typically, this is used to time something or to enforce a time-out period on an operation. Note that if a floating point value is assigned, then it will be implicitly cast to an integer, and in so doing may result in an integer out of range error (code 3104) if its value lies outside that of an integer.

Each task has its own copy of `Time`, and so setting the `Time` in one task will have no effect on the `Time` setting in another task. When an event is triggered, it also has its own copy of `Time`, which will be primed to be the same as that used in the parent task.

As a function, it allows the `Time` to be read:

```
Pause(NODELIVE(_busCANOPEN, 4) OrElse Time > 5000)
```

Note that this is provided for compatibility with older versions of Mint Basic, and that variables of type `Time` should be used instead (see page 3-8):

```
Dim timeTaken As Time

timeTaken = 0
Pause(NODELIVE(_busCANOPEN, 4) OrElse timeTaken > 5000)
```

Using variables of type `Time` allows many unique time variables to be declared and also provides a much simpler interface between tasks and events by using the normal scoping rules of the language, as shown below.

```
Dim t As Time
...

Event In0
  t = 0
  ...
End Event
```

The ability to set the time in an event and read its value in other tasks and events is not possible if the `Time` command/function is used.

10.9.15 UBound

This function returns the upper index bound of the array argument.

```
i = UBound(a)
```

If the array is multi-dimensional, then the required dimension can be specified using an optional second numeric argument:

```
i = UBound(a, 2)
```

This is typically used inside a subroutine or function to enable it to operate on arrays of varying size.

10.9.16 Wait

This command delays execution for the specified number of milliseconds.

```
Wait(1000)
```

Note that this command is designed to limit the impact on the performance of other tasks currently executing, so should be used in preference to loops that wait for a timer to reach a given value.

Note that if a floating point argument is used, then it will be implicitly cast to an integer, and in so doing may result in an integer out of range error (code 3104) if its value lies outside that of an integer.

10.9.17 Wrap

This function maps a value into the specified domain by adding or subtracting multiples of the domain range until the value falls inside the range (it actually uses a more elegant method than this, but the concept is valid). This is useful for rotary axes, where the position wraps at each revolution, i.e. 365° is equivalent to 5°, and -5° is equivalent to 355°, etc.

For example, to wrap ‘x’ to lie within the range 0° to 360°, the following command is used.

```
x = Wrap(x, 0, 360)
```

The value being wrapped may be many multiples of the wrap range outside the wrap limits. Note that this is only available when `Option MintV5.5Keywords` (see page 7-4) is enabled, and only for target formats 14 and above.

10.9.18 WrapOffset

This function calculates the offset required to move from one location to another within the specified wrap limits using the smallest move possible. This is useful for rotary axes, where the position wraps at each revolution, so moving from 355° to 5° would require a 10° movement rather than an -350° movement.

For example, to calculate the distance required to move from ‘x1’ to ‘x2’ within the wrap range 0° to 360° and assign it to ‘dx’, the following command is used.

```
dx = WrapOffset(x1, x2, 0, 360)
```

Both the source and the target may be many multiples of the wrap range outside the wrap limits. Note that this is only available when `Option MintV5.5Keywords` (see page 7-4) is enabled, and only for target formats 14 and above.

10.10 Default parameters

Mint Basic allows certain parameters to be omitted when making certain types of call, such as not specifying the terminal with the `Print` command, or not specifying an axis or axes with an MML call. This might seem useful, but it provides no performance benefit and can make program verification difficult. Consequently, it is not recommended that they be used, and should be viewed purely as an aid to backward compatibility. The commands and functions listed here all allow default parameters to be specified. It should be noted that each task has its own set of default parameters and that events use a copy of the parent task's default parameters (to avoid side-effects).

The compiler option `OptionalParameter` is used to control the use of default parameters; see *Error and warning options* on page 7-7.

10.10.1 Axes

This is both a command and a function. As a command, it allows the default axis bitmap that will be used with MML functions to be specified.

```
Axes(0, 1, 6, 7)
```

Each task has its own copy of `Axes`, so setting the `Axes` in one task will have no effect on the `Axes` setting in another task. When an event is triggered, it also has its own copy of `Axes`, which will be primed to be the same as that used in the parent task.

As a function, `Axes` allows the integer bit pattern that represents the axis string to be read:

```
If Axes And 2#100000 Then Exit Function
```

10.10.2 Bank

This is both a command and a function. As a command, it allows the default digital I/O bank that will be used with MML functions to be specified.

```
Bank = 1
```

Each task has its own copy of `Bank`, and so setting the `Bank` in one task will have no effect on the `Bank` setting in another task. When an event is triggered, it also has its own copy of `Bank`, which will be primed to be the same as that used in the parent task.

As a function, it allows the bank to be read:

```
If Bank = 4 Then Exit Function
```

10.10.3 Bus

This is both a command and a function. As a command, it allows the default bus that will be used with MML functions to be specified.

```
Bus = _busETHERNET
```

Each task has its own copy of `Bus`, and so setting `Bus` in one task will have no effect on the `Bus` in another task. When an event is triggered, it also has its own copy of `Bus`, which will be primed to be the same as that used in the parent task.

As a function, it allows the bus to be read:

```
If Bus = _busCANOPEN Then Exit Function
```

10.10.4 Terminal

This is both a command and a function. As a command, it allows the default terminal that will be used for I/O commands and functions to be specified.

```
Terminal = _TERM1
```

Each task has its own copy of `Terminal`, and so setting the `Terminal` in one task will have no effect on the `Terminal` setting in another task. When an event is triggered, it also has its own copy of `Terminal`, which will be primed to be the same as that used in the Parent task.

As a function, it allows the terminal to be read:

```
If Terminal = _TERM3 Then Exit Function
```

11.1 Introduction

A wide range of functions resident in the controller’s firmware provide direct access to the functionality of the hardware. This functionality allows, amongst other things, digital outputs to be set or interpolated moves to be performed. These functions are collectively called the Mint Motion Library (MML). The functionality available in the MML can vary greatly between different hardware types and even between different firmware builds for the same hardware. As such, the help file should be consulted for the specific details for the product you intend to use.

The MML can be accessed in a variety of ways:

- Mint ActiveX control.
- An embedded application
- Mint Basic.

The Mint ActiveX control is used when writing a host application, typically in Visual Basic or C++. An embedded application is written in C and linked directly to the supplied firmware libraries. Only the use of Mint Basic will be discussed here.

11.2 Overview

An overview of the functionality available within the MML and the syntax used to access it is discussed in the following sections. This includes how to make calls, pass parameters and redirect calls to other controllers.

11.2.1 Call types

There are three types of MML call, the ‘get’ call, the ‘set’ call and the ‘do’ call. Calls that allow the getting of data more often than not also allow the data to be set, while ‘do’ calls are commands that do not get or set a value, they simply perform an action.

Calls that return a value are termed ‘set’ calls, and are called in the same way as a user-defined function by simply using its name in an expression and appending to it any parameters enclosed within brackets. Below is an example of an expression that reads the positions of two axes using the `POS` function (an MML ‘get/set’ function).

```
distance = Sqrt(POS(0) ^ 2 + POS(1) ^ 2)
```

Calls that allow a value to be set are termed ‘get’ calls, and are called using assignment, with the call being on the left-hand side of the assignment operator and the value to be assigned being on the right-hand side. Below is an example showing the position of an axis being set:

```
POS(0) = 0
```

Calls that neither set or get a value are termed ‘do’ calls, and are called in the same way as a user-defined subroutine by simply using the name as a command and appending the parameters enclosed within brackets. Below is an example of triggering motion on axis 0 (zero):

```
GO(0)
```

It is possible for some ‘set’ calls to take more than one right-hand-side value, and for ‘do’ calls to take a variable number of parameters. The ability to take an arbitrary number of parameters is indicated in the help file by a trailing ‘…’ in the parameter list and in the right-hand side expressions for ‘set’ calls. This information is also shown in the Mint WorkBench editor when the mouse cursor is hovered over the keyword. For example:

```
VECTORA(0, 1) = x, y
VECTORA(2, 3) = 0, 0
GO(0, 2)
```

Previous versions of Mint Basic used ‘dot parameters’, where each parameter was appended to the keyword separated by a dot (period). This uncommon approach to parameter passing has its problems, which is why it is not used by any other main-stream programming languages. However, it is supported for compatibility reasons and is described in *Legacy MML parameters* on page 13-4.

11.2.2 Advanced parameter passing

A technique can be used to reduce the amount of code that is required to make a series of MML calls. The `MOVEA` command accepts only a single parameter (and thus a single right-hand side expression), as indicated by no ‘…’ in the help file or the hover-over help. This means a sequence of commands must be used, as shown below:

```
MOVEA(0) = 10
MOVEA(1) = 20
MOVEA(2) = -10
MOVEA(3) = 0
MOVEA(4) = 12
GO(0, 1, 2, 3, 4)
```

This requires many statements, but it is possible to use fewer statements by using the `[ ]` operator to specify multiple values for a single parameter (a compound parameter). The above example then becomes:

```
MOVEA([0, 1, 2, 3, 4]) = 10, 20, -10, 0, 12
GO(0, 1, 2, 3, 4)
```

This automatically generates a series of calls binding each RHS parameter to each LHS parameter. The code generated by the compiler is the same for both the long and short-hand versions. If, in the above example, it was required to move all the axes to the same position, then a single RHS value could have been used, with the semi-colon operator causing it be re-used the required number of times. The code would then become:

```
MOVEA([0, 1, 2, 3, 4]) = 0;
GO(0, 1, 2, 3, 4)
```

Note that if the `[ ]` operator is used to supply multiple values to a single parameter, it can only be used for the first parameter in the call. For example, the `CONTOURPARAMETER` MML call takes two parameters, so setting the stop angle of axes 0 and 2 to 10 and 20 degrees respectively, as shown below, is legal:

```
CONTOURPARAMETER([0, 2], _ctpSTOP_ANGLE) = 10, 20
```

Compare the above with attempting to set the stop angle of axis 0 to 30 degrees and the slew angle to 5 degrees, which is illegal:

```
CONTOURPARAMETER(0, [_ctpSTOP_ANGLE, _ctpSLEW_ANGLE]) = 30, 5
```

Note that this notation should only be used with a distinct parameter requiring multiple values. For calls that *expect* multiple parameters all of the same type, such as a coordinated move like `VECTORA`, this it is not required and should not be used. The following illustrates incorrect usage of the `[ ]` operator:

```
VECTORA([0, 1, 2, 3, 4]) = 10, 20, -10, 0, 12
```

The worst that could happen with the above code is that the call could be expanded to a series of calls, as was shown above for `MOVEA`. Doing this would create motion, but not coordinated motion, since it would be the equivalent of an individual call for each axis, each requiring individual triggering. This would be confusing and hard to trace, so the compiler detects this form of misuse and generates an error.

The `[ ]` operator can also be used with ‘get’ calls, and in this instance it expands it into a series of calls, the result of each being combined with the `AND` operator. Consequently, doing this only makes sense for ‘get’ calls that return 0 or 1, such as the `IDLE` function. The following example shows two ways to wait until two axes become idle:

```
Pause(IDLE([0, 1]))
Pause(IDLE(0) And IDLE(1))
```

The code generated by the compiler is the same for both of the above statements.

11.2.3 Redirection

It is possible to execute an MML call on another controller by using the redirection operator,

`->`. Redirection is possible over any bus that supports Immediate Command Mode (ICM), which currently includes CANopen and EPL (ETHERNET Powerlink). To use the redirection operator, a controller variable first needs to be created, as shown below:

```
Dim remoteController As Controller
```

The `Controller` data-type is a predefined structure that contains two members called `nBus` and `nNode`, both being integers. Once this has been created, it needs to be configured to point at a specific node on a specific bus, for example:

```
remoteController.nBus = _busETHERNET
remoteController.nNode = 5
```

A more concise way of doing this can be achieved by initializing the controller variable in its declaration:

```
Dim remoteController As Controller = {_busETHERNET, 5}
```

Once this has been done, redirected calls can be made using the `->` operator:

```
Print remoteController->POS(0)
```

The above example illustrates how to redirect to a controller connected directly to the one that was executing the Mint Basic program. It is also possible to redirect to a controller connected to the controller to which the initial redirection was made. This is done by creating another `Controller` variable that references the second controller. For example, assume that the first slave controller is connected to the master via Ethernet and is node 5, and that the second slave controller is connected to the first slave controller via CAN and is node 3. The code to perform this ‘two hop’ redirection is shown below:

```
Dim slave1 As Controller = {_busETHERNET, 5}
Dim slave2 As Controller = {_busCAN, 3}

Print slave1->slave2->POS(0)
```

A maximum of fifteen redirection hops are allowed. See the Mint help file to see if redirection is available on your product.

12 Tutorials

12.1 Introduction

The best way to learn a new programming language is by using it to write programs. This section provides a series of tutorials to show how this can be done, starting with the simplest of programs and slowly building in difficulty.

12.2 Hello world

Write a program to display some words on the screen.

One of the simplest programs that can be written is the *Hello world* program, which simply prints “Hello world” on the screen. This is done using the `Print` statement:

```
    Print "Hello world"
```

This program shows a literal string (see page 3-7) being used in a `Print` statement, and will cause this string to be displayed on the default terminal.

12.3 Variables and arithmetic

Write a program to tabulate temperatures in Celsius and Fahrenheit.

A temperature in Celsius (C) can be converted to Fahrenheit (F) using the equation F = 32 + 9 x C / 5.

The statements below will tabulate the temperatures 0, 10, 20 and 30 Celsius and their equivalents in Fahrenheit:

```
Print 0; 32 + 9 * 0 / 5
Print 10; 32 + 9 * 10 / 5
Print 20; 32 + 9 * 20 / 5
Print 30; 32 + 9 * 30 / 5
```

This technique is rather crude, since the range of values displayed cannot be changed easily, and the code involves much duplication. Both of these points make this solution error prone and inflexible. A better solution is to use a loop with a variable to calculate the temperatures. This can be achieved using the `For` statement, which allows a starting value, a final value and a step to be specified:

```
Dim celsius As Float

'Loop printing Celsius and the equivalent Fahrenheit
For celsius = 0 To 30 Step 10
  Print celsius; 32 + 9 * celsius / 5
Next celsius
```

12.4 Simple decision making and iteration

Write a program that picks a random value between 1 and 100 and allows the user to guess what it is. Report how many attempts it took to guess correctly.

The first task is to pick a random value. Mint Basic has a function called `Rnd` that returns a value greater than or equal to 0 and less than 1. Multiplying a value in this range by 100 gives a result between 0 and 99.9999. Using the `Int` function to take the integer (whole) part of this value gives a result in the range 0 to 99. Finally, adding 1 to this will give a value in the required range of 1 to 100.

```
Dim value As Integer
value = 1 + Int(Rnd * 100)
```

The next task is to read a guess from the user, which can be done using the `Input` command:

```
Dim guess As Integer
Input "Enter guess: ", guess
```

This will display the prompt `Enter guess:` on the terminal. When the user enters a value and presses enter, the variable `guess` will be set to the typed value.

The next task is to compare the guess with the required value, reporting whether it is too low, too high, or correct. This can be done using the `If` statement:

```
If guess < value Then
  Print "Higher"
ElseIf guess > value Then
  Print "Lower"
End If
```

These two steps need to be placed inside a loop that terminates when the correct guess is entered:

```
Dim value As Integer
Dim guess As Integer

value = 1 + Int(Rnd * 100)
Repeat
  Input "Enter guess: ", guess
  If guess < value Then
    Print "Higher"
  ElseIf guess > value Then
    Print "Lower"
  End If
Until guess = value
```

Note how the calculation of `value` is outside the loop, since it must not change while guesses are being made.

This program will work correctly, but it does not yet report how many attempts it took to guess the correct value. To do this, another variable is needed that increments each time a guess is entered. This variable will be printed when the loop terminates:

```
Dim value As Integer
Dim guess As Integer
Dim attempts As Integer
```

```
value = 1 + Int(Rnd * 100)
attempts = 0
Repeat
  Input "Enter guess: ", guess
  attempts = attempts + 1
  If guess < value Then
    Print "Higher"
  ElseIf guess > value Then
    Print "Lower"
  End If
Until guess = value
Print "Correct in ", attempts, " attempts"
```

Finally, the program can be made to continue indefinitely by putting most of the code inside an overall Loop statement.

```
Dim value As Integer
Dim guess As Integer
Dim attempts As Integer

Loop
  'Calculate a value in the range 1-100 and reset the attempt count
  value = 1 + Int(Rnd * 100)
  attempts = 0

  'Iterate until a correct guess is made
  Repeat
    'Read the guess and increment the number of attempts
    Input "Enter guess: ", guess
    attempts = attempts + 1

    'See if the guess was correct, prompting as required
    If guess < value Then
      Print "Higher"
    ElseIf guess > value Then
      Print "Lower"
    End If
  Until guess = value

  'Notify the user of success and the number of attempts used
  Print "Correct in ", attempts, " attempts"
End Loop
```

Since the program has become moderately complex, comments have been added to describe each stage.

12.5 Point to point moves 1

Write a program to move axis 0 repeatedly from -10 to 10 and back again.

There are two commands to move an axis to a given location, MOVEA and MOVER. These move to an absolute position (MOVEA), or a position relative to the current location (MOVER).

The first thing to do is to move to the starting location at -10:

```
MOVEA(0) = -10
```

Issuing this command alone does not make the axis move. This is because the specified move has been loaded into the *move buffer*, and is waiting to be triggered. Triggering is achieved using the GO command, so the program becomes:

```
MOVEA(0) = -10
GO(0)
```

The axis will now move to location -10.

The axis must move repeatedly backwards and forwards, so the commands need to be enclosed within a loop.

```
MOVEA(0) = -10
GO(0)
Loop
  MOVEA(0) = 10
  GO(0)
  MOVEA(0) = -10
  GO(0)
End Loop
```

The program could be simplified to merge the initial position into the back and forth motion:

```
Loop
  MOVEA(0) = -10
  GO(0)
  MOVEA(0) = 10
  GO(0)
End Loop
```

Repeatedly loading moves can cause the move buffer to become full. This is not necessarily a problem, but can be avoided by waiting for each move to complete before loading the next. This can be achieved using the IDLE function combined with the Pause command:

```
Loop
  MOVEA(0) = -10
  GO(0)
  Pause(IDLE(0))
  MOVEA(0) = 10
  GO(0)
  Pause(IDLE(0))
End Loop
```

12.6 Point to point moves 2

Write a program to move axes 0 and 1 repeatedly from (-10, -50) to (10, 50) and back again.

The program from the previous example could be used as a starting point for this problem:

```
Loop
  MOVEA(0) = -10
  MOVEA(1) = -50
  GO(0, 1)
  MOVEA(0) = 10
  MOVEA(1) = 50
  GO(0, 1)
End Loop
```

Note how each axis is triggered using `GO(0, 1)`. You will notice that the axes will get out of phase with each other (because the distance each is moving differs significantly), but this can be prevented by using the `IDLE` function combined with the `Pause` command, as before:

```
Loop
  MOVEA(0) = -10
  MOVEA(1) = -50
  GO(0, 1)
  Pause(IDLE(0) AndAlso IDLE(1))
  MOVEA(0) = 10
  MOVEA(1) = 50
  GO(0, 1)
  Pause(IDLE(0) AndAlso IDLE(1))
End Loop
```

Note how the program uses the `AndAlso` operator to wait for both axis 0 and axis 1 to be idle.

This program can be simplified by loading both axes' positions in the same command using the compound parameter. Waiting for the axes to become idle can also be simplified using the same technique:

```
Loop
  MOVEA([0, 1]) = -10, -50
  GO(0, 1)
  Pause(IDLE([0, 1])
  MOVEA([0, 1]) = 10, 50
  GO(0, 1)
  Pause(IDLE([0, 1])
End Loop
```

12.7 Point to point moves 3

Write a program to move axes 0 and 1 repeatedly from (-10, -50) to (10, 50) and back again, while ensuring that both axes arrive at each location at the same time.

There are two commands to move axes to a given location in a coordinated manner; `VECTORA` and `VECTORR`. These move to an absolute position (`VECTORA`), or a position relative to the current location (`VECTORR`).

The program from the previous example could be used as a starting point for this problem, but with the `MOVEA` commands replaced with `VECTORA`:

```
Loop
  VECTORA(0, 1) = -10, -50
  GO(0)
  VECTORA(0, 1) = 10, 50
  GO(0)
End Loop
```

Note how each axis is triggered using just `GO(0)`. This is because a coordinated move has a *master axis*, which is always the first axis specified in the move. Only the master axis needs to be specified when triggering the move or when reading whether it is idle. All the axes in a coordinated move are controlled by the master axis.

```
Loop
  VECTORA(0, 1) = -10, -50
  GO(0)
  Pause(IDLE(0))
  VECTORA(0, 1) = 10, 50
  GO(0)
  Pause(IDLE(0))
End Loop
```

13.1 Introduction

This section describes how Mint Basic differs from previous versions of Mint Basic and from Visual Basic. Also discussed are the options available to configure the language, and a keyword summary.

13.2 Porting to Mint v5.5

It is natural and unavoidable with any programming language that some changes result in compatibility issues, and this is no exception with Mint Basic. These issues are discussed below to help streamline the porting of programs developed with older versions of Mint Basic. Also discussed are the differences between Mint Basic and other popular languages.

13.2.1 From MintMT / Mint v5

The enhancements made to Mint Basic since MintMT / Mint v5 may require changes to be made to your Mint Basic programs so that they compile successfully under Mint v5.5. Many of the required changes will help avoid the warnings that are generated to help guide you into adopting the revisions to the syntax. This section outlines all of the differences and describes what must be done to make programs compile without generating warnings or errors.

The simplest approach would be to set the compiler's compatibility mode to 5000 (see *Compatibility* on page 7-3 for details), which should allow an existing program to compile without making any changes (except the repositioning of the `Startup` module to the end of the program). However, doing this will not allow any of the new language features, like structures, to be used. Alternatively, the compatibility mode can be set to 5400, which will have the same compatibility settings as 5000, but will allow the use of new language features. Note that there may be some compatibility problems when using this mode; for example the `Is` keyword will conflict with the abbreviation for `INSTATE`.

Using either of these modes will provide the most direct means of getting an existing program running quickly. However, if it is intended to port the program back to the original system, then mode 5000 should be used to avoid inadvertently using a feature that will not be supported by the older system.

13.2.1.1 Language enhancements

Mint Basic includes a number of enhancements compared with previous versions:

- 'Dot parameters' have been replaced with bracketed parameters. For example, `MOVER.0` becomes `MOVER(0)`.
- The use of square brackets `[ ]` to denote multiple parameters has changed - see *Advanced parameter passing* on page 11-2.
- Keywords that were followed by a space then an identifier or value now require brackets around the identifier(s) or value. For example, `Run myTask`, `TaskSuspend myTask` and `Wait 250` become `Run(myTask)`, `TaskSuspend(myTask)`and `Wait(250)`.
- The `Structure` keyword can be used to create data-types that contain multiple members (see *Structures* on page 5-6).
- Variables of type `Time` can be declared to overcome the scoping limitations inherent with the `Time` keyword (see page 3-8).
- The `Is` operator can be used in the `Select` statement (see page 6-5).
- The `Controller` data type can be used in conjunction with the redirection operator (`->`) to redirect MML calls (see pages 3-9 and 11-4).
- Event priorities can be programmed using `EventPriority` (see page 10-18).
- Strings can now be used in the `Select` statement (see page 6-5).
- The scope override operator (`::`) can be used to access subroutines, functions and user-defined data types (see page 8-20).
- The `Semaphore` block can be used to control shared resources (see 8-23).

- The `Bitfield` keyword can be used to create data types that simplify access to selected bits or bit-ranges within an integer value (see page 5-8).
- The intrinsic functions have been extended and some existing functions have been enhanced to allow them to do more.
- The 'immediate if' operator, `IIf`, can be used to make a choice within an expression (see page 4-5).
- The logical operators `AndAlso` and `OrElse` can be used to perform 'short-circuit' evaluation (see page 4-3).

These features can be enabled of disabled using a number of options discussed in section 7, *Directive Statements*. This can be useful, as it is possible that any of the new features mentioned above could conflict with a declaration's name in a program being ported.

13.2.1.2 The Startup module

The original language definition for Mint Basic made it illegal to place executable statements after a module declaration, but this rule was not enforced. This mistake meant that a statement like `x = 0` could be placed between two subroutine declarations, resulting in difficult to read code and the possibility that errors could go undetected (e.g. should the statement that was between two modules actually have been in one or the other?). Consequently, Mint v5.5 now enforces this rule.

Mint Workbench used to place an auto-generated `Startup` module at the head of the program, but doing this made the parent task's statements lie after a module declaration, and so this has now become illegal. If this occurs in an existing program, the error "*Error 2377 - Statement '*statement*' present after a module declaration on line* n" will be generated by the compiler. To avoid this error, simply cut out the `Startup` module and paste it at the end of the program. Mint WorkBench now correctly places an auto-generated `Startup` module at the end of the program.

13.2.1.3 Defines

The MintMT / Mint v5 compiler used the same scoping rules for `Define` statements as it did for every other declaration type, which meant that a `Define` was local to the module that it was declared in. However, in Mint v5.5, the compiler uses a pre-processor that has no concept of scope, and so a `Define` has global scope but only becomes valid from its point of declaration onwards. In general this difference should not present any compatibility issues, though it is considered bad practice to use a `Define` that was declared in another module.

For example, the following program will print '1', '2', and '1' under MintMT / Mint v5, but will print '1', '2' and '2' under Mint v5.5 (which will also generate the diagnostic "*Warning 2109 - Redefinition of Define 'myDefine' on line 8*" to alert the user that a `Define` had been redefined).

```
Define myDefine = 1

Print myDefine
mySub1
mySub2

Sub mySub1()
  Define myDefine = 2   'Redefinition
  Print myDefine
End Sub

Sub mySub2()
  Print myDefine        'Will behave differently between v5/v5.5
End Sub
```

For example, the following program will not compile under MintMT / Mint v5, but will compile under Mint v5.5:

```
mySub1
mySub2

Sub mySub1()
  Define myDefine = 2
  Print myDefine
End Sub

Sub mySub2()
  Print myDefine 'Allowed under Mint v5.5, but this use is bad practice
End Sub
```

In future releases, `Define` statements may revert to using conventional scoping rules that are more easily understood.

13.2.1.4 Labeled subroutines and events

Mint Basic continues to support labeled subroutines and events but these should ideally be converted to use the block-structured syntax (see sections 13.2.2.2 and 13.2.2.3). If these are not converted, they must be present after the main program's statements but ahead of any module declarations (e.g. subroutines, functions, tasks, etc.).

```
'Main program statements
...
End

#mySub
  ...
Return

#OnError
  ...
Return

Task myTask
  ...
End Task
```

This is because the statements that define the labeled subroutine 'mySub' and the labeled event handler 'OnError' are considered part of the main program's statements since they are not formal module declarations.

13.2.1.5 Legacy MML parameters

To allow legacy Mint Basic programs to function, the calling conventions used by previous versions of Mint Basic are also supported. The most common calling convention was the 'dot parameter', which allowed the following type of statements:

```
Print POS.0
CONTOURPARAMETER.0._ctpSTOP_ANGLE
```

This mechanism makes the job of the compiler considerably more difficult, as it has to attempt to differentiate between floating point values and integers separated by dots, which is not always an easy task. For example `CONTOURPARAMETER.1.2` needs to be resolved from `CONTOURPARAMETER, .1, .2` (or `CONTOURPARAMETER, ., 1.2`) into `CONTOURPARAMETER, ., 1, ., 2`. This adds a level of vagueness into the language definition that is not desirable, and the disambiguation rules required to resolve this make it difficult to prove correctness. Consequently, it is best to avoid dot parameters.

The next most common calling convention was the use of parameters inside square brackets, for example:

```
Print POS[0]
CONTOURPARAMETER[0]._ctpSTOP_ANGLE.
```

However, the main use of square bracket parameters was in the generation of multiple calls from a single statement, for example:

```
Pause(IDLE[0, 1])
POS[0, 1] = 0;
```

These can be converted to use round brackets: `IDLE([0, 1])` and `POS([0, 1]) = 0;`. Functions that take a variable number of parameters like `VECTORA[0, 1] = 10, 20` can be converted to `VECTORA(0, 1) = 10, 20`. Deciding whether to embed the square brackets inside round brackets, or to simply change the square brackets to round ones can cause some confusion. The rule is that if the function takes a variable number of arguments, (indicated by '`...`' in the help file topic's *Format* section, and in the hover-over help in Mint WorkBench edit windows) then the square brackets should be converted into round ones; otherwise put round brackets around the parameters (including dot parameters). So for example:

```
CONTOURPARAMETER[0]._ctpSTOP_ANGLE
```

becomes:

```
CONTOURPARAMETER(0, _ctpSTOP_ANGLE)
```

The final calling convention was to omit optional parameters, specifically the axes, bank, bus or node, the values for these being determined from a set of registers called `Axes`, `Bank` and `Bus` respectively (if the node number is omitted, the node number of the controller executing the program is assumed). Using these registers introduces the potential for error, since it is not self evident what parameters were being used in a call. There is also the potential for side effects if other parts of the program unexpectedly alter these registers.

While the Mint Basic compiler allows these constructs, it will generate a warning for each one encountered, and it is suggested that Mint Basic programs are modified to remove calls that use any of these constructs.

13.2.2 From Mint v4

Many enhancements were made to the Mint Basic language with the introduction of MintMT, but with the objective of retaining compatibility wherever possible. The changes required are described below, which once performed will put a program into a state compatible with MintMT / Mint v5 (i.e. equivalent to using compatibility mode 5000).

13.2.2.1 Configuration File

The program used to be split into two sections, the configuration and the program. This demarcation is no longer allowed and existing configuration files must be converted to a `Startup` module, which is simply achieved by copying the configurations contents and pasting them into a `Startup` module, as shown below:

```
Startup
  paste configuration here
End Startup
```

The `Startup` module should be placed with the other module declarations at the end of the program (see *Startup module* on page 8-15).

13.2.2.2 Labels

Mint v4 allowed a label to be any sequence of alphanumeric characters, but now only a valid identifier may be used. This should not normally pose any problems, as most labels were simple names, but those that are not can easily be made so as required.

13.2.2.3 Labeled Subroutines and Events

Subroutines used to be declared with a label, and terminated with a `Return` statement. They were called with the `GoSub` statement:

```
GoSub mySub
...

#mySub
  ...
Return
```

This is a crude mechanism with the only means of passing parameters being the use of global variables, and no means of hiding data local to the subroutine. While these will continue to operate, it is recommended that these are converted to use `Sub..End Sub`:

```
mySub
...

Sub mySub()
  ...
End Sub
```

Event handlers were also declared using labels and `Return`:

```
#OnError
  ...
Return
```

While this will continue to operate, it is recommended that these are converted to use:

```
Event..End Event:

Event ONERROR
  ...
End Event
```

It was also possible that labeled events could be called directly as subroutines:

```
GoSub OnError
...

#OnError
  ...
Return
```

Since it is not possible to directly call an event declared using `Event..End Event`, instances of this should be converted to a subroutine with an altered name to distinguish it from the event's name (e.g. prefixing it with 'do' would be adequate). The true event, declared using `Event..End Event`, can then call the subroutine:

```
doOnError
...

Sub doOnError()
```

```
  ...
End Sub

Event ONERROR
  doOnError
End Event
```

However, since the calling of labeled events as subroutines is not standard practice, the above construct will be rarely seen, if at all.

13.2.2.4 Scalar Array Usage

Previously, it was possible to use arrays as if they were also scalars, so `Dim a(10)` had 10 elements that were accessed by indexing, but a could also be used as if it was a scalar (without its use overwriting any of its elements). This quirk is no longer supported, and all such occurrences will now assign to the first element of the array. The only way to avoid this is to create a scalar variable and use that wherever the array is used without indexing.

13.2.3 From Visual Basic

While Mint Basic is similar to VB, it is not the same in all regards and the differences can result in some confusion for a programmer already familiar with VB. The differences that might cause problems are highlighted in this section. The two frames of reference for comparison are VB6 and VB.NET, and the term VB is used when a feature is common to both.

The `Int` function is equivalent to VB's `Fix` function, which truncates towards zero rather than -∞ (note that VB's `Int` function does the latter).

The `Select` statement may be used without the following `Case` keyword, which is not allowed in VB6, but is allowed in VB.NET.

The `Is` operator cannot be omitted from a `Case` expression as it can in VB.NET. The `Is` operator can be used in a more complex manner in Mint Basic, essentially being equivalent to the `Select` expression, allowing statements such as:

```
Case Is < 0 OrElse Is > 100
```

When printing, the meanings of the comma and semi-colon separators are reversed, and the comma is slightly different. In Mint Basic, the comma merely separates the arguments to be displayed and the semi-colon emits a tab character ahead of each argument it precedes. In VB6, the comma emits a tab ahead of each argument it precedes and the semi-colon emits a space ahead of each argument it precedes.

The Mint `Atan` function is called `Atn` in VB6.

All Mint trigonometric functions work in degrees, whereas VB uses radians.

The Mint keywords `Hex` and `Oct` are output modifiers, whereas in VB they are functions to convert a number into a string.

The Mint `IIf` function only evaluates the expression dictated by the condition, whereas VB.NET always evaluates both (which makes it almost useless). Since its initial release, VB.NET has had the poor behavior of `IIf` corrected, but to avoid any compatibility issues, it has been released as the `If` operator. There are some differences between the VB `IIf` operator and the VB.NET If operator, since the latter can take 2 or 3 arguments. The Mint Basic IIf operator should be considered as either the new VB.NET If operator with three arguments, or the existing VB IIf operator with short-circuiting.

The Mint Len function returns only the number of characters in a string, whereas in VB it can also return the number of bytes used by a variable.

The + operator is used to concatenate strings, whereas in VB both the + and the & operator can be used for this purpose. This is because in Mint the & operator is used for bitwise conjunction, but in VB the & always performs string concatenation with operands automatically cast to strings as required. The VB + operator is more complex in operation; it adds if either operand is numeric and concatenates if both operands are strings. Mint Basic will generate an error if one operand to + is numeric and the other operand is a string; operands must either be numeric or be strings, but not a combination (as allowed in VB).

The Mint Str and Val functions can take an optional second parameter indicating the base to use in the conversion. VB does not have this capability, but does have the functions Hex and Oct to convert an integer to a string.

The Auto statement is used to indicate that a program should auto-run on controller boot-up, whereas in VB.NET it specifies string marshalling rules.

The Mint STOP statement stops motion on the specified axis, whereas in VB.NET it suspends execution (like placing a break-point in the code).

The Mint Using keyword in used in an output statement (Print, Line) to specify the output format, whereas in VB.NET it specifies a block of code (terminated with End Using) that acquires a resource that is automatically freed on exiting the Using block.

In a Mint Dim statement, the default type (if none is specified) is Float, but in VB6 it is Variant and in VB.NET it is Object or the type of the initializer, if present. In Mint and VB6, each variable must be given its own type, but in VB.NET one type specification covers all preceding variable names after the previous type specification, if present, or the Dim keyword if not.

The default parameter passing mechanism in Mint is ByRef, which is the same as VB6, but VB.NET uses ByVal as default.

In Mint, the VB numeric formats for octal and hexadecimal (&O177777, &HFFFF, etc.) are not supported as they introduce ambiguity into the language, due to & being used for bitwise conjunction in Mint. Hence it would not be clear if something like i&HF represented i & HF or i &HF.

The Do looping types of VB are not supported as they introduce ambiguity into the language (due to the Loop block).

The Mint constants _false and _true have the values of 0 and -1 respectively. The equivalent VB constants False and True are generally 0 and -1 respectively, though in VB.NET, depending on how the cast is performed, True can also take the value +1.

Labels are defined by a hash character followed by an identifier, whereas in VB labels are defined by an identifier followed by a colon.

The `Shift` command is used to perform unsigned bit shifting with a negative shift indicating left, and a positive shift indicating a right shift. VB.NET uses the << and >> operators to shift bits left and right respectively and operates on both signed and unsigned values. To perform signed shifting in Mint Basic, the value should be multiplied by 2 (using `*`) to left shift, and divided by 2 (using `\`) to right shift.

Assigning a string that is longer than the destination variable’s size simply caps the string in VB, whereas in Mint Basic a ‘string overflow’ run-time error is generated.

Conversion from floating-point to integer is performed using rounding in VB, whereas Mint Basic uses truncation (apart from the case of the integer divide operator, which rounds). However, VB rounds using “Bankers’ rounding”, whereas Mint Basic rounds using simple rounding (“Round Half Up” or “Symmetric Arithmetic Rounding”).

`#Const` is not supported in Mint Basic, as the only requirement is that the expressions used for conditional compilation evaluate to a constant, whereas VB uses a more traditional pre-processor that requires explicit symbols declared with `#Const` that the pre-processor can see (i.e. it sees nothing else).

The following table gives a summary of how common areas of each language relate to each other.

Mint Basic	**VB6**	**VB.NET**
Int	Fix	Fix
-	Int	Int
Atan	Atn	Atan
Sgn	Sgn	Sign
Log10	-	Log(x, 10)
Sqrt	Sqr	Sqrt
Float	CSng	CSng
Str(x, 16)	Hex	Hex
Str(x, 8)	Oct	Oct
<, <=, >=, >, =, <>	StrComp	StrComp
Integer	Long	Integer
-	Integer	Short
Float	Single	Single
_false	False	False
_true	True	True
AndAlso	-	AndAlso
OrElse	-	OrElse
#label-name	label-name	label-name:
Shift	-	<<, >>
Rotate	-	-

13.2.4 From Structured Text

Structured Text (ST) is an IEC 61131-3 language that has a Pascal-like syntax and is the only IEC language similar to Mint Basic. This section is intended to help a programmer familiar with ST to port a program to Mint Basic or to write a program in Mint Basic.

Mint Basic uses one source file to define all the tasks and events that make up a program, while in the IEC environment, these would all be separate files. In Mint Basic, global data is declared at the outer level, while in the IEC environment, global data is specified in a VAR_GLOBAL block, which is shared amongst programs.

ST is a more strongly typed language, and so more explicit casts are required than are needed in an equivalent Mint Basic program.

Assignment uses the = symbol, but ST uses `:=` .

Strings are delimited with double quotes, but ST uses single quotes.

Non-printing characters within strings are denoted by a backslash followed by two hexadecimal digits, but ST uses a dollar character followed by two hexadecimal digits.

The `Integer` data-type is equivalent to `DINT` in ST (and `DInt` is a reserved word in Mint Basic used for disabling input events).

The `Float` data-type is equivalent to `REAL` in ST.

The `END_REPEAT` delimiter is not required after the `UNTIL` condition.

The `Select` statement is the equivalent of the CASE statement in ST.

ST does not support multi-tasking directly, though this can be achieved using multiple programs (possibly with scheduling limitations due to the IEC run-time model).

13.3 Array data files

Array data can be uploaded from a controller and stored in a file by using the Program, Data File, Upload... menu item in Mint WorkBench. Similarly, a previously uploaded file can be downloaded to a controller using the matching Download... item.

13.4 Source code reformatting

The Mint Basic compiler has the ability to reformat source code, which will correctly indent all language constructs, place spaces around operators, blank lines between modules, correctly capitalise identifier names, etc. This facility is available from Mint Workbench under the Program menu.

Since the compiler uses prescribed rules to format the code, the resulting code may differ in certain respects. For example

- While blank lines are honoured, multiple blank lines will be replaced with a single blank line (except between module declarations where multiple blank lines may be used).
- Blank lines may appear where there was no blank line in the original listing.
- Line continuations are not always honoured, as certain constructs are considered a single non-breakable unit, e.g. `Sub mySub` will not honour a line continuation placed between the `Sub` and `mySub`.

A number of listing generation options are available to refine the formatting, the details of which can be found in *Listing generation options* on page 7-10. The option *Sort declarations* allows declarations to be sorted into a prescribed order, although this may have some adverse effects. By default, the reformatted code will contain declarations in the same order as the supplied code. However, when a declaration is moved the comments immediately prior to it are moved with it, which can result in some comments moving incorrectly. For example, a block heading comment describing the program, authors, change history, etc. may move if the first declaration that follows it is a `Dim` and there are `Const`, `Define`, etc. declarations after the `Dim`.

13.5 C Format Strings

The `Using` clause may use C format strings provided `Option CFormatting` is enabled. This gives Mint Basic additional flexibility for justification, leading zeroes and formats available (signed, unsigned, hexadecimal, octal, float, etc.). Additionally, some extensions become available, such as the ability to display binary values and IP addresses.

A C format string takes the following form:

“%[sign][width][.precision][length]<format>"

Items enclosed in square brackets are optional, but the format (in angle brackets) is mandatory.

sign
This determines whether a numeric value is always preceded by its sign (whether it is positive or negative) or is only prefixed by a sign when it is negative. The ‘+’ character is used to force the sign to be displayed.

width
This specifies the minimum number of characters used, the justification (left or right) and the padding character. If the value is narrower than the field width then it will be padded, by default, with spaces, or with leading zeroes if the field width starts with a leading zero (left justification always pads with spaces). Padding will be to the right if the field width is prefixed with a ‘-‘ character (left justification) and to the left if this character is omitted (right justification).

precision
This is a value prefixed with a period, and specifies the number of decimal places for floating-point values, the number of characters to output for string values, and the number of digits to output for integer data. For integer data, the value will be padded with leading zeroes if it has fewer digits than the specified precision.

length
This indicates the size of the data being processed. The ‘l’ (lower case 'L') is used to indicate a long[7] rather than an int. For most controllers, integer is the same as long integer (i.e. 32 bits) and so does not need to be used. However, on Flex+Drive*II*, MintDrive*II*, MicroFlex *e*100 and MotiFlex *e*100 an integer is the same as a short integer (i.e. 16 bits), and so the ‘l’ modifier must be used. Note that the ‘l’ modifier may be safely used on all controllers for integer data, but is not required for floating-point data (32 bits) as Mint Basic does not allow the use of long floats (64 bits, called `double` in C).

format
This specifies the formatting that is to be applied to the value. Valid format characters are described in the following sections.

7.The terms `long`*,* `int` *and* `short` *are C programming language specific, but need to be mentioned as C format strings deal directly with the C data-types. A system’s natural word size is represented by an* `int`*, and the rules are that a* `long` *is at least as long as an* `int` *and a* `short` *is no longer than an* `int`*.*

13.5.1 b: binary notation

The integer argument is printed in unsigned binary notation.

```
Print 1234 Using("%lb")
10011010010
Print -121015514 Using("%lb")
11111000110010010111001100100110
```

13.5.2 c: character

The integer argument is printed as a single character.

```
Print -121015514 Using("%lc")
&
```

13.5.3 d: decimal

The integer argument is printed in signed decimal notation. The · characters in the following examples indicate the position of spaces in the output.

```
Print -121015514 Using("%ld")
-121015514

Print -121015514 Using("%12ld")
··-121015514

Print -121015514 Using("%-12ld")
-121015514··

Print -121015514 Using("%012ld")
-00121015514

Print -121015514 Using("%12.10ld")
·-0121015514

Print -121015514 Using("%-12.10ld")
-0121015514·
```

13.5.4 e: exponential (scientific) notation

The floating point argument is printed in decimal using scientific notation. This format will output non-numeric characters (if present) in lower case. If upper case characters are required, then the format may be converted to upper case E.

```
Print 1234.0 Using("%e")
1.234000e+03

Print 1234.0 Using("%E")
1.234000E+03
```

13.5.5 f: fixed point notation

The floating point argument is printed in decimal using fixed point notation. The output is in the format [-]mmm.nnnnn where the number of n's is specified by the precision, and the number of m's is that required to obtain the specified field width. The default precision is 6. The · characters in the following examples indicate the position of spaces in the output.

```
Print 12.34567 Using("%f")
12.345670
```

```
Print 12.34567 Using("%.2f")
12.35
```

```
Print 12.34567 Using("%12f")
···12.345670
```

```
Print 12.34567 Using("%12.2f")
·······12.35
```

```
Print 12.34567 Using("%+12.2f")
······+12.35
```

```
Print 12.34567 Using("%+-12.2f")
+12.35······
```

```
Print 12.34567 Using("%+012.2f")
+00000012.35
```

13.5.6 g: general (floating point) notation

The floating point argument is printed in decimal using either %e or %f, whichever is shorter, with trailing zeroes not being printed. This format will output non-numeric characters (if present) in lower case. If upper case characters are required, then the format may be converted to upper case G.

```
Print 1234.0 Using("%g")
1234
```

```
Print 12345678.0 Using("%g")
1.23457e+07
```

```
Print 12345678.0 Using("%G")
1.23457E+07
```

13.5.7 o: octal notation

The integer argument is printed in octal notation.

```
Print 1234 Using("%lo")
2322
```

13.5.8 q: quad (IP address) notation

The integer argument is printed in quad notation (also called dotted-quad or dot-decimal notation).

```
Print 12345678 Using("%lq")
0.188.97.78
```

```
Print -121015514 Using("%lq")
248.201.115.38
```

13.5.9 s: string

The string argument is printed. The · characters in the following examples indicate the position of spaces in the output.

```
Print "Hello world" Using("%s")
Hello world

Print "Hello world" Using("%10s")
Hello world

Print "Hello world" Using("%-10s")
Hello world

Print "Hello world" Using("%20s")
·········Hello world

Print "Hello world" Using("%-20s")
Hello world·········

Print "Hello world" Using("%20.10s")
··········Hello worl

Print "Hello world" Using("%-20.10s")
Hello worl··········

Print "Hello world" Using("%.10s")
Hello worl
```

13.5.10 u: unsigned decimal

The integer argument is printed in unsigned decimal notation.

```
Print -1234 Using("%lu")
4294966062

Print -121015514 Using("%lu")
4173951782
```

13.5.11 x: hexadecimal notation

The integer argument is printed in hexadecimal notation. This format will output non-numeric characters (if present) in lower case. If upper case characters are required, then the format may be converted to upper case X.

```
Print 1234 Using("%lx")
4d2

Print 16#4D2 Using("%lx")
4d2

Print 1234 Using("%lX")
4D2

Print -121015514 Using("%lx")
f8c97326

Print -121015514 Using("%lX")
F8C97326
```

Formats that would normally have a prefix when specified as literals in a Mint Basic program are output without any such prefix. For example, `16#ABC` would output as abc if `%lx` were used.

Note that there is no type checking performed, either during compilation or run-time, that validates that the data being printed is compatible with the format string used, and this can cause unpredictable behavior if they are incompatible. For example, displaying an integer with the formats ‘f’, ‘g’ or ‘e’ will not output what was expected, and displaying a floating-point value with the formats ‘b’, ‘c’, ‘d’, ‘q’, ‘u’ or ‘x’ will also cause unpredictable output. The format ‘s’ is only valid when used when displaying a string, and likewise will cause unpredictable behavior if any other data-type is used. For example, the following statement all use incompatible data and format strings:

```
Print 1234 Using("%f")       'Bad – Integer data/float format
Print 1234.0 Using("%d")     'Bad – Float data/integer format
Print "Hello" Using("%u")    'Bad – String data/integer format
Print 1234 Using("%s")       'Bad – Integer data/string format
```

The formats used by the Mint Basic output modifiers are as follows.

Type	**Modifier**	**C Format String**	**Description**
Integer	Dec (or none)	"%ld"	Decimal
	Hex	"%lX"	Hexadecimal
	Bin	"%lb"	Binary
Float	none	"%.4f"	Floating point
	Sci	"%e"	Scientific notation
String	none	"%s"	String

When the modifiers are used in conjunction with a `Using` clause that takes integer arguments, there are some important differences between this and the C format strings.

- The padding character is controlled by `Option ZeroPad`.
- The width value specifies the number of digits before the decimal point (excluding the sign) rather than the overall field width, except when the `Sci` modifier is used, in which case it represents to overall field width.
- A negative width value specifies that the sign should be emitted.
- The precision value is only applicable to floating-point values.

These limitations and anomalies, mainly to aid compatibility with previous versions of Mint Basic, make it generally easier to use C format strings rather than the output modifiers.

13.6 Error Codes

This section lists the errors that may be issued either during compilation or execution of a program.

13.6.1 Compilation Error Codes

A program may fail to compile for a number of reasons, such as using an undeclared identifier, supplying an incorrect number of parameters in a call, etc. The table below lists all compilation errors.

Error Code	Description	Extended Description
2100	Internal error	This should never occur and indicates an internal fault in the compiler. In such circumstances, please contact ABB with an example of the code that produced this error.
2101	Name is reserved	This should never occur and indicates that a predefined symbol shares its name with a reserved word. Please contact ABB if this occurs.
2102	Lexically incorrect	This should never occur and indicates that a predefined symbol has a name that does not conform to the naming rules for identifiers. Please contact ABB if this occurs.
2103	Anachronism	This is not an error, but merely an indication that Mint Basic now has a better feature than the one that has been used. For example, labelled subroutines called with `GoSub` and ended with `Return` should be replaced with the true subroutines now provided by Mint Basic. Also, if the newer constructs are used, they will be displayed in the Mint WorkBench tree view for easier program navigation.
2104	Too many errors	This is generated when the maximum number of errors allowed is exceeded.
2105	Symbol table read error	This should never occur and should be reported to ABB if it does.
2106	Failed to read default options	This should never occur and should be reported to ABB if it does.
2107	*Reserved*	
2108	Define redefinition	This occurs if an identifier that represents a define is used as the name of another define.
2109	Define is a function of itself	This occurs if a define is defined in terms of itself, either directly or mutually.
2150	Unterminated string	This occurs when a string literal is not terminated with a closing double quote before the end of the line is reached.
2151	Bad label	This occurs if a label is declared that uses a name that is a reserved word.

Error Code	Description	Extended Description
2152	Bad ASCII code	This occurs when an incorrect ASCII code is specified within a string literal. Valid ASCII codes are `\"` to specify a double quote, `\\` to specify a backslash, and `\hh` where hh is a two digit hexadecimal number. An invalid character such as `\£`, or an invalid ASCII code such as `\Fg`, will cause this error.
2153	Bad number	This occurs when a badly formed number is encountered, such as `1.128r-6` or `0xffgffff`.
2154	Bad identifier	This occurs if an identifier is composed entirely of underscores.
2200	Unexpected end-of-file	This occurs when end of file is reached while processing a construct that has not been terminated.
2201	Unexpected symbol	This indicates that a symbol has been encountered that is either out of context or not a recognized language element.
2202	Expected end-of-line	This occurs when the end of a line is required but is not present.
2203	Expected Then	This occurs while parsing an `If` statement that does not have a `Then` clause immediately after the conditional expression.
2204	Unexpected Else	This occurs while parsing an `If` block that has already had its `Else` clause processed, thus making further `Else` clauses illegal.
2205	Expected End If	This occurs when a block `If` statement is terminated with something other than `End If`.
2206	Expected Case	This occurs when the `Select` statement is not followed by the `Case` keyword.
2207	Expected End Select	This occurs when a `Select` statement is terminated with something other that `End Select`.
2208	Expected End While	This occurs when a `While` statement is terminated with something other than `End While`, `Endw` or `Wend`.
2209	Expected Until	This occurs when a `Repeat` statement is terminated with something other than `Until`.
2210	Expected End Loop	This occurs when a `Loop` statement is terminated with something other than `End Loop` or `Endl`.
2211	Expected identifier	This occurs when an identifier is expected and was not found, for example, tasks, events, subroutines and functions, amongst others, all require an identifier.
2212	Expected equals	This occurs when an assignment operator is expected, such as after a `For` loop variable, or after the name of a `Define`, but one is not found.
2213	Expected To	This occurs while parsing a `For` statement when there is no `To` keyword after the initial value expression.
2214	Expected Next	This occurs when a `For` statement is terminated with something other than `Next`.

Error Code	Description	Extended Description
2215	Incorrect identifier in Next	This occurs when the optional loop counter identifier is used, but which does not match that actually used.
2216	Expected expression	This occurs when an expression is required, but one is not present. For example `x = myFunc(a, b,)` would generate this error, as there is no expression after the last comma.
2217	Unexpected Return	This occurs when a `Return` statement is used anywhere other than the outer scope, i.e. inside a subroutine, task, etc.
2218	Unexpected Sub	This occurs when an attempt is made to declare a subroutine anywhere other than the outer scope or the outer scope of a task. For example, it is illegal to declare a subroutine inside any of the following: start-up module, event, subroutine, function or any block construct.
2219	Unexpected Function	This occurs when an attempt is made to declare a function anywhere other than the outer scope or the outer scope of a task. For example, it is illegal to declare a function inside any of the following: start-up module, event, subroutine, function or any block construct.
2220	Expected ‘(‘	This occurs when an opening bracket is expected, but not found. For example, subroutine and function parameter lists must be bracketed and intrinsic functions (`Sin`, `Cos`, `Atan2`, `TaskStatus`, etc.) must have their parameters bracketed.
2221	Expected ‘)’	This occurs when a closing bracket is expected, but not found. For example, subroutine and function parameter lists must be bracketed and intrinsic functions (`Sin`, `Cos`, `Atan2`, `TaskStatus`, etc.) must have their parameters bracketed.
2222	Expected ‘[‘	This occurs when the `Axes` statement is not immediately followed by `[`.
2223	Expected ‘]’	This occurs when an expression list initiated with `[` is not terminated with `]`.
2224	Expected ‘,’	This occurs when items required to be separated by a comma are not.
2225	Unexpected ‘,’	This occurs when a parameter list has a trailing comma with no expression after it.
2226	Expected End Sub	This occurs when a subroutine declaration is terminated with something other than `End Sub`.
2227	Expected data-type	This occurs when something other than integer, float or string follows an `As` clause.
2228	Expected End Function	This occurs when a function declaration is terminated with something other than `End Function`.
2229	Unexpected Task	This occurs when an attempt is made to declare a task at anywhere other than the outer scope.
2230	Expected End Task	This occurs when a task declaration is terminated with something other than `End Task`.

Error Code	Description	Extended Description
2231	Unexpected Event	This occurs when an attempt is made to declare an event at anywhere other than the outer scope.
2232	Expected End Event	This occurs when an event declaration is terminated with something other than `End Event`.
2233	Unexpected Startup	This occurs when an attempt is made to declare a start-up module at anywhere other than the outer scope.
2234	Expected End Startup	This occurs when a start-up module declaration is terminated with something other than `End Startup`.
2235	Non-reference array	This occurs when an array parameter is specified as being passed by value using the `ByVal` keyword. This is illegal, as arrays are always passed by reference.
2236	Cannot initialise parameters	Parameters receive their initial values from the call statement, and so it makes no sense, and is illegal, to try and initialize them.
2237	Else without If	This occurs when an `Else` clause is encountered outside an `If` statement.
2238	End If without If	This occurs when an `End If` clause is encountered outside an `If` statement.
2239	Case without Select	This occurs when a `Case` clause is encountered outside a `Select` statement.
2240	End Select without Select	This occurs when an `End Select` clause is encountered outside a `Select` statement.
2241	End While without While	This occurs when an `End While` clause is encountered outside a `While` statement.
2242	Until without Repeat	This occurs when an `Until` clause is encountered outside a `Repeat` statement.
2243	End Loop without Loop	This occurs when an `End Loop` clause is encountered outside a `Loop` statement.
2244	Next without For	This occurs when a `Next` clause is encountered outside a `For` statement.
2245	End Sub without Sub	This occurs when an `End Sub` clause is encountered outside a subroutine declaration.
2246	End Function without Function	This occurs when an `End Function` clause is encountered outside a function declaration.
2247	End Task without Task	This occurs when an `End Task` clause is encountered outside a task declaration.
2248	End Event without Event	This occurs when an `End Event` clause is encountered outside an `Event` declaration.
2249	End Startup without Startup	This occurs when an `End Startup` clause is encountered outside a start-up module declaration.

Error Code	Description	Extended Description
2250	Block not found	This occurs when an unqualified `Exit` or `Continue` statement is used without being in a block that can be exited or continued.
2251	Sub block not found	This occurs when an `Exit` statement qualified with `Sub` is used without being in a subroutine.
2252	Function block not found	This occurs when an `Exit` statement qualified with `Function` is used without being in a function.
2253	Task block not found	This occurs when an `Exit` statement qualified with `Task` is used without being in a task.
2254	Event block not found	This occurs when an `Exit` statement qualified with `Event` is used without being in an event.
2255	Startup block not found	This occurs when an `Exit` statement qualified with `Startup` is used without being in the start-up module.
2256	While block not found	This occurs when an `Exit` or `Continue` statement qualified with `While` is used without being in a `While` statement.
2257	Repeat block not found	This occurs when an `Exit` or `Continue` statement qualified with `Repeat` is used without being in a `Repeat` statement.
2258	For block not found	This occurs when an `Exit` or `Continue` statement qualified with `For` is used without being in a `For` statement.
2259	Loop block not found	This occurs when an `Exit` or `Continue` statement qualified with `Loop` is used without being in a `Loop` statement.
2260	Unexpected end-of-statement	This occurs when a statement ends unexpectedly, for example `Input` followed by no input variable.
2261	Expected statement separator	This occurs when a statement separator is expected but not found, for example: `While Rnd() < 0.5 Print "Mint"`
2262	Size string in Dim only	This occurs when an attempt is made to size a string outside a `Dim` statement, for example: `Function abc(s As String * 12)`
2263	Expected End Critical	This occurs when a critical block is terminated with something other than `End Critical`.
2264	*Reserved*	
2265	Size strings only	This occurs when a data-type other than a string is sized, for example: `Dim f As Float * 4`
2266	Expected End Structure	This occurs when a structure is terminated with something other than `End Structure`.
2267	Expected '}'	This occurs when a braced initialization has something other than a close brace after a semi-colon.
2268	Expected End DriveMacro	This occurs when a drive macro is terminated with something other than `End DriveMacro`.

Error Code	Description	Extended Description
2269	End DriveMacro without DriveMacro	This occurs when an `End DriveMacro` statement is encountered outside a drive macro statement.
2270	Expected End Shutdown	This occurs when a shutdown module is terminated with something other than `End Shutdown`.
2271	End Shutdown without Shutdown	This occurs when an `End Shutdown` is encountered outside a shutdown statement.
2272	Expected End Semaphore	This occurs when a semaphore is terminated with something other than `End Semaphore`.
2273	End Semaphore without Semaphore	This occurs when an `End Semaphore` is encountered outside a semaphore statement.
2274	Expected End Bitfield	This occurs when a bitfield is terminated with something other than `End Bitfield`.
2300	Invalid event name	This occurs when an event name is used that does not correspond to a valid event is used. Either the event specified is completely illegal, or the controller does not support that event.
2301	Multiple declaration	This occurs when an identifier is used twice in the same scope for declaring an object such as a variable, subroutine, task, etc.
2302	Event preclusion	This occurs when events that are mutually exclusive are declared. For example, if the general `FASTIN` event has been declared, then it is illegal to declare any of the specifically numbered events such as `FASTIN1` (and vice-versa), as this would cause ambiguity about which to call.
2303	Unused declaration	This occurs when an object is declared but never used. This is not an error, but indicates that the object in question could safely be removed. Note that if this warning is issued for an event, then the event mentioned has not been included in a call to `EventPriority` so will be ignored.
2304	Identifier not found	This occurs when an attempt is made to use something that has not been declared, such as a variable or subroutine.
2305	Identifier shared with predefined	This occurs when a user defined object is given the same name as a predefined symbol, such as an MML routine, event name or constant.
2306	Cannot call tasks or events	This occurs when an attempt is made to call a task or event as if were a subroutine or function.
2307	Expected value	This is generated whenever a value is required, but one is not present. For example, the string literal `"Value"` is not a value, but the literal `1.0` and variable `x` are values.
2308	Modify constant	This occurs when a constant is supplied as a parameter to a subroutine or function that modifies the parameters value.

Error Code	Description	Extended Description
2309	Expected variable	This occurs whenever a variable is expected but something else is supplied, such as a constant or literal.
2310	Expected constant	This occurs whenever a constant is expected but something else is supplied, such as a variable.
2311	Expected variable or value	This occurs when a parameter is supplied to a subroutine or function call that is neither a variable nor a value, such as a task or event name.
2312	Expected Null	This occurs when an MML routine is called that accepts an array parameter that may be null, but is supplied with a constant value other than zero (null).
2313	Bad parameter in call	This occurs when a reference parameter is modified, but is supplied with something that cannot be modified. Examples of this are a variable that is not the same type as the parameter, an MML call or an expression.
2314	Bad cast	This occurs when an attempt is made to cast an object to a type that is incompatible. An example of this would be the implicit casting of a string to an integer during an assignment, for example: `i = "Hello world"` where `i` is an integer variable.
2315	Incorrect number of parameters	This occurs when a call is made with the wrong number of parameters, and can occur with subroutines/functions and MML routines.
2316	Incorrect number of indices	This occurs when an array is indexed with the wrong number of indices. For array parameters, where the number of indices and index ranges are not explicitly specified, the first usage determines the number of indices.
2317	Cannot index scalar	This occurs when an attempt is made to index a scalar.
2318	Wrong call class	This occurs when a callable routine is used in the wrong context. Examples of this are functions called as subroutines, subroutines called as functions, MML command routines used as get/set routines and read only MML routines being written to.
2319	Call of non-callable object	This occurs when an attempt is made to call an object that is not callable, such as an event or a task.
2320	Declaration hides other	This occurs when a declaration hides a declaration in an outer scope that shares the same name.
2321	Expected Task	This occurs when a task name is expected, but is not found. The `TaskSuspend`, `TaskPriority` and `TaskStatus` keywords all require qualifying with a task name, and the `Run` and `End` commands may be qualified with a task name.

Error Code	Description	Extended Description
2322	Integer out of range	This occurs when the compiler evaluates a constant integer expression, and the result lies outside the range of a 32-bit signed integer.
2323	Float out of range	This occurs when the compiler evaluates a constant floating-point expression, and the result falls outside the range of a 4-byte IEEE float.
2324	Character out of range	This occurs when the value supplied to the `Chr` function lies outside the range 0 to 255.
2325	Division by zero	This occurs when the compiler evaluates a constant expression containing a division by zero.
2326	Using clause ignored	This is displayed when the `Using` clause is used with character and string data. *Note that this error is obsolete. It will not appear in more recent versions of Mint WorkBench that support string variables.*
2327	Hex or Bin used with bad decimals	The `Hex` and `Bin` print modifiers imply integer output, and so specifying a number of decimal places other than zero makes no sense.
2328	Expected array	This occurs when an MML call that requires an array parameter is not supplied with an array.
2329	Expected float array	This occurs when an MML call that requires an array of floats is supplied with an array of another type.
2330	Expected integer array	This occurs when an MML call that requires an array of integers is supplied with an array of another type.
2331	Too many right-hand-sides	This occurs when the number of right-hand side values supplied in a statement exceeds the number of left-hand sides. This error often occurs when using square bracket notation.
2332	Illegal jump into block	This occurs when a label inside the body of a `For` loop is referenced from outside the loop. This is not allowed because the initialization stage of the `For` loop will be bypassed leading to undefined behavior.
2333	Too many tasks	This occurs on controllers that support only a limited number of tasks when more than the maximum number of tasks allowed are declared. Some controllers, such as the Flex+Drive*II*, support only a single task.
2334	Unexpected ‘;’	This occurs when an attempt is made to use a semi-colon with a MML routine that does not require it. The semi-colon is only appropriate for use with axis related commands that either use an axis string (or have the axes specified in square brackets), or non axis related commands that make use of square bracket notation.

Error Code	Description	Extended Description
2335	Function return not assigned	This occurs when a function is declared, but the statements in the function's body do not include an assignment to the function's name to specify the return value. The process used for checking this error does not confirm that all paths through the function are valid.
2336	Expected static module	This occurs when a module name used with the scope override operator `::` is not the name of a task, event or start-up module.
2337	Expected string	This occurs when a string is expected in an assignment to a string array, but something else is supplied.

2338	String constant too long	This occurs when the string assigned to a string variable is longer than the variable. For example:
		`Dim s As String * 10`
		`s = "0123456789abcd"`
		The string assigned to `s` is longer than 10 characters so the error occurs. Similarly, the error can occur if a string array is initialized with a string that is too long. The default length for string variables is 64 characters. See *Strings* on page 3-7.

2339	Loss of precision	This warning occurs when a value is assigned to a destination of a different type. Depending on the value being assigned, this can lead to a significant change in the value, such as when assigning a float to an integer where any fractional data will be lost, or when an integer is assigned to a float where some of the lower digits may be lost. To check for this, if an integer was assigned to a float and the float assigned back to another integer, then would the two integer values match? This check is performed on assignments, calls to subroutines/ functions that take parameters and constant declarations where the type is explicitly specified. For example, the following program will generate two warnings about the assignments of `i` to `x` and `x` to `j`, and will display `123456789` and `123456784`, highlighting the problem:
		`Dim i As Integer`
		`Dim j As Integer`
		`Dim x As Float`
		`i = 123456789 : x = i : j = x`
		`Print i; j`

Error Code	Description	Extended Description
2340	Temporary used in call	This warns that a value passed to a subroutine will be stored in a temporary (invisible) variable so that the variable's address can be used as the reference. This is particularly important for strings, as using a temporary variable to contain an intermediate string result has the problem that the temporary variable may be too small, causing run-time string overflows. However, this warning is generated for any type of temporary variable that may have to be created. To avoid this warning, either ensure that the value is held in an appropriate variable before it is passed to the subroutine/function or, if the parameter is not intended to return a value, make it a value parameter by prefixing it with the `ByVal` keyword.

2341	Define used before declaration	This error is generated when a define name is used before it has been defined (see *Define* on page 5-2). To avoid this error, ensure that `Define` appears before the name is used, for example:
		`Define my_axes = 0, 1, 2, 3`
		`Print my_axes`

2342	Anachronistic MML call	This warning is generated when the specified 'old keyword' has been superseded by a differently named 'new keyword'. It is likely that both keywords operate in exactly the same way, so it will be possible to use the old keyword name without problems. However, it is better to use the new keyword name to maintain maximum compatibility with any future developments.

2343	Expected integer	This error is generated by the `Shift` and `Rotate` commands (when used as a statement) if the item to be shifted is not an integer, for example:
		`Dim x As Float`
		`Shift(x, 2)`

2344	Bad Using clause	This error is generated if bad parameters are used in a `Using` clause, such as a string. For example: `Print "Pos = ", POS(0) Using("6, 3")`
2345	Unrecognised option key	This will be generated if an option name is used that is not recognized, for example: `Option BillyGoatGruff 1`
2346	Unsupported option key	This will be generated if an option is set that is unsupported by a certain target format, for example: `Option CFormatting 1` on target formats below 10.
2347	Bad option value	This will be generated if an option is assigned a value that is invalid/out of range, for example: `Option OptLevel 6`
2348	Option multiply defined	This will be displayed if the same option is set more than once in a program.

Error Code	Description	Extended Description
2349	Option not in outer level	This will be displayed if an option is set inside a module of any type (task, subroutine, function, event, start-up or shutdown).
2350	String out of range	This will be displayed when the `Asc` function is supplied with an empty string.
2351	Label out of scope	This will be displayed if a `GoTo` or `GoSub` specifies a label that is not situated in the same scope as the `GoTo` or `GoSub` statement.
2352	Parameter must be reference	This occurs when a parameter that must be passed by reference is specified as being `ByVal`.
2353	Unexpected size	This occurs when a size specification is used with a parameter declaration or with data types that cannot be sized. Only strings can be sized.
2354	Incompatible operands	This occurs when an operator is given operands that are incompatible, such as when trying to divide a float by a string.
2355	Must be scalar	This occurs when an item that must be scalar has been given dimensions, such as a constant or a bitfield member.
2356	Bad module nesting	This occurs when modules are incorrectly nested, such as would occur if an event were declared inside a subroutine.
2357	ElseIf after Else	This occurs when an `ElseIf` statement is encountered after an `Else` statement.
2358	Multiple Else	This occurs when more than one `Else` statement is present.
2359	Case after Case Else	This occurs when a `Case` statement is encountered after a `Case Else` statement.
2360	Multiple Case Else	This occurs when more than one `Case Else` statement is present.
2361	Expected Case	This occurs when a `Case` statement was expected, but something else was encountered.
2362	Expected scalar	This occurs when a scalar (a single value) is expected, but an aggregate such as an array or structure is encountered.
2363	Expected 1D array	This occurs when a one-dimensional array is expected, but a multi-dimensional array is encountered.
2364	Expected array of Null	This occurs when an array or Null is expected, but something else is encountered.
2365	Result indeterminate	This occurs when a result cannot be determined from the given operands/parameters.
2366	Illegal initialisation	This occurs when an initialization is encountered that is not allowed, such as with parameters or structure members.
2367	Expected float	This occurs when a float is expected, but something else is encountered.
2368	Variable used but not initialised	This occurs when a variable has been used, but has not been assigned a value.

Error Code	Description	Extended Description
2369	Expected assignment	This occurs when as assignment is expected, but something else is encountered.
2370	Incorrect identifier in Next	This occurs when the expression used with `Next` does not match that used in the `For`.
2371	Anachronism	This occurs when an out of date language feature is used. For example: ■ The use of `GoSub` or `Return`. ■ The use of labelled events, for example `#OnError`, `#Timer` etc. ■ When redundant decimals have been ignored. For example, in the statement `Input i Using(6, 0)`, if `i` is an integer then the zero (which specifies the fractional places to display) is redundant. ■ The use of obsolete parameter formats, such as `POS.0` or `POS[0]`. See *From MintMT / Mint v5* on page 13-2. ■ The use of `Axes`, `Bank`, `Bus` or `Terminal` to set a default value. Each item should now be explicitly defined with each keyword. ■ Omitting an optional parameter to an MML function, like `POS = 0`. Each parameter should now be explicitly defined, e.g. `POS(axis) = 0`. ■ The use of an obsolete keyword, such as `Adc0`, `In0`, `Out0`, `ik`, `rk`, etc. See *Keyword support options* on page 7-4.
2372	Expected string array	This occurs when a string array is expected, but something else is encountered.
2373	Block invalid	This occurs when an invalid block type is encountered, such as when trying to continue a subroutine.
2374	Unexpected output modifier	This occurs when an output modifier is unexpectedly encountered, such as when one is used in an `Input` statement.
2375	Unexpected Using clause	This occurs when a `Using` clause is encountered when it was not expected, such as with the prompt string of an `Input` statement.
2376	Bad Is operator	This occurs when an `Is` operator is used outside a `Case` expression.

Error Code	Description	Extended Description
2377	Statement after module	This occurs when executable statements are present after any module declaration. When using compiler target versions 11 or above (required for firmware versions 5400 and above), all executable statements that are not contained within a module (subroutine, function, task, event, start-up or shutdown) must appear before the modules, as viewed in the Mint WorkBench editing window. A common cause of this error is when the start-up module has been placed at the very beginning of the program, before statements that are not contained within a module. The start-up module should be placed at the end of the program code.
2378	Not in outer level	This occurs when something is incorrectly nested within a module, such as an `Option` statement or a task declaration.
2379	Bad input parameter	This occurs when an incorrect `Input` parameter is encountered, such as when there are no parameters or just a prompt string.
2380	Too few right-hand-sides	This occurs when not enough parameters are supplied to the right of an assignment.
2381	Incompatible type	This occurs when an incompatible type is encountered, such as using something other than a floating-point or integer `For` loop counter, or when assigning structures of different types.
2382	Expected structure	This occurs when a structure is expected, but something else is encountered.
2383	Expected member	This occurs when a member is expected, but something else is encountered, which can occur with the structure member access operator and the scope override operator.
2384	Float equality	This occurs when a test for equality is made when either or both operands are floating-point.
2385	Expected redirect	This occurs when a redirection is expected, but something else is encountered.
2386	Expected redirect of MML API	This occurs when a redirection or a Mint Basic function is expected, but something else is encountered.
2387	Too many redirections	This occurs when too many redirections are used (the current maximum allowed is 15).
2388	Expected Case statement	This occurs when a `Case` statement is expected, but something else is encountered.
2389	Expected type name	This occurs when a type name is expected, but something else is encountered.
2390	Expected time	This occurs when a variable of type time is expected, but something else is encountered.
2391	Expected time array	This occurs when an array of type time is expected, but something else is encountered.

Error Code	Description	Extended Description
2392	Recursive structure	This occurs when a structure contains a member that is of the same type as the structure being declared, which can also be through mutually recursive structures.
2393	Expected ‘{‘	This occurs when a `{` is expected, but something else is encountered.
2394	Unexpected ‘{‘	This occurs when a `{` is encountered, but something else is expected.
2395	Too few elements	This occurs when there are fewer initializers than structure members.
2396	Too many elements	This occurs when there are more initializers than structure members.
2397	Expected variable or MML API	This occurs when a variable or a Mint Basic function is expected, but something else is encountered.
2398	Multiple EventPriority	This occurs when the `EventPriority` command is used more than once in a program.
2399	Incorrect event enumeration	This occurs when an event enumeration is used that is not recognized in a call to `EventPriority`.
2400	Expected indexing	This occurs when an array indexing operation is expected, but is not present.
2401	Block not found	This occurs when an unqualified `Exit` or `Continue` statement is used without being in a block that can be exited or continued.
2402	Expected statement	This error results when a statement is expected, but something else is found. For example, `Rnd + 1` is not a statement, and would generate this error if used as one.
2403	Drive macro data overflow	This error is issued when a drive macro contains more than 1006 bytes of data.
2404	Too many drive macro’s	This error is issued when a program contains more than 5 drive macro declarations.
2405	Drive macro name too long	This error is issued when a drive macro name exceeds 16 characters.
2406	Case value already used	This warning is issued when a case value is used more than once, including overlapping ranges or ranges that include any previously used values.
2407	Expected simple type	This error is issued when a function return type is not a simple type. It must be an intrinsic type, like Float, and must not be an array.
2408	Expected semaphore	This error is issued when the expression in a semaphore block is not of type Semaphore.
2409	Expected bitfield	This error is issued when a bitfield is expected but something else is found, e.g. `myBitfieldVar = 1`.

Error Code	Description	Extended Description
2410	Expected bitrange	This error is issued when a bit range is expected but something else is encountered, for example declaring a bitfield member a using a `As Float`.
2411	Result determinate	This warning is issued when an expression's result is determinate, even though it is not composed entirely of literal values. For example, the expression `(x > 10) = 2` will always be false because `x > 10` can only take the values 0 or 1, so can never be equal to 2.
2412	Expected label	This error is issued when a `GoTo` statement uses something other than a label for its target, e.g. `GoTo _false`.
2413	Statement ignored	This occurs when a statement is encountered that has no effect and so is ignored, for example running a task from the start-up or shutdown modules.
2414	Bad Defined parameter	This occurs when the parameter to the `Defined` function is not a simple identifier.
2415	Defined function not in #If	This error is issued when the `Defined` function is used anywhere other than in a `#If` or `#ElseIf` expression.
2416	Bad #If expression	This indicates an expression that is invalid, such as one that is non-numeric, one that uses a variable or function call, one that uses a constant ahead of its declaration, or one that uses an unevaluated constant. For example: `Const _n = _m, _m = 12` will result in _n being unevaluated because it is a function of a constant declared after itself.
2417	Message	This indicates a user generated error or warning from using the keywords `#Error` or `#Warning`, the text of which being arbitrary.

13.6.2 Run-Time Error Codes

Execution of a program may result in an error condition, such as when evaluating the square root of a negative value, which will generate an invalid argument error. The table below lists all the errors that are generated by the MVM, whether they are fatal and what the outcome is after an error.

Error Code	**Description**	**Fatal**	**Outcome**
3100	Division by zero	No	Numerator
3101	Invalid argument	No	Argument
3102	Stack overflow	Yes	-
3103	Index out of range	Yes	-
3104	Integer out of range	No	_minInt or _maxInt
3105	Bank out of range	No	-
3106	Bus out of range	No	-
3107	Axis out of range	No	-
3108	Stack underflow	Yes	-
3109	String overflow	No	String unaltered
3110	Error registers not primed	No	Random value
3111	Evaluation error	No	0.0, 0 or “”
3112	Out of memory	Yes	-

The MML has its own set of error codes, which are all treated as non-fatal.

MML Error Range	**Description**
0 to 499	Synchronous errors.
500 to 799	Asynchronous errors.
800 to 999	H2 errors.

As mentioned in *Run-time Errors* on page 3-17, an error causes one of two things to occur; either program termination or the calling of the error handler. Fatal errors always cause immediate termination (preceded by execution of the shutdown module, if present), as will non-fatal errors when no error handler is present, otherwise the error handler (`ONERROR`) is invoked.

13.7 Reserved words

Reserved words follow the naming conventions for identifiers, but their use is reserved by Mint Basic for use in specifying program structure, intrinsic commands, operators, etc. Note that some reserved words are compound, i.e. made up of multiple components, such as `End While`, which is not allowed for user identifiers.

13.7.1 Constants

The following table lists all the names reserved for use as constants in Mint Basic:

Name	Description
`_false`	0
`_FlexPlusDrive2`	15
`_maxInt`	2147483647
`_minInt`	-2147483648
`_MicroFlexE100`	27
`_MintDrive2`	14
`_MotiFlexE100`	32
`_MotiFlexE100MintCard`	35
`_NextMoveBX`	2
`_NextMoveBX2`	17
`_NextMoveES`	20
`_NextMoveESB`	23
`_NextMoveESB2`	36
`_NextMoveE100`	31
`_NextMovePCI`	9
`_NextMovePCI2`	29
`_NextMoveST`	19
`_Null`	0
`_pi`	3.1415927
`_platform`	The platform code of the controller/drive (e.g. 9 for NextMove PCI, 14 for MintDrive^{II}, etc.)
`_true`	1
`_tskRunning`	1
`_tskSuspended`	2
`_tskTerminated`	0
`_ver`	The version string of the compiler (e.g. "Version 13.0")
`_VirtualController`	12

13.7.2 Operators

The following table lists all the names reserved for use as operators in Mint Basic:

Name	**Description**
`And`	Bitwise conjunction
`AndAlso`	Logical conjunction
`Bool`	Logical affirmation
`IIf`	Immediate if
`Is`	Access `Select` expression
`Mod`	Modulus
`Not`	Logical negation
`Or`	Bitwise inclusive disjunction
`OrElse`	Logical inclusive disjunction
`Xor`	Bitwise exclusive disjunction

13.7.3 Intrinsic commands

The following table lists all the names reserved for use as commands in Mint Basic:

Name	Description
`Bank`	Sets the default bank when assigned to.
`Bus`	Sets the default bus when assigned to.
`Dint`	Disables digital input events.
`Echo`	Sets the input-echo mode for the specified terminal.
`EInt`	Enables digital input events.
`End`	Ends execution of the current program or specified task(s).
`EventPriority`	Allows the priority of events to be specified.
`GoTo`	Makes execution continue at the location of the specified label.
`IPend`	Sets the pending state of the digital input events for the specified bank when assigned to.
`Mid`	Sets the string from the specified location, optionally capping the number of characters changed to the specified amount.
`Nop`	No operation (consumes one clock cycle).
`Pause`	Pauses until the specified condition is met.
`Rotate`	Rotates the content of the given variable by the specified number of bits.
`Run`	Runs the program or the specified task(s).
`Shift`	Shifts the contents of the given variable by the specified number of bits.
`TaskPriority`	Sets the priority of the specified task.
`TaskQuantum`	Sets the quantum size of the specified task.
`TaskResume`	Resumes execution of the specified task.
`TaskSuspend`	Suspends execution of the specified task.
`Terminal`	Sets the default terminal bitmap when assigned to.
`Time`	Sets the time in milliseconds when assigned to.
`Troff`	Turns trace on.
`Tron`	Turns trace off.
`Wait`	Wait for the specified number of milliseconds.

13.7.4 Intrinsic functions

The following table lists all the names reserved for use as functions in Mint Basic:

Name	Description
`Abs`	Absolute value.
`Acos`	Arc-cosine of the numeric argument, result in degrees.
`Asc`	ASCII code of the first character of the string argument.
`Asin`	Arc-sine of the numeric argument, result in degrees.
`Atan`	Arc-tangent of the numeric argument, result in degrees.
`Atan2`	Arc-tangent of the two numeric arguments in the correct quadrant, result in degrees.
`Axes`	Default axis bitmap.
`Bank`	Default bank.
`Bus`	Default bus.
`Chr`	Cast integer to a single character string.
`Cos`	Cosine of the numeric argument in degrees.
`CvtFlt2Ieee`	Converts a native float bit-pattern to an IEEE float bit-pattern.
`CvtFlt2Int`	Converts a native float bit-pattern to an integer.
`CvtIeee2Flt`	Converts an IEEE float bit-pattern to a native float.
`CvtInt2Flt`	Converts an integer bit-pattern to a native float.
`DprEventCode`	The code of the pended DPR event.
`Echo`	The input echo state for the given terminal parameter.
`Erl`	The line number of the last error encountered (non *e*100 only).
`Err`	The error code of the last error (non *e*100 only).
`ErrAxis`	The axis number of the last error (non *e*100 only).
`ErrStr`	The error string of the last error (non *e*100 only).
`Eval`	The evaluation of the string argument.
`Exp`	The exponential of the numeric argument.
`Float`	The floating-point value of the integer argument.
`Frac`	The fractional component of the floating-point argument.
`InKey`	Reads the next character from the input buffer for the given terminal parameter.
`InStr`	The character location of the location of one string in another.
`Int`	The integer part of the integer argument.
`IsAlnum`	Is the character argument alphanumeric.
`IsAlpha`	Is the character argument alphabetical.
`IsAscii`	Is the character argument an ASCII code.
`IsCntrl`	Is the character argument a control code.
`IsDigit`	Is the character argument a decimal digit.

Name	Description
`IsLower`	Is the character argument lowercase.
`IsUpper`	Is the character argument uppercase.
`IsXDigit`	Is the character argument a hexadecimal digit.
`IPend`	Bitmap of any pending digital input events of the specified bank.
`LastKey`	The last key read using either `InKey` or `ReadKey`.
`LBound`	Lower bound of the specified array.
`Left`	The left portion of the string argument.
`Len`	The number of characters in the specified string.
`Log`	The natural logarithm of the numeric parameter.
`Log10`	The common (base 10) logarithm of the numeric parameter.
`Max`	The largest of the arguments.
`Mid`	The middle portion of the string argument.
`Min`	The smallest of the arguments.
`Pow`	Raises the first numeric argument to the power of the second numeric argument.
`ReadKey`	The key currently depressed on a CAN keypad node.
`Right`	The right portion of a string argument.
`Rnd`	A random number in the range $0 \leq x < 1$.
`Rotate`	The numeric argument rotated the specified number of bits.
`Round`	The nearest integer to the floating-point argument, or if a number of decimal places is specified (as a second argument), returns the floating-point argument rounded to that number of decimal places.
`Sgn`	The sign of the numeric argument (-1, 0, +1).
`Shift`	The numeric argument shifted the specified number of bits.
`Sin`	Sine of the numeric parameter in degrees.
`Sqrt`	Square root of the numeric argument.
`Str`	The string representation of the numeric argument.
`Tan`	Tangent of the numeric argument in degrees.
`TaskStatus`	Status of the specified task.
`Terminal`	Default terminal bitmap.
`Time`	Current time in milliseconds.
`UBound`	Upper bound of the specified array.
`Val`	Numeric representation of the string argument.

13.7.5 Block constructs

The following table lists all of the names reserved for use as language elements in Mint Basic:

Name	Description

`Continue`
`Continue For`
`Continue Loop`
`Continue Repeat`
`Continue While`
	Force continuation of the closest surrounding loop.
	Force continuation of the closest surrounding `For` loop.
	Force continuation of the closest surrounding `Loop` loop.
	Force continuation of the closest surrounding `Repeat` loop.
	Force continuation of the closest surrounding `While` loop.

`Critical`
`End Critical`
	Mark the start of a critical section of code.
	Mark the end of a critical section of code.

`Event`
`End Event`
	Declare an event.
	Mark the end of an event declaration.

`Exit`
`Exit For`
`Exit Loop`
`Exit Repeat`
`Exit Select`
`Exit While`
	Force exiting of the closest surrounding loop.
	Force exiting of the closest surrounding `For` loop.
	Force exiting of the closest surrounding `Loop` loop.
	Force exiting of the closest surrounding `Repeat` loop.
	Force exiting of the closest surrounding `Select` statement.
	Force exiting of the closest surrounding `While` loop.

`Exit Event`
`Exit Function`
`Exit Shutdown`
`Exit Startup`
`Exit Task`
`Exit Sub`
	Exit the event.
	Exit the function.
	Exit the Shutdown module.
	Exit the Startup module.
	Exit the task.
	Exit the subroutine.

`For`
`To`
`Step`
`Next`
	Define a `For` loop.
	Specify the initial and final values of the loop.
	Specify the step used on each iteration of the loop.
	Marks the end of the loop.

`If`
`Then`
`ElseIf`
`End If`
	Define an `If` statement.
	Marks the end of the condition.
	Follow on condition to test if the prior condition was not met.
	Marks the end of the `If` statement.

`Loop`
`End Loop`
	Define an unconditional loop.
	Marks the end of the loop.

`Repeat` `Until`	Define a ‘repeat until condition true’ loop. Marks the end of the loop.

`Select Case`
`Case`
`Case Else`
`End Select`
	Define a select statement.
	Define a value or range of values.
	Define what to do if all prior conditions not met.
	Marks the end of the select statement.

`Semaphore`
`End Semaphore`
	Mark the start of a semaphore block.
	Mark the end of a semaphore block.

`Startup`
`End Startup`
	Declare a `Startup` module.
	Marks the end of the `Startup` module.

Name	Description

`While`
`End While`
	Define a ‘while condition true’ loop.
	Marks the end of the loop.

13.7.6 Data types

The following table lists all the names reserved for use as data-types.

Name	Description
`Controller`	Specifies that the type is `Controller`.
`Float`	Specifies that the type is `Float`.
`Integer`	Specifies that the type is `Integer`.
`Semaphore`	Specifies that the type is `Semaphore`.
`String`	Specifies that the type is `String`.
`Time`	Specifies that the type is `Time`.

13.7.7 Simple declaration

The following table lists all the names reserved for use as language elements in Mint Basic:

Name	Description
`As`	Used to specify the type of a variable or parameter.
`ByRef`	Used to specify that a parameter be passed by reference.
`ByVal`	Used to specify that a parameter be passed by value.
`Const`	Used to declare a named constant.
`Define`	Used to declare a define.
`Dim`	Used to declare a variable.

13.7.8 Block declaration

The following table lists all the block constructs reserved for the declaration of data-types and modules.

Name	**Description**

`Bitfield`
`End Bitfield`
	Declare a bitfield.
	Mark the end of a bitfield.

`Event`
`End Event`
	Declare an event.
	Mark the end of an event declaration.

`Function`
`End Function`
	Declare a function.
	Marks the end of the function.

`Shutdown`
`End Shutdown`
	Declare a Shutdown module.
	Marks the end of the Shutdown module.

`Sub`
`End Sub`
	Declare a subroutine.
	Marks the end of the subroutine.

`Startup`
`End Startup`
	Declare a Startup module.
	Marks the end of the Startup module.

`Structure`
`End Structure`
	Declare a structure.
	Marks the end of the structure.

`Task`
`End Task`
	Declare a task.
	Marks the end of the task.

13.7.9 Mint Motion Library functions

The following keywords are for use in accessing MML API. Please note that not all the functions listed below will be available on all controllers, and there are likely to be new functions available that are not listed below. Consult the latest help file for a current list of the MML functions available on each controller.

Name	Description
ABORT	To abort motion on all axes.
ABORTMODE	To control the default action taken in the event of an abort.
ABSENCODER	To read the current resolver, EnDat or Hiperface encoder position.
ABSENCODERMODE	To compensate for abnormal Hiperface encoder wiring.
ABSENCODEROFFSET	To set the zero point for an EnDat or Hiperface encoder.
ABSENCODERTURNS	To set or read the number of turns of unique information available on an absolute encoder.
ACCEL	To define the acceleration rate of an axis.
ACCELDEMAND	To read the instantaneous demand acceleration.
ACCELJERK	To define the jerk rate to be used during periods of acceleration.
ACCELJERKTIME	To define the jerk rate to be used during periods of acceleration.
ACCELSCALEFACTOR	To scale axis encoder counts, or steps, into user defined acceleration units.
ACCELSCALEUNITS	To define a text description for the acceleration scale factor.
ACCELTIME	To define the acceleration rate of an axis.
ACCELTIMEMAX	To define the acceleration rate of an axis.
ACTIVERS485NODE	To enable the transmitter on a controller's RS485 port.
ADC	To read an analog input value.
ADCDEADBAND	To set the deadband to be applied to an ADC input.
ADCDEADBANDHYSTERE-SIS	To set a hysteresis level for entering and leaving the deadband on the ADC inputs.
ADCDEADBANDOFFSET	To set the deadband offset to be applied to an ADC input.
ADCERROR	To read back the analog inputs currently in error.
ADCERRORMODE	To control the default action taken in the event of an ADC limit being exceeded on an associated channel.
ADCFILTER	To set the amount of filtering to be applied to the specified analog input.
ADCGAIN	To set the gain to be applied to an ADC input.
ADCMAX	To set the upper analog limit value for the specified analog input.
ADCMIN	To set the lower analog limit value for the specified analog input.

Name	Description
ADCMODE	To set the analog input mode.
ADCMONITOR	To specify the analog inputs that an axis will monitor for analog limit checking.
ADCOFFSET	To set the offset to be applied to an ADC input.
ADCOFFSETTRIM	To zero (trim) the specified analog input.
ADCTIMECONSTANT	To set the time constant of the low pass filter applied to an ADC input.
ASYNCERRORPRESENT	To determine whether an asynchronous error is present.
AUTOHOMEMODE	To set the autohome mode for the specified configuration.
AUTOSTARTMODE	To set the autostart mode for the specified configuration.
AUXDAC	To set or read the auxiliary DAC outputs.
AUXDACOFFSET	To apply a voltage offset to an auxiliary DAC.
AUXENCODER	To set or read the auxiliary encoder input.
AUXENCODERMODE	To make miscellaneous changes to the auxiliary encoders.
AUXENCODERPRESCALE	To scale down the auxiliary encoder input.
AUXENCODERROLLOVER	To count the number of wraps of the auxiliary encoder value.
AUXENCODERSCALE	To set or read the scale factor for the auxiliary encoder input.
AUXENCODERSPEED	To specify a (virtual) speed reference for the auxiliary encoder.
AUXENCODERVEL	To read the velocity of the auxiliary encoder input.
AUXENCODERWRAP	To set or read the encoder wrap range for the auxiliary encoder input.
AUXENCODERZERO-ENABLE	To re-enable Z pulse capturing on the auxiliary encoder
AUXENCODERZERO-LATCHMODE	To control the latching mechanism for the auxiliary encoder's Z latch.
AUXENCODERZERO-POSITION	To read the auxiliary encoder position at the last Z capture.
AUXENCODERZLATCH	To read the state of the auxiliary encoder's Z latch.
AXISBUS	To read the fieldbus used to host this axis.
AXISCHANNEL	To allow user mapping of hardware to axis numbers.
AXISDAC	To read the DAC channel used to control the specified axis.
AXISERROR	To read back the motion error.
AXISMODE	To return the current mode of motion.
AXISNODE	To read the node number used to host the axis.
AXISPDOOUTPUT	To read the stepper pulse/direction output channel used to control the specified axis.
AXISPOSENCODER	To select the source of the position signal used in dual encoder feedback systems.

Name	Description
AXISREMOTECHANNEL	To read the remote channel number on the node used to host the axis.
AXISSTATUS	To return the current error status from the specified axis.
AXISSTATUSWORD	To read the DS 402 status word for an axis.
AXISVELENCODER	To select the source of the velocity signal used in dual encoder feedback systems.
AXISWARNING	To read or clear present axis warnings.
AXISWARNINGDISABLE	To allow individual axis warnings to be enabled and disabled.
BACKLASH	To set the size of the backlash present on an axis.
BACKLASHINTERVAL	To set the rate at which backlash compensation is applied.
BACKLASHMODE	To control the use of backlash compensation.
BLEND	To start blending the current move with the next move in the buffer.
BLENDDISTANCE	To specify the distance, before the end of the vector path, where blending will begin.
BLENDMODE	To enable blending for interpolated moves.
BOOST	To control the stepper boost outputs.
BRIDGECOMPENABLE	To enable or disable bridge circuit compensation.
BRIDGEERRORCURRENT	To set the current parameter used when compensating for non-linearities in the drive's PWM bridge.
BRIDGEERRORVOLTAGE	To set the voltage parameter used when compensating for non-linearities in the drive's PWM bridge.
BUSBAUD	To specify the bus baud rate.
BUSCOMMANDMASK	To define a bit mask for CANopen, DeviceNet and PROFIBUS Command telegrams.
BUSENABLE	To enable or disable the operation of a fieldbus.
BUSEVENT	To return the next event in the bus event queue of a specific bus.
BUSEVENTINFO	To return the additional information associated with a bus event.
BUSNODE	To set or read the node ID used by this node for the specified bus.
BUSPROCESSDATAIN	To configure the drive for the type of process data that will be received from the master.
BUSPROCESSDATAIN-DATATYPE	To configure the data type for process data that will be received from the master.
BUSPROCESSDATAIN-PARAMETER	To define the associated parameter for items received in process data telegrams.
BUSPROCESSDATAOUT	To configure the type of process data that will be sent by the drive.

Name	Description
BUSPROCESSDATAOUT-DATATYPE	To configure the data type for process data that will be sent by the drive.
BUSPROCESSDATAOUT-INTERVAL	To define the update interval for information sent in process data telegrams.
BUSPROCESSDATAOUT-PARAMETER	To define the associated parameter for items sent in process data telegrams.
BUSPROTOCOL	To read the protocol currently supported on a particular fieldbus.
BUSRESET	To reset the bus controller.
BUSSTATE	To return the status of the bus controller.
BUSTIMEOUT	To alter the inter-character timeout for MODBUS ASCII.
CAM	To perform a cam profile.
CAMAMPLITUDE	To modify the amplitude of a cam profile.
CAMBOX	To start or stop a CAMBox channel.
CAMBOXDATA	To load data associated with a CAMBox channel.
CAMEND	To define an end point in the cam table if multiple cams are required.
CAMINDEX	To returns the currently executing cam segment number.
CAMPHASE	To allow a cam profile to be shifted forwards or backwards over a fixed number of cam segments.
CAMPHASESTATUS	To get the state of the CAMPHASE for a specific axis.
CAMSEGMENT	To change CAM table data.
CAMSTART	To define a start point in the cam table if multiple cams are required.
CAMTABLE	To specify the array names to be used in a cam profile on the specified axis.
CANCEL	To stop motion and clear errors on an axis.
CANCELALL	To stop motion and clear errors on all axes.
CAPTURE	To control the operation of capture.
CAPTUREAXIS	To set or read the axis for a capture channel.
CAPTUREBUFFERSIZE	To read the total size of the capture buffer.
CAPTURECHANNEL-INTEGERUPLOAD	To allow an entire channel of captured data values to be uploaded as integer data into an array.
CAPTURECHANNEL-UPLOAD	To allow an entire channel of captured data values to be uploaded into an array.
CAPTURECOMMAND	To control the operation of capture.
CAPTUREDURATION	To define the total duration of the data capture.
CAPTUREEVENT	To configure capturing to stop on an event.
CAPTUREEVENTAXIS	To set the axis to monitor for the capture trigger event.

Name	Description
CAPTUREEVENTDELAY	To define the post-trigger delay for event capture.
CAPTUREHSMODE	To set or read the mode of a high speed capture channel.
CAPTUREINTERVAL	To define the interval between data captures, relative to the servo frequency.
CAPTUREMODE	To set or read the mode on a capture channel.
CAPTUREMODE-PARAMETER	To specify a parameter associated with CAPTUREMODE.
CAPTURENUMPOINTS	To read the number of captured points per channel.
CAPTUREPERIOD	To define the interval between data captures.
CAPTUREPOINT	To allow individual capture values to be read.
CAPTUREPOINTINTEGER	To allow individual capture values to be read as integer values.
CAPTUREPRETRIGGER-DURATION	To set the duration of the pre-trigger phase.
CAPTUREPROGRESS	To return the progress of the pre-trigger or post-trigger capture phase.
CAPTURESTATUS	To return the progress of the capture.
CAPTURETRIGGER	To generate a capture trigger.
CAPTURETRIGGER-ABSOLUTE	To ignore the sign of the trigger value when triggering from a capture channel source.
CAPTURETRIGGER-CHANNEL	To set the channel to be used as the reference source for triggering.
CAPTURETRIGGERMODE	To set the method used to evaluate the trigger source.
CAPTURETRIGGER-SOURCE	To set the reference source to be used for triggering.
CAPTURETRIGGERVALUE	To set the trigger value when triggering from a capture channel source.
CHANNELTYPE	To determine what hardware is available to a specific channel.
CIRCLEA	To perform a circular move with absolute co-ordinates.
CIRCLER	To perform a circular move with relative co-ordinates.
CLEARERRORLOG	To clear the error log.
COMMISSIONED	To set or read whether the axis/drive has been commissioned.
COMMS	To access the reserved comms array.
COMMSINTEGER	To access the reserved comms array, storing values as integers.
COMMSMAPDATATYPE	To define the data type of a comms element.
COMMSMAPMODE	To set or read the comms mapping for a comms element.
COMMSMAPPARAMETER	To set or read the associated parameter for a mapped comms element.
COMMSMODE	To select comms use over either RS485 or CANopen.

Name	Description
COMMSRETRIES	To set the maximum number of re-tries for a RS485/422 comms telegram.
COMPAREENABLE	To enable/disable the position compare control of a specific digital output.
COMPARELATCH	To read the state of the position compare latch.
COMPAREMODE	To enable and disable the position compare on an axis.
COMPAREOUTPUT	To specify the digital output used for position compare.
COMPAREPOS	To write to the position compare registers.
CONFIG	To set the configuration of an axis for different control types.
CONNECT	To enable a connection between two remote nodes to be made or broken.
CONNECTSTATUS	To return the status of the connection between this node and another node.
CONTOURANGLE	To set the inter-vector angle threshold for contoured moves.
CONTOURMODE	To enable contouring for interpolated moves.
CONTOURPARAMETER	To set the parameters for contoured moves.
CONTROLMODE	To set or read the control mode.
CONTROLMODESTARTUP	To set or read the control mode used when the drive is turned on.
CONTROLRATE	To set the control loop and profiler sampling rates.
CONTROLREFCHANNEL	To specify a channel for the source of the control reference command.
CONTROLREFSOURCE	To specify the source of the control reference command.
CONTROLREFSOURCE-STARTUP	To set or read the source of the control reference command used when the drive is turned on.
CURRENTDEMAND	To read the demands to the current controllers.
CURRENTLIMIT	To restrict the current output to a defined range.
CURRENTMEAS	To read the measured current.
CURRENTSENSORMODE	To enable a current sensor temperature drift compensation scheme.
DAC	To write a value to the DAC or read the present DAC value.
DACLIMITMAX	To restrict the DAC output voltage to a defined range.
DACMODE	To control the use of the DAC.
DACMONITORABSOLUTE	To specify whether only absolute (positive) values should be output when monitoring using an auxiliary DAC channel.
DACMONITORAXIS	To specify which axis to monitor during DAC monitoring.
DACMONITORGAIN	To specify a multiplying factor for use during DAC monitoring.
DACMONITORMODE	To specify which axis parameter to monitor during DAC monitoring.

Name	Description
DACMONITORMODE-PARAMETER	To specify a parameter associated with DACMONITORMODE.
DACMONITOROFFSET	To specify an offset to add to the output when monitoring using an auxiliary DAC channel.
DACOFFSET	To apply a voltage offset to a DAC channel.
DACRAMP	To specify the number of milliseconds over which the maximum DAC output will be ramped to zero.
DECEL	To set the deceleration rate on the axis.
DECELJERK	To define the jerk rate to be used during periods of deceleration.
DECELJERKTIME	To define the jerk rate to be used during periods of deceleration.
DECELTIME	To set the deceleration rate on the axis.
DECELTIMEMAX	To define the deceleration rate of an axis.
DEFAULT	To return axis motion variables to their power-up state.
DEFAULTALL	To return all axis motion variables to their power-up state.
DPREVENT	To interrupt the host PC and generate a trappable event, using the Dual Port RAM (DPR).
DPRFLOAT	To read and write a 32-bit floating-point value to Dual Port RAM (DPR)
DPRLONG	To read and write a 32-bit integer value to Dual Port RAM (DPR).
DRIVEBUSNOMINAL-VOLTS	To return the nominal value of the DC bus voltage for the drive.
DRIVEBUSOVERVOLTS	To set or return the overvoltage trip level for the drive.
DRIVEBUSUNDERVOLTS	To set or return the undervoltage trip level for the drive.
DRIVEBUSVOLTS	To return the current level of the DC bus.
DRIVEDISABLEMODE	To prevent moves in the move buffer being cleared when an axis is disabled.
DRIVEENABLE	To enable or disable the drive for the specified axis.
DRIVEENABLEINPUT-MODE	To control the action taken in the event of the drive being disabled from the drive enable input/enable DIP switch.
DRIVEENABLEMODE	To set the drive to auto-enable on power on.
DRIVEENABLEOUTPUT	To specify an output as a drive enable.
DRIVEENABLESWITCH	To read the state of the drive enable input.
DRIVEERROR	To report errors on the drive or to clear current drive errors.
DRIVEFEEDBACK	To read the type of feedback module.
DRIVEID	To define a text description for the drive.
DRIVEOKOUTPUT	To assign a digital output as the Drive OK output.
DRIVEOVERLOADAREA	To read the extent of a drive overload condition.

Name	Description
DRIVEOVERLOADMODE	To set or read the action taken in the event of a drive overload condition.
DRIVEPEAKCURRENT	To read the peak current rating of the drive.
DRIVEPEAKDURATION	To read the duration for which peak drive current can be sustained.
DRIVERATEDCURRENT	To read the continuous current rating for the drive.
DRIVERATINGZONE	To specify the rating conditions under which the drive operates.
DRIVESPEEDFATAL	To define the overspeed trip level.
DRIVESPEEDMAX	To set or read the maximum motor speed to be used.
EFFORT	To read the instantaneous effort applied by the current controllers.
ENABLESWITCH	To read the state of the Drive Enable DIP switch.
ENCODER	To set or read the axis encoder value.
ENCODERCYCLESIZE	To set or read the size of a sin/cos cycle on an encoder.
ENCODERLINESIN	To set or read the number of encoder lines (pre-quadrature) for the drive feedback.
ENCODERLINESIN-SPEEDMAX	To read the maximum allowable speed when using a resolver feedback device.
ENCODERLINESOUT	To define the resolution of the encoder output.
ENCODERMODE	To make miscellaneous changes to the encoders.
ENCODEROFFSET	To set or read the offset used to calculate encoder position for absolute encoders.
ENCODEROUTCHANNEL	To set or read the encoder channel to be output on a simulated encoder output.
ENCODEROUT-RESOLUTION	To set or read the resolution of a simulated encoder output.
ENCODERPRESCALE	To scale down the encoder input.
ENCODERRESOLUTION	To set or read the number of encoder lines (pre-quadrature) for the motor.
ENCODERSCALE	To set or read the scale factor for the encoder channel.
ENCODERTYPE	To set or read the feedback type of the motor.
ENCODERVEL	To read the velocity from an encoder channel.
ENCODERWRAP	To set or read the encoder wrap range for the encoder channel.
ENCODERZACTIVELEVEL	To specify the active level of the encoder Z pulse.
ENCODERZLATCH	To get and reset the state of an axis' encoder Z latch.
ERRCODE	To return the last error code read from the error list.
ERRDATA	To return data associated with the last error read from the error list.
ERRLINE	To return the line number of the last error read from the error list.

Name	Description
ERRORCLEAR	To clear all errors in the specified group.
ERRORCODEENABLE	To allow or prevent specific errors to be created.
ERRORDECEL	To set the deceleration rate on the axis for powered stops, in the event of an error or stop input.
ERRORINPUT	To set or return the digital input to be used as the error input for the specified axis.
ERRORINPUTMODE	To control the default action taken in the event of an external error input.
ERRORLOGCLEAR	To clear the error log.
ERRORLOGMODE	To specify how the error log is updated.
ERRORLOGSAVE	To save the error log to non-volatile EEPROM memory.
ERRORMASK	To prevent specific error conditions calling the ONERROR event.
ERRORPRESENT	To determine if errors in a particular group are present in the error list.
ERRORREADCODE	To determine if a particular error is present in the error list.
ERRORREADNEXT	To return the next entry in the specified group from the error list.
ERRORSWITCH	To return the state of the error input.
ERRSTRING	To return the error string for the last error code read from the error list.
ERRTIME	To return the time stamp for the last error code read from the error list.
EVENTACTIVE	To indicate whether an event is currently active.
EVENTDISABLE	To selectively enable and disable Mint events.
EVENTPEND	To manually cause an event to occur.
EVENTPENDING	To indicate whether an event is currently pending.
EVENTUNPEND	To manually remove a pending event.
FACTORYDEFAULTS	To reset parameter table entries to their default values.
FASTAUXENABLE	To manually clear the auxiliary encoder's fast position latch.
FASTAUXENCODER	To return the instantaneous auxiliary encoder value that was recorded on the fast interrupt.
FASTAUXLATCH	To read the auxiliary encoder fast interrupt latch.
FASTAUXLATCH-DISTANCE	To specify the distance over which further auxiliary encoder latch edges will be ignored.
FASTAUXLATCHEDGE	To select the capture edge for fast capture on the auxiliary encoder.
FASTAUXLATCHMODE	To set the default action to be taken to clear the auxiliary encoder's fast position latch.
FASTAUXSELECT	To select which of the fast position capture inputs will capture an auxiliary encoder channel.

Name	Description
FASTENABLE	To manually clear the encoder's fast position latch.
FASTENCODER	To return the instantaneous encoder value that was recorded on the fast interrupt.
FASTLATCH	To read the axis fast interrupt latch.
FASTLATCHDISTANCE	To specify the distance over which further position latch edges will be ignored.
FASTLATCHEDGE	To define which edge polarity should cause the fast position to be captured.
FASTLATCHMODE	To set the default action to be taken to clear the encoder's fast position latch.
FASTPOS	To return the instantaneous axis position that was recorded on the fast interrupt.
FASTSELECT	To select which of the fast position capture inputs (or outputs) will cause axis position to be captured.
FASTSOURCE	To select whether fast position capture is triggered by a digital input or a digital output.
FEEDBACKFAULTENABLE	To enable or disable detection of motor feedback faults.
FEEDRATE	To set the slew speed of an individual move loaded in the move buffer.
FEEDRATEMODE	To control the use of slew speed, acceleration, deceleration and feedrate override.
FEEDRATEOVERRIDE	To override the current speed or feedrate.
FEEDRATEPARAMETER	To set the parameters for the current speed or feedrate being used.
FIRMWARERELEASE	To read the release number of the firmware.
FIRMWAREVERSION	To read the version number of the firmware.
FLY	To create a flying shear by following a master axis with controlled acceleration and deceleration.
FOLERROR	To return the instantaneous following error value.
FOLERRORFATAL	To set the maximum permissible following error before an error is generated.
FOLERRORMODE	To determine the action taken on the axis in the event of a following error.
FOLERRORWARNING	To set the following error threshold before an axis warning is generated.
FOLLOW	To enable encoder following with a specified gear ratio.
FOLLOWDENOM	To set or read the follow ratio's denominator.
FOLLOWMODE	To define the mode of operation of the FOLLOW keyword.
FOLLOWNUMERATOR	To set or read the follow ratio's numerator.
FREQ	To set a constant frequency output.

Name	Description
GEARING	To set the percentage size for gearing compensation.
GEARINGMODE	To turn gearing compensation on or off.
GLOBALERROROUTPUT	To specify a global error output which will be deactivated in the event of an error.
GO	To begin synchronized motion.
GROUP	To set or read whether a node is a member of a group.
GROUPCOMMS	To write to the comms arrays of all the nodes within a specified group.
GROUPMASTER	To set a node as the master of a group or to return the node ID of the group master.
GROUPMASTERSTATUS	To determine whether the current node is master of the group.
GROUPSTATUS	To determine whether the current node is a member of the group.
HALL	To read the current Hall state on feedback devices which use Hall sensors.
HALLFORWARDANGLE	To define the electrical angles at which Hall states change, when the motor is running in the forward direction, for feedback devices which use Hall sensors.
HALLREVERSEANGLE	To define the electrical angles at which Hall states change, when the motor is running in the reverse direction, for feedback devices which use Hall sensors.
HALLTABLE	To define the Hall table for an encoder motor.
HELIXA	To load a helix move into the move buffer.
HELIXR	To load a helix move into the move buffer.
HOLDSWITCH	To read the current state of the Hold DIP switch.
HOME	To find the home position on an axis.
HOMEACCEL	To set the acceleration rate for the homing profile.
HOMEBACKOFF	To set the home back-off speed factor.
HOMECREEPSPEED	To set the creep speed for homing moves.
HOMEDECEL	To set the deceleration rate for the homing profile.
HOMEINPUT	To set a digital input to be the home switch input for the specified axis.
HOMEOFFSET	To apply an offset to the homing sequence.
HOMEPHASE	To find the phase of the homing sequence currently in progress.
HOMEPOS	To read the axis position at the completion of the homing sequence.
HOMEREFPOS	To define a reference position for homing moves.
HOMESPEED	To set the speed for the initial seek phase of the homing sequence.

Name	Description
HOMESTATUS	To set or read the status of a homing sequence.
HOMESWITCH	To return the state of the home input.
HOMETYPE	To set the homing mode to be performed at start up.
HTA	To start the hold to analog mode of motion.
HTACHANNEL	To specify the analog input to use for a particular axis while in Hold To Analog (HTA) mode.
HTADAMPING	To specify the damping term used in the Hold To Analog (HTA) algorithm.
HTADEADBAND	To specify the analog error deadband.
HTAFILTER	To set the filter factor for the analog input.
HTAKINT	To specify the integral gain term used in the Hold To Analog (HTA) force loop.
HTAKPROP	To specify the proportional gain term used in the Hold To Analog (HTA) force loop.
IDLE	To indicate if a move has finished executing and the axis has finished moving.
IDLEMODE	To control the checks performed when determining if an axis idle.
IDLEPOS	To read or set the idle following error limit.
IDLESETTLINGTIME	To read the time taken for an axis to become idle.
IDLETIME	To specify the period for which the axis must meet its idle conditions before becoming idle.
IDLEVEL	To read or set the idle velocity limit.
IMASK	To mask off Mint events IN0 .. INx
IN	To read the state of all the inputs on an input bank.
INCA	To set up an incremental move to an absolute position.
INCR	To set up an incremental move to a relative position.
INITERROR	To report any errors detected during start up.
INITWARNING	To return the sum of a bit pattern describing initialization warnings generated at start up.
INPUTACTIVELEVEL	To set the active level on the digital inputs.
INPUTDEBOUNCE	To set or return the number of samples used to 'debounce' a digital input bank.
INPUTMODE	To set or return the sum of a bit pattern describing which of the user digital inputs should be edge or level triggered.
INPUTNEGTRIGGER	To set or return the user inputs that become active on negative edges.
INPUTPOSTRIGGER	To set or return the user inputs that become active on positive edges.
INSTATE	To read the state of all digital inputs.

Name	Description
INSTATEX	To read the state of an individual digital input.
INX	To read the state of an individual digital input.
JOG	To set an axis for speed control.
JOGCOMMAND	To start or stop a jog by giving a direction command.
JOGDURATION	To specify the duration of a timed jog.
JOGMODE	To specify the control mode for profiling a jog move.
JOGSPEED	To define a preset jog speed.
JOGTIME	To return the remaining jog time before deceleration.
KACCEL	To set the servo loop acceleration feed forward gain.
KDERIV	To set the servo loop derivative gain on the servo axes.
KEYS	To remap the layout of the keys on a BaldorCAN KeypadNode.
KFINT	To set or read the integral gain of the flux controller for induction motor control.
KFPROP	To set or read the proportional gain of the flux controller for induction motor control.
KIINT	To set the integral gain used by the current controller.
KINT	To set the servo loop integral gain.
KINTLIMIT	To restrict the overall effect of the integral gain KINT.
KINTMODE	To control when integral action will be applied in the servo loop.
KIPROP	To set the proportional gain used by the current controller.
KITRACK	To set the tracking factor used by the current controller.
KNIFE	To load a tangential knife move on the specified axis.
KNIFEAXIS	To specify the master axis that the knife axis should follow.
KNIFEMODE	To specify the knife mode with which moves on the knife master axis are loaded.
KNIFESTATUS	To read or set the status of the knife axis.
KPROP	To set the proportional gain for the position controller.
KVDERIV	To set the derivative gain used by the speed controller.
KVDERIVTCONST	To set the time constant used by the filter on the derivative gain term of the speed controller.
KVEL	To set the servo loop velocity feedback gain term.
KVELFF	To set the velocity feedforward term for the position controller.
KVINT	To set the integral gain used by the speed controller.
KVPROP	To set the proportional gain used by the speed controller.
KVTIME	To set the time constant of a low pass filter, applied to measured speed.
KVTRACK	To set the tracking factor used by the speed controller.
LATCH	To read the state of a fast latch channel.

Name	Description
LATCHENABLE	To manually re-enable a fast latch channel.
LATCHINHIBITTIME	To specify a period during which further fast triggers will be ignored.
LATCHINHIBITVALUE	To specify a range of values within which further fast triggers will be ignored.
LATCHMODE	To set the default action to be taken to clear a fast latch.
LATCHSOURCE	To define the source of data to be latched by a fast latch channel.
LATCHSOURCECHANNEL	To define the channel of the source of data to be latched by a fast latch channel.
LATCHTRIGGERCHANNEL	To select which of the fast latch inputs (or outputs) will trigger a fast latch channel.
LATCHTRIGGEREDGE	To define which edge polarity should cause the fast latch to be triggered.
LATCHTRIGGERMODE	To select whether a fast latch is triggered by a digital input or a digital output.
LATCHVALUE	To return the instantaneous latch value that was recorded by a fast latch.
LED	To set or read the display mode for the seven segment display.
LEDDISPLAY	To set or read the value for the seven segment display.
LIFETIME	To return a lifetime counter for the drive.
LIMIT	To return the state of the forward and reverse limit switch inputs for the given axis.
LIMITFORWARD	To return the state of the forward limit switch input for the given axis.
LIMITFORWARDINPUT	To set the user digital input configured to be the forward end of travel limit switch input for the specified axis.
LIMITMODE	To control the default action taken in the event of a forward or reverse hardware limit switch input becoming active.
LIMITREVERSE	To return the state of the reverse limit switch input for the given axis.
LIMITREVERSEINPUT	To set the user digital input configured to be the reverse end of travel limit switch input for the specified axis.
LOADDAMPING	To define the equivalent viscous damping coefficient for the motor and load.
LOADINERTIA	To define the combined inertia of the motor and load.
LOOPTIME	To set the servo loop update interval in microseconds.
MASTERCHANNEL	To set or read the channel of the input device used for gearing.
MASTERDISTANCE	To set the distance on the master axis over which the slave will travel for a 'segment' in master-slave move types.
MASTERSOURCE	To set or read the source of the input device used for gearing.

Name	**Description**
MAXSPEED	To set a limit for the speed demanded on an axis.
MISCERROR	To read or clear the miscellaneous error flag.
MISCERRORDISABLE	To enable or disable miscellaneous errors calling the error event.
MOTORBRAKE	To manually override motor brake control.
MOTORBRAKEDELAY	To specify engage/disengage delays associated with motor brake control.
MOTORBRAKEMODE	To activate or deactivate motor brake control.
MOTORBRAKEOUTPUT	To specify an output to be used as a control signal for a braked motor.
MOTORBRAKESTATUS	To determine the state of the motor brake control.
MOTORCATALOGNUMBER	To return the catalog number of the motor.
MOTORDIRECTION	To set or read the electrical direction of the motor.
MOTORENCODERLINES	To set or read the number of encoder lines (pre-quadrature) for the motor.
MOTORFEEDBACK	To set or read the feedback type of the motor.
MOTORFEEDBACKANGLE	Reads the instantaneous value of commutation angle for the motor.
MOTORFEEDBACKOFFSET	To set or read the electrical angle at which the absolute position read from an EnDat, Hiperface or SSI encoder is zero.
MOTORFEEDBACK-PROTOCOLERROR	To read the type of feedback error when using a Hiperface encoder.
MOTORFEEDBACK-PROTOCOLRETRIES	To set or read the number of retries to attempt when an error occurs on a Hiperface encoder.
MOTORFEEDBACKSTATUS	To read the current status of the EnDat or Hiperface encoder.
MOTORFLUX	To set the motor's magnetic flux level, to allow the drive to accurately calculate motor torque and compensate for back-EMF.
MOTORLINEARENCODER-RESOLUTION	To set the resolution of the encoder on a linear motor.
MOTORLINEARPOLE-PITCH	To set or read the distance between north poles on a linear motor.
MOTORLS	To set or read the motor leakage inductance.
MOTORMAGCURRENT	To set or read the magnetizing current (I_m) of an induction motor.
MOTORMAGIND	To set or read the magnetizing inductance (L_m) of an induction motor.
MOTOROVERLOADAREA	To read the extent of an overload condition.
MOTOROVERLOADMODE	To set or read the action taken in the event of a motor overload condition.

Name	Description
MOTORPEAKCURRENT	To set or read the peak current rating of the motor.
MOTORPEAKDURATION	To set or read the duration for which peak motor current can be sustained.
MOTORPOLES	To set or read the number of motor poles.
MOTORPOWERMEASURED	To read the instantaneous electrical power applied to the motor.
MOTORRATEDCURRENT	To set or read the rated current of the motor.
MOTORRATEDFREQ	To set or read the rated frequency of an induction motor.
MOTORRATEDSPEEDRPM	To set or read the rated speed of an induction motor.
MOTORRATEDVOLTS	To set or read the rated voltage of an induction motor.
MOTORRESOLVEROFFSET	To set the feedback alignment for a resolver motor.
MOTORROTORLEAKAGE-IND	To set or read the rotor leakage inductance of an induction motor.
MOTORROTORRES	To set or read the rotor resistance of an induction motor.
MOTORRS	To set the motor stator resistance.
MOTORSLIP	To read the slip of an induction motor.
MOTORSPECNUMBER	To return the spec number of the motor.
MOTORSTATORLEAKAGE-IND	To set or read the stator leakage inductance of an induction motor.
MOTORSTATORRES	To set or read the stator resistance of an induction motor.
MOTORTEMPERATURE-INPUT	To assign a digital input as the motor overtemperature trip input.
MOTORTEMPERATURE-MODE	To set or read the action taken in the event of the motor temperature trip input becoming active.
MOTORTEMPERATURE-SWITCH	To read the state of the motor overtemperature trip input.
MOTORTYPE	To read or set the type of motor.
MOVEA	To set up a positional move to an absolute position.
MOVEBUFFERBACKUP	To save or restore an axis move buffer.
MOVEBUFFERFREE	To return the number of free spaces in the move buffer for the specified axis.
MOVEBUFFERID	To attach or read back a 16-bit identifier from the move buffer.
MOVEBUFFERIDLAST	To read a 16-bit identifier from the move buffer.
MOVEBUFFERLOW	To set or return the number of free spaces in the move buffer before a move buffer low event is generated.
MOVEBUFFERSIZE	To set or return the size of the move buffer allocated on the specified axis.
MOVEBUFFERSTATUS	
MOVEDWELL	To load a dwell move into the move buffer.

Name	Description
MOVEOUT	To load a digital output bit pattern into the move buffer.
MOVEOUTX	To load a change of state for a specific digital output into the move buffer.
MOVEPULSEOUTX	To load a pulsed change of state for a specific digital output into the move buffer.
MOVER	To set up a positional move to a relative position.
MOVESTATUS	To return information about the progress of the current move.
NETFLOAT	To access a controller's network data array, storing values in floating-point format.
NETINTEGER	To access a controller's network data array, storing values as integers.
NODE	To set or read the node ID used by this node.
NODELIVE	To determine if a CAN node on the bus is currently live or dead.
NODESCAN	To scan a specific CAN bus for the presence of a specific node.
NODETYPE	To add or remove a CAN node to/from the CAN network. Can also be read to determine the node type.
NUMBEROF	To return information about the abilities of the controller.
NUMBEROFEXTENDED	To return information about the abilities of the controller.
NVFLOAT	To read or write a floating-point value in non-volatile memory.
NVLONG	To read or write a long integer value in non-volatile memory.
NVRAMDEFAULT	To clear the contents of non-volatile RAM (NVRAM).
OFFSET	To perform a positional offset move.
OFFSETDISTANCE	To specify the distance over which offset moves of mode 4 will occur.
OFFSETMODE	To define the mode of operation for the OFFSET keyword.
OFFSETSPEEDLIMIT	To set the maximum speed limit of an axis during an offset move.
OFFSETSTATUS	To read the status of the previous offset move.
OUT	To set or read the state of all the outputs on an output bank.
OUTPUTACTIVELEVEL	To set the active level on the digital outputs.
OUTX	To set or read an individual digital output.
PARAMETERSAVE	To save drive parameters to non-volatile memory.
PARAMSAVEMODE	To allow parameters to be stored in EEPROM during run-time.
PHASESEARCHBACKOFF	To select the back-off distance used to clear an end stop during the phase search sequence.
PHASESEARCHBAND-WIDTH	To define the bandwidth used to design the 'debounce' controller used during the initial alignment stage of the phase search sequence.
PHASESEARCHCURRENT	To select amount of current applied to the motor during the phase search sequence.

Name	Description
PHASESEARCHINPUT	To set or read the digital input to be used as the phase search trigger input.
PHASESEARCHMODE	To turn on the 'debounce' controller used during the initial alignment stage of the phase search sequence.
PHASESEARCHOUTPUT	To assign a digital output as the phase search output.
PHASESEARCHSPEED	To select the speed of travel during the search sections of a phase search sequence.
PHASESEARCHSTATUS	To determine whether commutation is aligned on an axis.
PHASESEARCHSWITCH	To return the current state of the phase search input for the axis.
PHASESEARCHTRAVEL	To select the amount of travel during the search sections of a phase search sequence.
PLATFORM	To return the platform type.
PLCACTION	To read the action assigned to a PLC Task channel.
PLCACTIONPARAMETER	To read the associated parameter for an action assigned to a PLC Task channel.
PLCAUTOENABLE	To specify whether the PLC Task will automatically be enabled on power-up.
PLCCONDITION	To read a PLC Task channel's test condition.
PLCDEFAULT	To reset the PLC Task table to default settings.
PLCENABLE	To enable/disable the PLC Task.
PLCENABLEACTION	To enable/disable individual PLC Task channels.
PLCGEARFACTOR	To set or read the gear factor used by the 'Fast Gear' PLC Action.
PLCOPERATOR	To read the operator for a PLC Task channel.
PLCPARAMETER	To read the associated parameter used by a PLC Task channel's condition.
PLCSTATUS	To read a bit pattern of active (true) PLC Task channels.
PLCTASK	To set up PLC Task channels.
PLCTASKSTATUS	To read the current state of an individual PLC Task.
PLCTIME	To set or read the frequency of the PLC Task.
POS	To set or read the current axis position.
POSACHIEVED	To indicate whether the axis is ‘in position’.
POSDEMAND	To set or read the instantaneous position demand.
POSOFFSET	To set or read the offset used to calculate axis position for absolute encoders.
POSREF	To read the position reference value for an axis.
POSREMAINING	To indicate the remaining move distance.
POSROLLOVER	To count the number of wraps of the axis position value.

Name	**Description**
POSROLLOVERDEMAND	To return the number of position wraps required by the current move.
POSSCALEFACTOR	To scale axis encoder counts, or steps, into user defined position units.
POSSCALEUNITS	To define a text description for the position scale factor.
POSTARGET	To read the target position of the current positional move.
POSTARGETLAST	To read the target position of the last move in the move buffer.
POSWRAP	To set or read the position wrap range for the axis.
POWERREADYINPUT	To set or read the input used to inform a DC bus slave that mains power has been applied to the master.
POWERREADYOUTPUT	To set or read the output used by a DC bus master to inform a DC bus slave that mains power has been applied to the master.
PRECISIONINCREMENT	To read or set the theoretical distance between each of the values in the leadscrew compensation tables.
PRECISIONMODE	To control the action of leadscrew compensation.
PRECISIONOFFSET	To set the distance between the start of the leadscrew and axis zero position.
PRECISIONTABLE	To load the leadscrew compensation tables.
PRESETCANCEL	To set up a preset 'move' to perform a cancel command.
PRESETDWELLTIME	To specify a dwell time between a hardware trigger and the preset move starting.
PRESETHOME	To set up a homing type preset move.
PRESETINDEX	To read the current preset index or set a new index.
PRESETINDEXMODE	To set the controller's response to changes in a preset index.
PRESETINDEXSOURCE	To define the source for preset index changes.
PRESETINPUTSMAX	To define the number of preset moves available in the preset table.
PRESETINPUTSTATE	To read the current state of digital inputs representing the preset index.
PRESETJOG	To set up a jog preset move.
PRESETMOVEA	To set up an absolute preset move.
PRESETMOVEENABLE	To enable or disable preset moves.
PRESETMOVEPARAMETER	To define a preset move's parameters.
PRESETMOVER	To set up a relative preset move.
PRESETMOVESUSPEND	To pause a preset move.
PRESETMOVETYPE	To define the type of preset move.
PRESETPOS	To set up a preset 'move' to set the axis position value.
PRESETSELECTORINPUT	To assign the base input for preset index selection.

Name	Description
PRESETSPEEDREF	To set up a fixed point speed reference preset move.
PRESETSTOP	To set up a preset 'move' to perform a stop command.
PRESETTORQUEREF	To set up a fixed point torque reference preset move.
PRESETTRIGGERINPUT	To assign the input to be used as the preset index trigger.
PRODUCTCATALOG-NUMBER	To return the catalog number of the controller.
PRODUCTPOWERCYCLES	To return the number of times the controller has been power cycled.
PRODUCTSERIALNUMBER	To return the serial number of the controller.
PROFILEMODE	To select the type of velocity profiler to use.
PROFILETIME	To set the profiler update rate.
PULSECOUNTER	To return the value of the pulse input counter.
PULSEDIRMODE	To set the control mode for the step (pulse) & direction digital inputs.
PULSEOUTX	To activate a digital output for a specified number of milliseconds.
RELAY	To enable or disable the relay.
RELAYOUTPUT	To set the relay compatibility mode.
REMOTEADC	To read the value of a remote analog input (ADC).
REMOTEADCDELTA	To control the rate of change on a remote analog input before a REMOTEADC message is sent.
REMOTEBAUD	To specify the CAN baud rate of a remote BaldorCAN node (I/O or Keypad).
REMOTECOMMS	To access the reserved comms array on another controller.
REMOTECOMMSINTEGER	To access the reserved comms array on another controller, storing values as integers.
REMOTEDAC	To control the value of a remote analog output channel (DAC). The value is a percentage (positive and negative) of the full-scale output value.
REMOTEDEBOUNCE	To control the number of samples used to debounce an input on a remote CAN node.
REMOTEEMERGENCY-MESSAGE	Returns the error code from the last emergency message received from a particular CANopen node.
REMOTEENCODER	To read the value of a remote encoder channel.
REMOTEERROR	To read the CANopen error register information reported within the last emergency message received from a specific node.
REMOTEESTOP	To control the emergency stop state of a remote CAN node.
REMOTEIN	To read the state of all the digital inputs on a remote CAN node.
REMOTEINBANK	To read the state of a bank of digital inputs on a remote CAN node.

Name	Description
REMOTEINHIBITTIME	To set or read the CANopen PDO inhibit time.
REMOTEINPUT-ACTIVE-LEVEL	To control the active state of digital inputs on a remote CAN node.
REMOTEINX	To read the state of individual digital inputs from a remote CAN node.
REMOTEMODE	To control the update mode for a remote node.
REMOTENODE	To specify the node ID of a remote BaldorCAN node (I/O or Keypad).
REMOTEOBJECT	To access the Object Dictionary of any CANopen node present on the network.
REMOTEOBJECTFLOAT	To access ‘floating-point’ entries in the Object Dictionary of a remote node present on the network.
REMOTEOBJECTSTRING	To access 'Vis-String' entries in the Object Dictionary of any CANopen node present on the network.
REMOTEOUT	To control the state of digital outputs on a remote CAN node.
REMOTEOUTBANK	To read the state of a bank of digital outputs on a remote CAN node.
REMOTEOUTPUT-ACTIVELEVEL	To control the active state of digital outputs on a remote CAN node.
REMOTEOUTPUTERROR	To read or reset the digital outputs that are in error on a remote BaldorCAN node.
REMOTEOUTX	To control the state of individual digital outputs on a remote CAN node.
REMOTEPDOIN	To request data from a node in the form of a PDO message.
REMOTEPDOOUT	To force a controller node to transmit a variable length PDO message with a specific COB-ID. The PDO will contain up to 64 bits of data that can be passed in the form of two 32-bit values.
REMOTEPDOVALID	To read the status of the PDO (process data object) data for a node.
REMOTERESET	To force a remote CAN node to perform a software reset.
REMOTESTATUS	To set or read the status register on a remote CAN node.
RESET	To clear motion errors, set the position to zero and re-enable the drive.
RESETALL	To perform a reset on all axes.
RESETINPUT	To define the reset input for an axis.
SCALEFACTOR	To scale axis encoder counts, or steps, into user defined units.
SENTINEL	To set up sentinel channels.
SENTINELACTION	To control the action of a sentinel channel.
SENTINELACTIONMODE	To control how the action of a sentinel channel is performed.

Name	Description
SENTINELACTION-PARAMETER	To specify a parameter to fully define the sentinel action.
SENTINELLATCH	To determine whether a sentinel channel has become true since it was last checked.
SENTINELPERIOD	To control the time interval between sentinel samples.
SENTINELSOURCE	To read the source used by a sentinel channel.
SENTINELSOURCE2	To set or read the secondary source used by a sentinel channel.
SENTINELSOURCE2-PARAMETER	To set or read the parameter used to qualify the secondary sentinel source.
SENTINELSOURCE-PARAMETER	To read the source parameter used by a sentinel channel.
SENTINELSTATE	To read the current state of a sentinel channel.
SENTINELTRIGGER-ABSOLUTE	To read the 'absolute' parameter used by a sentinel channel.
SENTINELTRIGGERMODE	To read the 'mode' parameter used by a sentinel channel.
SENTINELTRIGGER-VALUE	To read the 'lowVal' or 'highVal' parameter used by a sentinel channel.
SENTINELTRIGGER-VALUEFLOAT	To specify the ‘lowVal’ or ‘highVal’ parameter, as a floating-point number, to be used in a sentinel channel's trigger criteria.
SENTINELTRIGGER-VALUEINTEGER	To specify the ‘lowVal’ or ‘highVal’ parameter, as an integer number, to be used in a sentinel channel's trigger criteria.
SERIALBAUD	To set the baud rate of the RS232 / RS485/422 port.
SERIALPROTOCOL	To select the serial protocol to be used.
SEXTANT	To read the current sextant value for a motor using Hall sensors.
SOFTLIMITFORWARD	To set the forward software limit position on a specified axis.
SOFTLIMITMODE	To set or read the default action taken if a forward or reverse software limit position is exceeded.
SOFTLIMITREVERSE	To set or read the reverse software limit position on a specified axis.
SPEED	To set or read the slew speed of positional moves loaded in the move buffer.
SPEEDDEMAND	To read the speed demand.
SPEEDERROR	To return the error between the demanded speed and the measured speed.
SPEEDERRORFATAL	To set or read the trip limit for the error between demanded and measured speed.
SPEEDMEASURED	To return the measured speed.
SPEEDREF	To set or read a fixed point speed reference.
SPEEDREFACCELTIME	To set or read the acceleration ramp for a speed profile.

Name	Description
SPEEDREFDECELTIME	To set or read the deceleration ramp for a speed profile.
SPEEDREFENABLE	To enable speed command mode.
SPEEDREFERROR-DECELTIME	To set a deceleration ramp for a speed profile in the event of an error.
SPEEDREFSOURCE	To specify the source of the speed reference command.
SPLINE	To perform a spline move.
SPLINEEND	To define the end segment in the spline table for a spline move.
SPLINEINDEX	To read the currently executing spline segment number.
SPLINESEGMENT	To change spline table data.
SPLINESTART	To define the start segment in a spline table for a spline move.
SPLINESUSPENDTIME	To set the segment duration for a controlled stop during a spline move.
SPLINETABLE	To specify the array names to be used in a spline move on the specified axis.
SPLINETIME	To set the segment duration for all segments for a spline move.
SRAMP	To set the percentage of S-ramping applied to linear moves.
STEPPER	To set or read the stepper axis value.
STEPPERDELAY	To enforce a time delay between state changes on step and direction outputs.
STEPPERIO	To manually control the step and direction pins of a stepper channel.
STEPPERMODE	To make miscellaneous changes to the steppers.
STEPPERSCALE	To set or read the scale factor for the stepper output channel.
STEPPERVEL	To read the velocity from a stepper output channel.
STEPPERWRAP	To set or read the stepper wrap range for a stepper output channel.
STOP	To perform a controlled stop during motion.
STOPINPUT	To set or read the digital input to be used as the stop switch input for the specified axis.
STOPINPUTMODE	To set or read the action taken in the event of a stop input becoming active.
STOPMODE	To set or read the action taken when an axis is stopped.
STOPSWITCH	To return the current state of the stop input for the axis.
SUSPEND	To pause the current move.
SUSPENDINPUT	To set or read the digital input to be used as the suspend switch input for the specified axis.
SUSPENDSWITCH	To return the current state of the suspend input for the axis.
SYSTEMDEFAULTS	To reset parameter table entries to their default values and erase the Mint program, non-volatile RAM and error log.

Name	Description
SYSTEMSECONDS	To set or read a programmable system lifetime counter for the drive.
SYSTEMTIMEMODE	To specify whether system time data is stored to non-volatile memory.
TEMPERATURE	To report the internal drive temperature.
TEMPERATURELIMIT-FATAL	To set or read the temperature fatal limit.
TEMPERATURELIMIT-WARNING	To read the temperature warning limit.
TERMINALADDRESS	To set or read the node ID for a CAN node associated with a terminal.
TERMINALDEVICE	To set or read the device type associated with a given terminal.
TERMINALMODE	To set or read handshaking modes for a terminal.
TERMINALPORT	To set or read the communication port associated with a given terminal.
TIMEREVENT	To set or read the rate of the timer event.
TIMESCALE	To scale speed related values into user time units.
TORQUE	To execute torque control (constant current) on a servo axis.
TORQUEDEMAND	To return the instantaneous torque demand.
TORQUEFILTERBAND	To define the band of operation for a torque filter stage.
TORQUEFILTERDEPTH	To define the reduction in gain for a notch torque filter stage.
TORQUEFILTERFREQ	To define a characteristic frequency for a torque filter stage.
TORQUEFILTERTYPE	To define the type of characteristic used for the given torque filter stage.
TORQUELIMITNEG	To set or read the maximum negative torque limit.
TORQUELIMITPOS	To set or read the maximum positive torque limit.
TORQUEREF	To set or read a torque reference for torque (constant current) mode on a servo axis.
TORQUEREFENABLE	To set the drive into torque command mode.
TORQUEREFERRORFALL-TIME	To set or read the 'deceleration ramp' for a torque profile in the event of an error.
TORQUEREFFALLTIME	To set or read the 'deceleration ramp' for a torque profile.
TORQUEREFRISETIME	To set or read the 'acceleration ramp' for a torque profile.
TORQUEREFSOURCE	To specify the source of the torque reference command.
TRIGGERCHANNEL	To specify the input used for triggering, when triggering on an axis source or encoder.
TRIGGERCOMPENSATION	To specify the size of the compensation term used when axis triggering on an axis/encoder position.
TRIGGERINPUT	To specify the input used for triggering, when triggering on a digital input.

Name	Description
TRIGGERLATCH	To specify the latch channel used for triggering, when triggering on a latch channel.
TRIGGERMODE	To control the triggering of a move.
TRIGGERSOURCE	To specify the source when axis triggering is using an axis/ encoder position.
TRIGGERVALUE	To specify an absolute value on which to trigger motion.
USERPARAMETER	To provide access to user-programmable parameters stored in EEPROM.
USERPOSITIONUNITS	To define a text description for the user unit.
USERTIMEUNITS	To define a text description for the user time unit.
VECTORA	To perform an interpolated vector move on two or more axes with absolute co-ordinates.
VECTORR	To perform an interpolated vector move on two or more axes with relative co-ordinates.
VEL	To return the instantaneous axis velocity.
VELDEMAND	To read the current instantaneous demand velocity.
VELDEMANDPATH	To read the instantaneous demand velocity along the path of a multi-axis move.
VELERROR	To report the velocity following error.
VELFATAL	To set or read the threshold for the maximum difference between demand and actual velocity.
VELFATALMODE	To control the default action taken in the event of the velocity threshold being exceeded.
VELREF	To set or read a fixed point speed reference.
VELSCALEFACTOR	To scale axis encoder counts, or steps, into user defined velocity units.
VELSCALEUNITS	To define a text description for the velocity scale factor.
VELSETPOINTMAX	To set or read the maximum limit of a velocity band.
VELSETPOINTMIN	To set the minimum limit of a velocity band.
VOLTAGEDEMAND	To read the voltage demand outputs from the current controllers.

13.8 Glossary

A

Action statement One that performs a computational action, like an assignment or a subroutine call (c.f. declaration statement).

ActiveX A control used to allow access to a resource, the Mint ActiveX allowing the host computer access to a controller.

Ada A programming language (derived from Pascal and named after Ada Lovelace) commissioned by the US department of defense to replace the plethora of languages they once used.

Aggregate A collection of values, usually an array or a structure.

Alphabetic A character in the range ‘a’ to ‘z or ‘A’ to ‘Z.

Alphanumeric A character that is either alphabetic or numeric.

API Application Programming Interface.

Array A collection of values all of the same type that may be accessed by indexing a specific element.

ASCII American Standard Code for Information Interchange, a specification of the characters with ordinal values 0 to 127.

B

Bankers’ rounding Round to the nearest even digit in the case of the following digit being exactly 5. More correctly called “Round to Nearest Even" or “Unbiased Rounding".

Base An alternative to “radix", so base 2 is binary, base 10 is decimal, etc.

Binary Base 2 (may use digits ‘0’ and ‘1’), usually denoted in text using the subscript 2, e.g. 111010_2, or in Mint Basic using the prefix `2#`, e.g. `2#111010`.

Bitfield A data-type that allows integers mapped to specific bit ranges within a 32-bit value to be read/written by name.

Block A statement which when executed causes the statements it contains to be executed.

Boolean A logical state that may take only the values false or true.

Boot-up A system boots-up when it starts operating, either on power-up or after resetting, and configures itself for operation during this phase.

Byte The smallest unit of storage that a processing unit can access. Typically, this is 8 bits long, but some systems use 16 or 32 bits, and some early computer architectures used 9 bits. Though the architectures that the MVM runs on support 8, 16 and 32 bit bytes, this only matters for embedded programming and so whenever the term is used in this document, 8 bit bytes are meant.

C

CAN Controller Area Network. A network originally developed by Bosch in 1985 for in-vehicle networks with the objective of reducing wiring loom sizes. It proved popular and was made an ISO standard in 1993 (ISO 11898) and is now widely used in many application areas, such as in medical, avionics and automation systems.

Cast In a computing sense, a contraction of the term “type cast".

Character An integer value in the range 0 to 255 that represents an ASCII code, which for convenience may be defined by enclosing the required character within single quotes, e.g. `'0'` defines that character for the digit zero, and has the integer value 48.

Compiler A program that converts from one language to another, usually from a high-level programming language to a low-level machine language.

Concatenation The joining together of two items, usually strings.

Conditional compilation The ability to use or ignore sections of code based on a condition that is evaluated during compilation, thus not incurring any run-time penalty and allowing increased flexibility with regard to what the program contains and how it is targeted.

Contiguous Something that is continuous, containing no gaps.
Controller A programmable electronic device used to control hardware.
Control character A character with ordinal value 0 to 31 or 127.
CPU Central Processing Unit.
Critical section A section of code that must be protected from the influence of other tasks, for example, the body of a `Semaphore` block or a `Critical` block are examples of critical sections.
Cursor The location where the next character will be written, usually signified by a flashing block or similar.
C# An object oriented language based on C++ and Java that was developed by Microsoft to be the principle programming language of the .NET environment.

D

Data-type The nature of the data stored in a variable is defined by its data-type (or 'type' for short). Data-types must be differentiated because they each have fundamentally different properties and storage requirements.
Deadlock A state where multiple tasks are completely inhibiting the progress of each other.
Decimal Base 10 (may use digits '0' to '9'), no denotation usually used, though in text can be denoted using a subscript 10, e.g. 101_{10}, or in Mint Basic using the prefix `10#`, e.g. `10#101`.
Declaration statement One that creates a named entity that can be accessed by using its name, such as a constant or a subroutine (c.f. action statement).
Directive statement One that merely directs the compiler to act in a particular way, for example to perform certain optimizations.
DPR Dual Port RAM.
Drive An electronic amplifier designed to drive an electric motor. A drive that is not programmable relies on an external input (often from a controller) to specify the required motor movement.
DSP Digital Signal Processor.

E

Event A named block of code that is automatically executed in response to some event occurring.
Exception An abnormal condition that results in an error.
Exponent The value used to raise a number to the power of (exponentiation). Usually used in scientific notation where the value is 10 (for decimal numbers) and the exponent is a whole number, positive for scaling up and negative for scaling down.
Exponentiation The raising of one number to the power of another.

F

Fatal error An error that causes immediate termination of the program (cf. non-fatal error).
Floating-point A decimal number that includes a decimal point and optionally an exponent.
Function A named block of code that executes sequentially in the calling task and which returns a result.

G

Globals A term commonly used to describe variables with global scope that can be accessed from anywhere in a program (cf. Locals).

H

Hexadecimal Base 16 (may use digits ‘0’ to ‘9’ and ‘a’ to ‘f’), usually denoted in text using the subscript 16, e.g. 190_{16}, or in Mint Basic using the prefix `16#`, e.g. `16#190`.

HMI Human Machine Interface. Usually a keypad with a display, or a touchscreen device.

I

ICM Immediate Command Mode. A proprietary high-level communications protocol used to execute functions on the controller or drive, such as reading or writing an axis position, loading a move, running a Mint Basic program, downloading firmware, etc.

Identifier A name used to identify a declaration.

IEC International Electrotechnical Commission, the international standards and conformity assessment body for all fields of electrotechnology.

IEC 61131-3 A standard that defines a number of programming languages for use in programming PLCs.

IEEE 754 A standard for floating-point data that uses binary encoding.

IEEE 854 A standard for floating-point data that may use either binary or decimal encoding.

Integer A whole number, possibly negative.

Intrinsic Something built into the core language, like the Sin function.

I/O Input/Output.

Iteration Repetition.

J

Jerk The rate of change of acceleration, used in S-ramping to define the extent of the S, a low jerk giving a large S and a high value giving a small S. As jerk tends to infinity, the motion will tend to trapezoidal.

K

Keyword A special word used by Mint Basic that may not be used for other purposes. These are also referred to as “Reserved words".

L

Label A name that defines a location in the program used by GoTo and the blocks on which Exit and Continue operate.

Lifetime A term used to describe the period that a variable exists.

Literal A value expressed directly, e.g. a number like `12` or `0.375`, or a string like ‘`Enter value: `’, or a character like ‘=’ (which is really just a number).

Livelock A state similar to deadlock, but where execution is proceeding, though in such a way that nothing useful is being achieved.

Locals A term commonly used to describe variables with local scope that can only be accessed from within the module in which they are declared (cf. Globals).

Lower case Non-capitalized letters, the name deriving from a time when typesetters stored these letters in a case below the case containing the capital letters.

M

Machine language The native language of a machine, typically a CPU or DSP, or in the case of Mint Basic, the MVM.

MML Mint Motion Library. A library of commands and functions that allow access to the hardware of a controller or drive.

Module A block of code that can be executed, usually via reference to its name (Sub, Function, Task and Startup (the latter two using the Run command)), or automatically (Startup, Shutdown and Event).

Mutex A means of preventing simultaneous access to a common resource, typically a section of code (called a ‘critical section’), in a multi-tasking environment. A mutex is like a binary semaphore, and, depending on the implementation, may be re-entrant/recursive, which means that a process may acquire the same mutex multiple times without blocking, though it must be released as many times as it was acquired in order to become free for acquisition by other processes.
MVM Mint Virtual Machine. This is a VM used by all ABB controllers and drives that are programmable using Mint Basic. It makes all the devices behave similarly, providing a consistent range of features across the entire product range.

N
Non-fatal error An error that is not so severe that termination is required, allowing the error handler to be invoked if present to try and rectify the problem, otherwise causing immediate termination.
Numeric A number, or with reference to a character, one in the range “0” to “9”.

O
O Big ‘O’ notation is used to indicate what some property is proportional to, typically execution time. For example a linear process is O(*n*), a quadratic process is O(n^2), a cubic process is O(n^3), etc., where n indicates the amount of data to process. Clearly, the performance of a higher order process degrades at a more rapid rate than a corresponding lower order process (as the amount of data increases), the best case being a constant time process, denoted by O(*1*), the performance of which is independent of size.
Octal Base 8 (may use digits ‘0’ to ‘7’), usually denoted in text using the subscript 8, e.g. 177_8, or in Mint Basic using the prefix 8#, e.g. 8#177.
Operator Something that operates on its operand data. Operators can be named (Xor), symbolic (*) or both (Mod/%), and may take an arbitrary number of operands, though only unary and binary operators are common, with ternary operators either looking like function calls (Mint Basic / VB) or using a multi-symbol operator (e.g. C/C++). Ternary operators, being operators rather than function calls, are evaluated differently, as the parameters to a function are always evaluated, whereas the operands to a ternary operator need not be.
Operand An item of data manipulated by an operator, e.g. ‘x’ and ‘y’ are the operands to the ‘+’ operator in the expression ‘x + y’.

P
PAC Programmable Automation Controller. A powerful PLC that combines the reliability of a PLC with the capabilities of a PC based control system. This gives the capability to do such things as process control, data acquisition, machine visualization, remote monitoring, motion control, etc.
Parameter A variable used to pass data to a subroutine or function. These can also be called “Arguments” or sometimes “Operands”.
PLC Programmable Logic Controller. This is a device that can process multiple inputs and programmatically control multiple outputs in real time. They are used in many industries, typically to control some part of a machine.
Pointer A variable that contains a memory address, thus making it ‘point’ to that memory location.
Precedence A level of importance that is used to decide in what order to process things, used in expressions (operator precedence) and event execution (event precedence).
Priority A measure of the importance of a task or an event.
Program A sequence of statements designed to achieve some objective.
Pseudo-Random An apparently random sequence determined by some mathematical equation, thus making it not truly random.

R

Radix The base of a number, so radix 10 is decimal, radix 8 is octal, etc.
Recursion A subroutine or function that calls itself, either directly or indirectly (mutual recursion).
Reserved word A word that is reserved for use in the language definition, and as such it cannot be used as an identifier.

Q

Quantum The size of a task’s time-slice, generally measured in time units, but in the case of the MVM, measured in instructions.

R

Radix The base of a number, so radix 10 is decimal, radix 8 is octal, etc.
RAM Random Access Memory.
Recursion A subroutine or function that calls itself, either directly or indirectly (mutual recursion).
Reserved word A word that is reserved for use in the language definition, and as such cannot be used as an identifier.
Rotate Data is rotated by moving its bit pattern left or right a specified number of bits, with bits that 'fall off' one end reappearing at the other end. Rotating may be performed on signed or unsigned data, but Mint Basic only supports the latter.

S

SCADA Supervisory control and data acquisition. A system that supervises control (i.e. coordinates processes but does not directly control in real time) and monitors the components of the system, allowing operators to view its status and have its history stored in a database.
Scientific notation A notation used to scale numbers using an exponent to make them more readable. In text form this is denoted by appending “$\times 10^{-12}$”, and in Mint Basic by appending “e-12” (this example uses an exponent of -12, but any integer is valid).
Scope The visibility of declarations based on the nesting of modules such that the current module is searched first, followed by the module enclosing it, etc., until it is found.
Semaphore A means of preventing simultaneous access to a common resource, typically a section of code (called a ‘critical section’), in a multi-tasking environment. Semaphores are not re-entrant/recursive, and so a task can block itself if it attempts to acquire a semaphore that it has already acquired. While a binary semaphore is like a mutex, a semaphore also has the ability to allow a number of processes access to a resource when a mutex can only ever allow one process access.
Semaphore variable A variable used to control access to a resource by using it in a semaphore block. The number of acquisitions allowed may be specified when it is declared, and if not done so, then a binary semaphore is assumed.
Semaphore block A block that contains a sequence of statements that can be executed only when the specified semaphore can be acquired.
Shift Data is shifted by moving its bit pattern left or right a specified number of bits, with bits that 'fall off' one end being lost and bits that appear at the other being zeroed. Shifting may be performed on signed or unsigned data, but Mint Basic only supports the latter.
Short circuit A term used in computing to describe the cutting short of an operation once the outcome is known.
Shutdown module A block of code that automatically executes whenever a program terminates.
Side effect Occurs when a module alters data declared outside of itself, making verification difficult.

S-ramp In terms of motion profiling, a velocity graph that has curved acceleration and deceleration phases for smoother motion, controlled by the jerk setting (cf. trapezoidal profiling).
Startup module A block of code that automatically executes on start up or when the Run Startup command is issued.
Starvation A state where a task is so starved of access to a resource or CPU time that it makes no useful progress.
Statement An element of code used to either declare something or do something (c.f. declaration statement and action statement), which can be simple (e.g. an assignment or call) or structured (e.g. a loop or a task declaration).
String A sequence of characters enclosed within double quotes, for example "`Speed = `".
Structure A collection of values of arbitrary type that may be accessed by accessing its members.
Structured text A textual programming language similar to Pascal that is part of the IEC 61131-3 framework.
Subroutine A named block of code that executes sequentially in the calling task.

T

Target format In terms of the MVM, it encapsulates the instruction set and the format of the MEX file that the MVM must decode. The MVM validates the MEX file’s target format prior to execution, only running it if it matches.
Task A named block of code that executes in parallel with other tasks.
Terminal An I/O device that may send or receive character data (e.g. a VT100 terminal, or CAN keypad).
Trapezoidal In terms of motion profiling, the velocity graph will be composed of straight lines (cf. S-ramp).
Type See Data-type.
Type cast The conversion from one data-type to another, which may be performed implicitly by the compiler or explicitly by using a function like `Int`.

U

Upper case Capitalized letters, the name deriving from when typesetters stored these letters in a case above the case containing the non-capital letters.

V

VB Visual Basic, Microsoft’s dialect of Basic used for programming visual applications (i.e. using a graphical user interface).
VM Virtual Machine. This is an abstract machine, typically used to mimic the operation of a real machine but using a software implementation rather than hardware. The machine implemented is often abstract itself, as it is simply a means to an end, the objective usually being a common run-time system to aid porting to a variety of target platforms. Examples of this are the P-machine that enabled the widespread use of Pascal, the Java Virtual Machine, the Common Language Infrastructure that is the heart of the .NET framework, and the Mint Virtual Machine.
VHDL A language commissioned by the US department of defense that shares many features with Ada. Its name is derived from VHSIC Hardware Description Language (Very High Speed Integrated Circuits).

W

Warning These are used to indicate that something may be wrong, such as passing an integer variable to a parameter that is a floating-point reference. It is good practice to eliminate warnings from a program.

Wrap Wrapping occurs when something overflows off one end of a scale and reappears on the other end.

If you have any suggestions for improvements to this manual, please let us know. Write your comments in the space provided below, remove this page from the manual and mail it to:

Manuals
ABB Ltd
Motion Control
6 Bristol Distribution Park
Hawkley Drive
Bristol
BS32 0BF
United Kingdom.

Alternatively, you can e-mail your comments to:

manuals.uk@baldor.com

Comment:

continued...

Thank you for taking the time to help us.

Further information

Product and service inquiries

Address any inquiries about the product to your local ABB representative, quoting the type designation and serial number of the unit in question. A listing of ABB sales, support and service contacts can be found by navigating to www.abb.com/drives and selecting *Sales, Support and Service network*.

Product training

For information on ABB product training, navigate to www.abb.com/drives and select *Training courses*.

Providing feedback on ABB Drives manuals

Your comments on our manuals are welcome. Go to www.abb.com/drives and select *Document Library – Manuals feedback form (LV AC drives)*.

Document library on the Internet

You can find manuals and other product documents in PDF format on the Internet. Go to www.abb.com/drives and select *Document Library.* You can browse the library or enter selection criteria, for example a document code, in the search field.

Contact us

ABB Oy
Drives
P.O. Box 184
FI-00381 HELSINKI
FINLAND

Telephone	+358 10 22 11
Fax	+358 10 22 22681

www.abb.com/drives

ABB Inc.
Automation Technologies
Drives & Motors
16250 West Glendale Drive
New Berlin, WI 53151
USA

Telephone	262 785-3200
	1-800-HELP-365
Fax	262 780-5135

www.abb.com/drives

ABB Beijing Drive Systems Co. Ltd.
No. 1, Block D, A-10 Jiuxianqiao Beilu
Chaoyang District
Beijing, P.R. China, 100015

Telephone	+86 10 5821 7788
Fax	+86 10 5821 7618

www.abb.com/drives

Baldor Electric Company
(A member of the ABB group)
5711 R.S Boreham, Jr. St.
P.O. Box 2400
Fort Smith, AR 72901
USA

Telephone	+1 479 646 4711
Fax	+1 479 648 5792

www.baldor.com

ABB Ltd
Motion Control
6 Bristol Distribution Park
Hawkley Drive
Bristol, BS32 0BF
United Kingdom

Telephone	+44 (0) 1454 850000
Fax	+44 (0) 1454 859001

www.abb.com/drives

LT0255A03 (EN) EFFECTIVE: 2012-09-01

LT0255A03EN

Power and productivity
for a better world™