MMBasic Structures User Manual

Introduction

Structures (also known as User-Defined Types) allow you to group related variables of different types together under a single name. This is useful for organizing complex data such as coordinates, records, or any collection of related values.

Defining a Structure Type

Use the "TYPE...END TYPE" block to define a new structure type:

```
Type typename
  member1 As type
  member2 As type
  ...
End Type
```

Supported Member Types

- "INTEGER" - 64-bit signed integer
- "FLOAT" - 64-bit floating point number
- "STRING" - String up to 255 characters (use "LENGTH n" to specify maximum length)

Examples

Simple structure:

```
Type Point
  x As INTEGER
  y As INTEGER
End Type
```

Structure with mixed types:

```
Type Person
  age As INTEGER
  height As FLOAT
  name As STRING
End Type
```

Structure with string length specified:

```
Type Record
  id As INTEGER
  description As STRING LENGTH 100
End Type
```

Memory Layout and Alignment

Understanding how structures are packed in memory is important when working with binary file I/O or calculating memory usage.

Member Storage Sizes

Type	Storage Size
`INTEGER`	8 bytes
`FLOAT`	8 bytes
`STRING`	length + 1 bytes (1 byte for length prefix + specified/default leng...
`STRING LENGTH n`	n + 1 bytes
Nested structure	Size of the nested structure type

Alignment Rules

Members are placed sequentially in memory with the following alignment rules:
- **Strings**: No alignment requirement. Placed immediately after the previous member.
- **INTEGER, FLOAT, and nested structures**: Aligned to 8-byte boundaries. If the current offset is not divisible by 8, padding bytes are inserted before the member.

Padding Example

When a numeric type follows a string whose total storage is not aligned to 8 bytes, padding is automatically inserted:

```
Type Example1
  name As STRING LENGTH 10   ' Offset 0, size 11 bytes (10 + 1 length byte)
  value As INTEGER           ' Offset 16 (padded from 11 to align to 8)
End Type
' Total size: 24 bytes (11 + 5 padding + 8)

Type Example2
  name As STRING LENGTH 15   ' Offset 0, size 16 bytes (15 + 1 length byte)
  value As INTEGER           ' Offset 16 (no padding needed, already aligned)
End Type
' Total size: 24 bytes (16 + 8)

Type Example3
  a As INTEGER               ' Offset 0, size 8 bytes
  name As STRING LENGTH 5    ' Offset 8, size 6 bytes (5 + 1 length byte)
  b As INTEGER               ' Offset 16 (padded from 14 to align to 8)
End Type
' Total size: 24 bytes (8 + 6 + 2 padding + 8)
```

Optimizing Structure Size

To minimize wasted space from padding, consider:
1. **Grouping numeric members together**: Place all INTEGER and FLOAT members consecutively.
2. **Using string lengths that result in 8-byte aligned totals**: String lengths of 7, 15, 23, 31, etc. (where length + 1 is divisible by 8) avoid padding when followed by numeric types.

Note on structure end padding: Padding is always added to the end of a structure to ensure the total size is aligned to 8 bytes. This is required so that arrays of structures maintain proper memory alignment for all elements. Without end padding, the second element of an array would start at a misaligned address, potentially causing memory access errors.

```
' Less efficient (has internal padding):
Type Inefficient
  flag As INTEGER            ' 8 bytes
  name As STRING LENGTH 10   ' 11 bytes
  count As INTEGER           ' 8 bytes (but needs 5 bytes padding before it)
End Type
' Total: 32 bytes

' More efficient (no internal padding, but end padding still applies):
Type Efficient
  flag As INTEGER            ' 8 bytes
  count As INTEGER           ' 8 bytes
  name As STRING LENGTH 10   ' 11 bytes + 5 bytes end padding
End Type
' Total: 32 bytes (padded to 8-byte boundary)
```

Use "STRUCT(SIZEOF "typename")" to verify the actual size of your structures.

Declaring Structure Variables

Use "DIM" to declare variables of a structure type:

```
Dim variablename As typename
```

Examples

Simple structure variable:

```
Type Point
  x As INTEGER
  y As INTEGER
End Type

Dim p As Point
Dim origin As Point
```

Multiple variables:

```
Dim p1 As Point, p2 As Point, p3 As Point
```

Accessing Structure Members

Use the dot (".") notation to access individual members:

```
variablename.membername
```

Examples

Setting values:

```
Dim p As Point
p.x = 100
p.y = 200
```

Reading values:

```
Print p.x, p.y
result = p.x + p.y
```

Using in expressions:

```
distance = Sqr(p.x * p.x + p.y * p.y)
```

Arrays of Structures

You can create arrays where each element is a structure:

```
Dim arrayname(size) As typename
```

Examples

Declaring an array of structures:

```
Dim points(10) As Point
```

Accessing array elements:

```
points(0).x = 10
points(0).y = 20
points(1).x = 30
points(1).y = 40

Print points(0).x, points(0).y
```

Using variable indices:

```
For i = 0 To 10
  points(i).x = i * 10
  points(i).y = i * 20
Next i
```

Multi-dimensional arrays:

```
Dim grid(10, 10) As Point
grid(5, 5).x = 100
```

```
  grid(5, 5).y = 200
```

Initializing Structures

Structures can be initialized when declared using parentheses with comma-separated values:

```
  Dim variablename As typename = (value1, value2, ...)
```

Values must be provided in the order the members are defined in the TYPE block.

Examples

Simple structure initialization:

```
  Type Point
    x As INTEGER
    y As INTEGER
  End Type

  Dim p As Point = (100, 200)
  ' p.x = 100, p.y = 200
```

Structure with string:

```
  Type Person
    age As INTEGER
    height As FLOAT
    name As STRING
  End Type

  Dim person1 As Person = (25, 1.75, "Alice")
```

Array of structures initialization:

```
  Dim points(2) As Point = (10, 20, 30, 40, 50, 60)
  ' points(0).x = 10, points(0).y = 20
  ' points(1).x = 30, points(1).y = 40
  ' points(2).x = 50, points(2).y = 60
```

Values are assigned sequentially: all members of element 0, then all members of element 1, etc.

Copying Structures

Structures can be copied using direct assignment or the "STRUCT COPY" command.

Direct Assignment

The simplest way to copy a structure is using direct assignment:

```
  destination = source
```

This works for:

- Single structure variables
- Individual array elements

Example - Single structures:

```
  Dim src As Point, dst As Point
  src.x = 100
  src.y = 200

  dst = src
  ' dst.x = 100, dst.y = 200
```

Example - Array elements:

```
  Dim points(10) As Point
  points(0).x = 50 : points(0).y = 60
```

```
points(5) = points(0)    ' Copy element 0 to element 5
' points(5).x = 50, points(5).y = 60
```

Example - Cross-array copy:

```
Dim src(5) As Person
Dim dst(5) As Person
' ... populate src ...

dst(2) = src(3)    ' Copy element 3 from src to element 2 of dst
```

Important: Both source and destination must be the same structure type. Attempting to assign structures of different types will cause an error.

STRUCT COPY Command

The "STRUCT COPY" command provides the same functionality with explicit syntax:

```
Struct Copy source To destination
```

Both variables must be of the same structure type.

Example:

```
Dim src As Point, dst As Point
src.x = 100
src.y = 200

Struct Copy src To dst
' dst.x = 100, dst.y = 200
```

Copying Entire Arrays

You can copy entire structure arrays using empty parentheses:

```
Struct Copy sourceArray() To destinationArray()
```

Requirements:

- Both arrays must be the same structure type
- The destination array must be at least as large as the source array
- Both must use the "()" syntax, or both must be single elements
- Only the source elements are copied (extra destination elements are preserved)

Example:

```
Dim src(2) As Point
src(0).x = 10 : src(0).y = 11
src(1).x = 20 : src(1).y = 21
src(2).x = 30 : src(2).y = 31

Dim dst(4) As Point  ' Larger destination is OK
Struct Copy src() To dst()
' dst(0), dst(1), dst(2) now contain copies from src
' dst(3), dst(4) are unchanged
```

Example - Same size arrays:

```
Dim original(10) As Person
' ... populate array ...

Dim backup(10) As Person
Struct Copy original() To backup()
```

Sorting Structure Arrays

Use the "STRUCT SORT" command to sort an array of structures in-place based on any member field:

```
  Struct Sort array().membername [, flags]
```

Parameters

- "array().membername" - The structure array with the member to sort by. The array name must include empty parentheses followed by a dot and the member name (e.g., "people().age")
- "flags" - Optional flags to modify sort behavior (can be combined by adding):

Value	Description
:-----:	----------
0	Default: ascending sort, case-sensitive
1	Reverse sort (descending order)
2	Case-insensitive sort (strings only)
4	Empty strings sort to end of array (strings only)

Flags can be combined by adding values (e.g., 3 = descending + case-insensitive).

Examples

Sort by integer field (ascending):

```
  Type Person
    age As INTEGER
    name As STRING
  End Type

  Dim people(3) As Person
  people(0).age = 35 : people(0).name = "Charlie"
  people(1).age = 25 : people(1).name = "Alice"
  people(2).age = 45 : people(2).name = "David"
  people(3).age = 30 : people(3).name = "Bob"

  Struct Sort people().age
  ' Result: Alice(25), Bob(30), Charlie(35), David(45)
```

Sort by string field:

```
  Struct Sort people().name
  ' Result: Alice, Bob, Charlie, David
```

Reverse sort (descending):

```
  Struct Sort people().age, 1
  ' Result: David(45), Charlie(35), Bob(30), Alice(25)
```

Case-insensitive string sort:

```
  Struct Sort people().name, 2
```

Combine flags (reverse + case-insensitive):

```
  Struct Sort people().name, 3
  ' 3 = 1 + 2 (reverse + case insensitive)
```

Empty strings at end:

```
  Struct Sort people().name, 4
  ' Non-empty strings sorted first, empty strings at end
```

Notes

- The entire structure is moved during sorting, not just the sort key
- All other member values are preserved with their corresponding records
- The sort is performed in-place (no additional array is created)
- Array members within structures cannot be used as sort keys

- Supports INTEGER, FLOAT, and STRING member types

Extracting Structure Members to Arrays

Use the "STRUCT EXTRACT" command to copy a single member from each element of a structure array into a simple contiguous array. This is essential when you need to use structure data with commands that expect arrays with fixed 8-byte stride (such as "LINE PLOT", "MATH SCALE", etc.).

```
Struct Extract structarray().membername, destarray()
```

Parameters

- "structarray().membername" - The source structure array with the member to extract. The array name must include empty parentheses followed by a dot and the member name (e.g., "readings().temperature")
- "destarray()" - The destination simple array (must include empty parentheses)

Requirements

- Source must be a **1-dimensional** structure array
- Member must be a simple type (INTEGER, FLOAT, or STRING) - not an array member or nested structure
- Destination must be a **1-dimensional** simple array (not a structure)
- **Both arrays must have the same cardinality** (number of elements)
- **Both must have the same type** (INTEGER, FLOAT, or STRING)
- **For strings, both must have the same maximum length**

Why This Command is Needed

Some MMBasic commands that process arrays assume data is stored contiguously with a fixed stride of 8 bytes between elements. When you have data in a structure array like:

```
Type Room
  temperature As FLOAT   ' offset 0
  humidity As FLOAT      ' offset 8
End Type

Dim readings(144) As Room
```

The "temperature" values are 16 bytes apart (the size of the structure), not 8 bytes. Commands that are not structure-aware would read incorrect memory locations for elements after the first.
"STRUCT EXTRACT" solves this by copying the desired member values into a properly-formatted simple array for use with commands that don't support structure member arrays directly.
Note: Many commands (like "LINE PLOT", "POLYGON", various "MATH" commands, and most drawing commands) now support structure member arrays directly. See the "Using Structure Arrays with Drawing Commands" and "Using Structure Arrays with MATH Commands" sections for details.

Examples

Extract temperatures for plotting:

```
Type Room
  temperature As FLOAT
  humidity As FLOAT
End Type

Dim readings(144) As Room
Dim temps(144)

' ... populate readings() with sensor data ...

' Extract temperature values into a simple array
Struct Extract readings().temperature, temps()
```

```
' Now temps() can be used with LINE PLOT
LINE PLOT temps()
```

Extract integer values:

```
Type DataPoint
  timestamp As INTEGER
  value As INTEGER
End Type

Dim samples(100) As DataPoint
Dim values%(100)

' ... populate samples() ...

' Extract just the values for processing
Struct Extract samples().value, values%()

' Now values%() can be used with MATH commands
Math Scale values%(), 2, values%()
```

Extract string data:

```
Type Person
  name As STRING LENGTH 20
  age As INTEGER
End Type

Dim people(50) As Person
Dim names$(50) LENGTH 20

' ... populate people() ...

' Extract all names
Struct Extract people().name, names$()
```

Notes

- The extraction is a one-time copy - changes to the destination array do not affect the source structures
- Use this command when you need to pass structure member data to array-processing functions
- For best performance, extract data once before processing rather than repeatedly in loops

Inserting Array Values into Structures

Use the "STRUCT INSERT" command to copy values from a simple array into a specific member of each element in a structure array. This is the reverse of "STRUCT EXTRACT".

```
Struct Insert srcarray(), structarray().membername
```

Parameters

- "srcarray()" - The source simple array (must include empty parentheses)
- "structarray().membername" - The destination structure array with the member to write to. The array name must include empty parentheses followed by a dot and the member name (e.g., "readings().temperature")

Requirements

- Source must be a **1-dimensional** simple array (not a structure)
- Destination must be a **1-dimensional** structure array
- Member must be a simple type (INTEGER, FLOAT, or STRING) - not an array member or nested structure
- **Both arrays must have the same cardinality** (number of elements)

- **Both must have the same type** (INTEGER, FLOAT, or STRING)
- **For strings, both must have the same maximum length**

Examples

Insert processed values back into structures:

```
Type Room
  temperature As FLOAT
  humidity As FLOAT
End Type

Dim readings(144) As Room
Dim temps!(144)

' Extract temperatures
Struct Extract readings().temperature, temps!()

' Process the data (e.g., apply calibration)
Math Scale temps!(), 1.02, temps!()
Math Add temps!(), -0.5, temps!()

' Insert corrected values back into structures
Struct Insert temps!(), readings().temperature
```

Populate structure members from computed arrays:

```
Type DataPoint
  x As INTEGER
  y As INTEGER
End Type

Dim points(100) As DataPoint
Dim xvals%(100), yvals%(100)

' Generate coordinate arrays
For i% = 0 To 100
  xvals%(i%) = i% * 10
  yvals%(i%) = Sin(i% * 0.1) * 100
Next i%

' Insert into structure array
Struct Insert xvals%(), points().x
Struct Insert yvals%(), points().y
```

Notes

- Only the specified member is modified; all other members retain their values
- The insertion is a one-time copy - subsequent changes to the source array do not affect the structures
- Useful for updating structure data after processing with array-based functions

Using Structure Arrays with Drawing Commands

Several MMBasic drawing commands accept arrays of coordinates to draw multiple shapes efficiently. These commands now support **direct use of structure member arrays**, allowing you to pass structure data without first extracting it to separate arrays.

Supported Drawing Commands

The following commands support structure member array syntax:

Command	Array Parameters
`LINE PLOT`	y-coordinates array

`LINE GRAPH`	x(), y() coordinate arrays
`PIXEL`	x(), y() coordinate arrays
`LINE` (array mode)	x1(), y1(), x2(), y2() endpoint arrays
`POLYGON`	x(), y() vertex arrays
`CIRCLE`	x(), y(), r() center and radius arrays
`BOX`	x(), y(), w(), h() position and size arrays
`RBOX`	x(), y(), w(), h() position and size arrays
`TRIANGLE`	x1(), y1(), x2(), y2(), x3(), y3() vertex arrays

Syntax

Use the standard structure member array syntax with empty parentheses:

```
COMMAND structarray().member, ...
```

The empty parentheses "()" after the array name indicate "all elements", and the ".member" specifies which member to use from each structure element.

How It Works

When you pass a structure member array like "points().x", the drawing command automatically handles the non-contiguous memory layout. Structure members are spaced by the structure's total size (stride), not the standard 8-byte spacing of simple arrays.
For example, with:

```
Type Point
  x As FLOAT    ' offset 0
  y As FLOAT    ' offset 8
End Type
Dim points(100) As Point
```

The "x" values are 16 bytes apart (the size of Point), not 8 bytes. The drawing commands account for this stride automatically.

Examples

Drawing pixels from a Point array:

```
Type Point
  x As FLOAT
  y As FLOAT
End Type

Dim stars(50) As Point
Dim INTEGER i

' Initialize random star positions
For i = 0 To 50
  stars(i).x = Rnd * MM.HRES
  stars(i).y = Rnd * MM.VRES
Next i

' Draw all stars with a single command
PIXEL stars().x, stars().y, RGB(WHITE)
```

Drawing circles from a CircleDef array:

```
Type CircleDef
  x As FLOAT
  y As FLOAT
  r As FLOAT
End Type
```

```
Dim circles(10) As CircleDef

' Initialize circles
For i = 0 To 10
  circles(i).x = 50 + i * 50
  circles(i).y = 100
  circles(i).r = 20 + i * 2
Next i

' Draw all circles
CIRCLE circles().x, circles().y, circles().r,,, RGB(CYAN)
```

Drawing connected lines (LINE PLOT) from Y values:

```
Type DataPoint
  timestamp As INTEGER
  value As FLOAT
End Type

Dim readings(200) As DataPoint

' ... populate readings with sensor data ...

' Plot the values as a connected line graph
LINE PLOT readings().value, 1, RGB(GREEN)
```

Drawing boxes from a rectangle structure:

```
Type RectDef
  x As FLOAT
  y As FLOAT
  w As FLOAT
  h As FLOAT
End Type

Dim boxes(5) As RectDef

' Initialize boxes
For i = 0 To 5
  boxes(i).x = 10 + i * 100
  boxes(i).y = 50
  boxes(i).w = 80
  boxes(i).h = 60
Next i

' Draw all boxes
BOX boxes().x, boxes().y, boxes().w, boxes().h,, RGB(YELLOW)
```

Drawing triangles:

```
Type TriDef
  x1 As FLOAT : y1 As FLOAT
  x2 As FLOAT : y2 As FLOAT
  x3 As FLOAT : y3 As FLOAT
End Type

Dim triangles(4) As TriDef

' Initialize triangles
For i = 0 To 4
  triangles(i).x1 = 30 + i * 70
  triangles(i).y1 = 120
  triangles(i).x2 = 60 + i * 70
  triangles(i).y2 = 80
  triangles(i).x3 = 90 + i * 70
  triangles(i).y3 = 120
```

```
  Next i

  ' Draw all triangles
  TRIANGLE triangles().x1, triangles().y1, triangles().x2, triangles().y2, triangles().x3, triangles().y3
```

Drawing lines between two Point arrays:

```
  Type Point
    x As FLOAT
    y As FLOAT
  End Type

  Dim lineStart(10) As Point
  Dim lineEnd(10) As Point

  ' Initialize line endpoints
  For i = 0 To 10
    lineStart(i).x = 0
    lineStart(i).y = i * 20
    lineEnd(i).x = MM.HRES - 1
    lineEnd(i).y = i * 20
  Next i

  ' Draw all lines
  LINE lineStart().x, lineStart().y, lineEnd().x, lineEnd().y,, RGB(BLUE)
```

Mixing Structure and Simple Arrays

You can mix structure member arrays with simple arrays or scalar values in the same command:

```
  Type Point
    x As FLOAT
    y As FLOAT
  End Type

  Dim centers(10) As Point
  Dim FLOAT radii(10)

  ' Use structure for centers, simple array for radii
  For i = 0 To 10
    centers(i).x = 50 + i * 50
    centers(i).y = 100
    radii(i) = 15 + i * 3
  Next i

  CIRCLE centers().x, centers().y, radii(),,, RGB(MAGENTA)
```

When to Use STRUCT EXTRACT Instead

While direct structure member array syntax is convenient, there are cases where "STRUCT EXTRACT" is still preferred:

1. **Repeated use of the same data**: If you need to pass the same member data to multiple commands or in a loop, extracting once is more efficient.
2. **Performance-critical code**: Direct pointer arithmetic on simple arrays is slightly faster than strided access.

```
  ' Extract once, use multiple times
  Dim FLOAT xvals(100), yvals(100)
  Struct Extract points().x, xvals()
  Struct Extract points().y, yvals()

  ' Now use with any array command
  LINE PLOT xvals()
  MATH SCALE yvals(), 2, yvals()
```

Notes

- All coordinate arrays in a single command must have compatible dimensions
- The command uses the minimum array size if dimensions differ
- Integer and float structure members are both supported
- String members cannot be used with drawing commands

Using Structure Arrays with MATH Commands

Several MMBasic MATH commands now support **direct use of structure member arrays**, allowing you to perform mathematical operations on structure data without first extracting it to separate arrays. This is particularly useful for processing coordinate data, sensor readings, or any numerical data stored in structures.

Supported MATH Commands

The following MATH commands support structure member array syntax:

Element-wise Operations (3 arrays)

Command	Operation	Description
`MATH C_ADD`	a() + b() = c()	Element-wise addition
`MATH C_SUB`	a() - b() = c()	Element-wise subtraction
`MATH C_MUL`	a() * b() = c()	Element-wise multiplication
`MATH C_MULT`	a() * b() = c()	Same as C_MUL
`MATH C_DIV`	a() / b() = c()	Element-wise division

Scalar Transformations

Command	Operation	Description
`MATH SCALE`	a() * k = b()	Multiply all elements by scalar
`MATH ADD`	a() + k = b()	Add scalar to all elements
`MATH POWER`	a()^k = b()	Raise all elements to a power

Statistical Functions

Function	Description
`MATH(MAX(a()))`	Find maximum value
`MATH(MAX(a(), idx%))`	Find maximum and its index
`MATH(MIN(a()))`	Find minimum value
`MATH(MIN(a(), idx%))`	Find minimum and its index
`MATH(MEAN(a()))`	Calculate arithmetic mean
`MATH(SUM(a()))`	Calculate sum of all elements

Coordinate and Data Transformations

Command	Description
`MATH V_ROTATE`	Rotate 2D coordinate arrays around a point
`MATH WINDOW`	Normalize array values to a specified range

Syntax

Use the standard structure member array syntax with empty parentheses:

```
MATH command structarray().member, ...
result = MATH(function(structarray().member))
```

How It Works

When you pass a structure member array like "points().x", the MATH command automatically handles the non-contiguous memory layout. Structure members are spaced by the structure's total size (stride), not the standard 8-byte spacing of simple arrays.
For example, with:

```
Type Point
  x As FLOAT    ' offset 0
  y As FLOAT    ' offset 8
End Type
Dim points(100) As Point
```

The "x" values are 16 bytes apart (the size of Point), not 8 bytes. The MATH commands account for this stride automatically.

Examples

Element-wise Operations

Adding two structure member arrays:

```
Type Vector
  x As FLOAT
  y As FLOAT
  z As FLOAT
End Type

Dim a(10) As Vector
Dim b(10) As Vector
Dim result(10) As Vector

' Initialize vectors
For i = 0 To 10
  a(i).x = i : a(i).y = i * 2 : a(i).z = i * 3
  b(i).x = 1 : b(i).y = 2 : b(i).z = 3
Next i

' Add x components
MATH C_ADD a().x, b().x, result().x
' Add y components
MATH C_ADD a().y, b().y, result().y
' Add z components
MATH C_ADD a().z, b().z, result().z
```

Multiplying struct arrays element-wise:

```
Type DataPoint
  value As FLOAT
  weight As FLOAT
  weighted As FLOAT
End Type

Dim data(100) As DataPoint
' ... populate data().value and data().weight ...

' Calculate weighted values: weighted = value * weight
MATH C_MUL data().value, data().weight, data().weighted
```

Scalar Transformations

Scaling coordinates:

```
Type Point
  x As FLOAT
  y As FLOAT
End Type
```

```
Dim pts(50) As Point
' ... populate points ...

' Double all X coordinates
MATH SCALE pts().x, 2.0, pts().x

' Add offset to all Y coordinates
MATH ADD pts().y, 100, pts().y
```

Calculating squares:

```
Type DataRec
  value As FLOAT
  squared As FLOAT
End Type

Dim nums(100) As DataRec
' ... populate nums().value ...

' Square all values
MATH POWER nums().value, 2, nums().squared
```

Statistical Functions

Finding extremes in structure data:

```
Type SensorReading
  timestamp As INTEGER
  temperature As FLOAT
  humidity As FLOAT
End Type

Dim readings(144) As SensorReading
' ... populate with 24 hours of readings (every 10 min) ...

' Find temperature statistics
Dim maxTemp!, minTemp!, avgTemp!, sumTemp!
Dim maxIdx%, minIdx%

maxTemp! = MATH(MAX(readings().temperature, maxIdx%))
minTemp! = MATH(MIN(readings().temperature, minIdx%))
avgTemp! = MATH(MEAN(readings().temperature))
sumTemp! = MATH(SUM(readings().temperature))

Print "Max temperature: "; maxTemp!; " at reading "; maxIdx%
Print "Min temperature: "; minTemp!; " at reading "; minIdx%
Print "Average temperature: "; avgTemp!
```

Coordinate Rotation (V_ROTATE)

Rotating point coordinates around a center:

```
Type Point
  x As FLOAT
  y As FLOAT
End Type

Dim shape(20) As Point
' ... define a shape's vertices ...

' Rotate all points 45 degrees around point (100, 100)
MATH V_ROTATE 100, 100, 45, shape().x, shape().y, shape().x, shape().y

' Draw the rotated shape
```

```
  PIXEL shape().x, shape().y
```

Rotating with separate output arrays:

```
  Dim original(20) As Point
  Dim rotated(20) As Point

  ' Preserve original, store rotated in new array
  MATH V_ROTATE 0, 0, 90, original().x, original().y, rotated().x, rotated().y
```

Data Normalization (WINDOW)

Normalizing sensor data to screen coordinates:

```
  Type DataPoint
    raw_value As FLOAT
    screen_y As FLOAT
  End Type

  Dim samples(200) As DataPoint
  ' ... populate samples().raw_value with sensor data ...

  ' Normalize to screen Y range (0 to 480)
  MATH WINDOW samples().raw_value, 0, 480, samples().screen_y

  ' Now samples().screen_y contains Y positions for plotting
  LINE PLOT samples().screen_y
```

Getting min/max during normalization:

```
  Dim dataMin!, dataMax!
  MATH WINDOW samples().raw_value, 0, 100, samples().screen_y, dataMin!, dataMax!
  Print "Data range: "; dataMin!; " to "; dataMax!
```

Mixing Structure and Simple Arrays

You can mix structure member arrays with simple arrays in the same MATH command:

```
  Type Point
    x As FLOAT
    y As FLOAT
  End Type

  Dim pts(50) As Point
  Dim FLOAT offsets(50)
  Dim FLOAT results(50)

  ' Add structure member to simple array
  MATH C_ADD pts().x, offsets(), results()

  ' Or output to structure member from simple arrays
  MATH C_ADD offsets(), offsets(), pts().x
```

Integer Structure Members

Integer structure members work with MATH commands that support integers:

```
  Type IntData
    a As INTEGER
    b As INTEGER
    result As INTEGER
  End Type

  Dim idata(100) As IntData

  ' Integer element-wise operations
```

```
MATH C_ADD idata().a, idata().b, idata().result
MATH C_MUL idata().a, idata().b, idata().result

' Integer statistics
Dim maxVal%, minVal%, idx%
maxVal% = MATH(MAX(idata().a, idx%))
minVal% = MATH(MIN(idata().b))
```

MATH Commands NOT Supporting Structures

The following MATH commands do **not** support structure member arrays because they operate on fixed-size arrays that are conceptually structures themselves:

- **Quaternion operations**: Q_CREATE, Q_EULER, Q_INVERT, Q_MUL, Q_ROTATE, Q_VECTOR
- **Matrix operations**: M_INVERSE, M_MULT, M_TRANSPOSE, M_PRINT
- **FFT operations**: FFT, MAGNITUDE
- **Other**: CRC, INTERPOLATE, INPUT_CODE, V_CROSS, V_MULT, V_NORMALISE, V_PRINT

For these commands, use "STRUCT EXTRACT" if you need to work with structure data.

Performance Considerations

1. **Direct use is convenient**: For simple operations, passing structure members directly is clean and readable.
2. **Extract for repeated use**: If you need to perform many operations on the same member data, extracting once may be more efficient:

```
    Dim FLOAT xvals(100)
    Struct Extract points().x, xvals()
    ' Now use xvals() for multiple operations
```

3. **Memory efficiency**: Direct structure access avoids allocating temporary arrays.

Complete Example: Sensor Data Processing

```
' Define structure for sensor readings
Type SensorData
  timestamp As INTEGER
  raw_value As FLOAT
  calibrated As FLOAT
  normalized As FLOAT
End Type

Dim readings(200) As SensorData
Dim calibration_factor! = 1.023
Dim calibration_offset! = -2.5

' ... populate readings().timestamp and readings().raw_value from sensor ...

' Apply calibration: calibrated = raw * factor + offset
MATH SCALE readings().raw_value, calibration_factor!, readings().calibrated
MATH ADD readings().calibrated, calibration_offset!, readings().calibrated

' Get statistics on calibrated data
Dim maxVal!, minVal!, avgVal!, maxIdx%, minIdx%
maxVal! = MATH(MAX(readings().calibrated, maxIdx%))
minVal! = MATH(MIN(readings().calibrated, minIdx%))
avgVal! = MATH(MEAN(readings().calibrated))

Print "Calibrated data statistics:"
Print "  Max: "; maxVal!; " at sample "; maxIdx%
Print "  Min: "; minVal!; " at sample "; minIdx%
Print "  Avg: "; avgVal!

' Normalize to display range (0-200 pixels)
MATH WINDOW readings().calibrated, 0, 200, readings().normalized
```

```
' Plot the normalized data
CLS
LINE PLOT readings().normalized, 1, RGB(GREEN)
```

Using Structure Arrays with ARRAY Commands

In addition to MATH commands, several ARRAY commands also support structure member arrays directly.

Supported ARRAY Commands

Command	Description
`ARRAY SET`	Set all elements of a member array to a single value
`ARRAY ADD`	Add a scalar to all elements, storing in destination

Syntax

```
ARRAY SET value, structarray().member
ARRAY ADD structarray().member, scalar, destarray().member
```

Examples

Setting all members to a value:

```
Type DataPoint
  timestamp As INTEGER
  value As FLOAT
  status As INTEGER
End Type

Dim readings(100) As DataPoint

' Initialize all value members to zero
ARRAY SET 0, readings().value

' Set all status flags to 1
ARRAY SET 1, readings().status
```

Adding a constant to all elements:

```
Type Measurement
  raw As FLOAT
  calibrated As FLOAT
  offset As FLOAT
End Type

Dim data(50) As Measurement

' Apply offset: calibrated = raw + 5.5
ARRAY ADD data().raw, 5.5, data().calibrated

' Copy array (add 0)
ARRAY ADD data().raw, 0, data().calibrated
```

Mixing struct members with regular arrays:

```
Dim offsets!(50)
' ... populate offsets ...

' You can use struct source with regular destination (or vice versa)
' as long as array sizes match
```

ARRAY Commands NOT Supporting Structures

The following ARRAY commands do **not** support structure member arrays because they operate on multi-dimensional arrays:

- **ARRAY INSERT** - Requires 2D or higher dimensional arrays
- **ARRAY SLICE** - Requires 2D or higher dimensional arrays

Structure member arrays are inherently 1D (the array is of structures, not the member itself), so these commands cannot be used with structure members.

Clearing Structures

Use the "STRUCT CLEAR" command to reset all members of a structure to their default values (0 for numbers, empty string for strings):

```
Struct Clear variable
Struct Clear array()
```

Examples

Clear a single structure:

```
Dim p As Point
p.x = 100
p.y = 200

Struct Clear p
' p.x = 0, p.y = 0
```

Clear an entire array of structures:

```
Dim people(10) As Person
' ... populate array ...

Struct Clear people()
' All elements reset to defaults
```

Swapping Structures

Use the "STRUCT SWAP" command to exchange the contents of two structure variables:

```
Struct Swap var1, var2
```

Both variables must be of the same structure type. This is useful when implementing sorting algorithms or reordering records.

Examples

```
Dim a As Point, b As Point
a.x = 10 : a.y = 20
b.x = 30 : b.y = 40

Struct Swap a, b
' Now: a.x = 30, a.y = 40, b.x = 10, b.y = 20
```

Swapping array elements:

```
Dim people(5) As Person
' ... populate array ...

Struct Swap people(2), people(4)
' Elements 2 and 4 are exchanged
```

Printing Structures

Use the "STRUCT PRINT" command to display all members of a structure for debugging:

```
Struct Print variable
Struct Print array()
Struct Print array(index)
```

Forms

- "Struct Print variable" - Print a single structure variable
- "Struct Print array()" - Print all elements of a structure array
- "Struct Print array(n)" - Print a specific element of a structure array

Examples

Print a single structure:

```
Dim p As Person
p.name = "Alice"
p.age = 25
p.height = 1.65

Struct Print p
```

Output:

```
Person:
  .name = "Alice"
  .age = 25
  .height = 1.65
```

Print an array element:

```
Dim people(10) As Person
' ... populate array ...

Struct Print people(0)
```

Print entire array:

```
Struct Print people()
```

Output:

```
Person array (11 elements):
[0]:
  .name = "Alice"
  .age = 25
  .height = 1.65
[1]:
  .name = "Bob"
  .age = 30
  .height = 1.80
...
```

Notes

- Array members are printed as comma-separated values
- Strings are displayed with surrounding quotes
- Useful for debugging and inspecting structure contents

Searching Structure Arrays

Use the "STRUCT(FIND ...)" function to search a structure array for an element with a matching member value:

```
index = Struct(FIND array().membername, value [, start])
```

Parameters

- "FIND" - The subfunction name
- "array().membername" - The structure array with the member to search. The array name must include empty parentheses followed by a dot and the member name (e.g., "people().age")
- "value" - The value to search for (must match the member's type)
- "start" - Optional. The index to start searching from (default: first element)

Return Value

- Returns the index of the first matching element (starting from "start")
- Returns -1 if no match is found

Examples

Find by integer:

```
Type Person
  age As INTEGER
  name As STRING
End Type

Dim people(10) As Person
' ... populate array ...

idx = Struct(FIND people().age, 35)
If idx >= 0 Then
  Print "Found at index"; idx; ": "; people(idx).name
Else
  Print "Not found"
EndIf
```

Find by string:

```
idx = Struct(FIND people().name, "Alice")
If idx >= 0 Then
  Print "Alice is at index"; idx
EndIf
```

Find by float:

```
idx = Struct(FIND people().height, 1.75)
```

Iterate through all matches using start parameter:

```
' Find all people aged 30
idx = Struct(FIND people().age, 30)
Do While idx >= 0
  Print "Found at index"; idx; ": "; people(idx).name
  idx = Struct(FIND people().age, 30, idx + 1)
Loop
```

Notes

- Search is performed linearly from the start position
- Only the first match (from the start position) is returned
- Use the start parameter to iterate through multiple matches
- For strings, comparison is case-sensitive and exact
- Array members within structures cannot be searched

Getting Array Bounds

Use the standard "BOUND()" function to get the upper bound of a structure array dimension:

```
upperBound = Bound(array() [, dimension])
```

Parameters

- "array()" - The structure array (must include empty parentheses)
- "dimension" - Optional. Which dimension to query (0 returns OPTION BASE, 1+ for dimensions)

Return Value

- Returns the upper bound of the specified dimension

Examples

Basic usage (1D array):

```
Dim points(9) As Point
bound = Bound(points())    ' Returns 9
```

Multi-dimensional array:

```
Dim grid(4, 7) As Point
dim1 = Bound(grid(), 1)    ' Returns 4
dim2 = Bound(grid(), 2)    ' Returns 7
```

Use in loops:

```
Dim people(n) As Person
' ... populate array ...

For i = 0 To Bound(people())
  Print people(i).name
Next i
```

Notes

- The standard "BOUND()" function works for both regular arrays and structure arrays
- Dimension numbering: 0 returns OPTION BASE, 1 = first dimension, 2 = second, etc.
- If dimension is omitted, returns the bound of the first dimension

Getting Member Offset

Use the "STRUCT(OFFSET ...)" function to get the byte offset of a member within a structure type:

```
offset = Struct(OFFSET typename$, element$)
```

Parameters

- "OFFSET" - The subfunction name
- "typename$" - A string containing the structure type name
- "element$" - A string containing the member/element name

Return Value

- Returns the byte offset of the specified member from the start of the structure

Examples

Basic usage:

```
Type Point
  x As INTEGER
  y As INTEGER
End Type

offset_x = Struct(OFFSET "Point", "x")   ' Returns 0 (first member)
offset_y = Struct(OFFSET "Point", "y")   ' Returns 8 (after 8-byte integer)
```

```
  Print "Offset of x:"; offset_x
  Print "Offset of y:"; offset_y
```

Using with PEEK(VARADDR to access memory:

```
  Type Person
    age As INTEGER        ' offset 0
    height As FLOAT       ' offset 8
    name As STRING        ' offset 16
  End Type

  Dim p As Person
  p.age = 25
  p.height = 1.75
  p.name = "Alice"

  ' Get the base address of the structure variable
  baseAddr = Peek(VARADDR p)

  ' Calculate address of specific member
  ageAddr = baseAddr + Struct(OFFSET "Person", "age")
  heightAddr = baseAddr + Struct(OFFSET "Person", "height")

  Print "Age value via PEEK:"; Peek(INTEGER ageAddr)
```

Structure with arrays:

```
  Type Data
    header As INTEGER
    values(9) As FLOAT
  End Type

  ' Get offset of the values array start
  arrOffset = Struct(OFFSET "Data", "values")
  Print "Array starts at offset:"; arrOffset
```

Notes

- The offset is always in bytes from the start of the structure
- Useful for low-level memory access combined with "PEEK(VARADDR ...)"
- The type name and element name comparisons are case-insensitive
- Returns an error if the structure type or member is not found
- For nested structures, only top-level member offsets are returned

Getting Structure Size

Use the "STRUCT(SIZEOF ...)" function to get the size in bytes of a structure type:

```
  byteSize = Struct(SIZEOF typename$)
```

Parameters

- "SIZEOF" - The subfunction name
- "typename$" - A string containing the structure type name

Return Value

- Returns the total size in bytes of the specified structure type

Examples

Basic usage:

```
  Type Point
```

```
  x As INTEGER
  y As INTEGER
End Type

size = Struct(SIZEOF "Point")    ' Returns 16 (2 x 8-byte integers)
Print "Point size:"; size; "bytes"
```

Using with different types:

```
Type Person
  age As INTEGER        ' 8 bytes
  height As FLOAT       ' 8 bytes
  name As STRING        ' 256 bytes (default string)
End Type

size = Struct(SIZEOF "Person")   ' Returns 272
```

Dynamic type name:

```
typename$ = "Point"
size = Struct(SIZEOF typename$)
```

Use for memory calculations:

```
Dim data(99) As Record
totalBytes = 100 * Struct(SIZEOF "Record")
Print "Array uses"; totalBytes; "bytes"
```

Notes

- The type name comparison is case-insensitive
- Returns an error if the structure type is not defined
- Useful for calculating memory requirements or file sizes before STRUCT SAVE

Saving and Loading Structures

Structures can be saved to and loaded from files in binary format. This is useful for persisting data between program runs or exchanging data.

STRUCT SAVE

Writes structure data to an already-open file:

```
Struct Save #filenumber, variable
Struct Save #filenumber, array()
Struct Save #filenumber, array(index)
```

The file must be opened before using "STRUCT SAVE". You manage the file opening and closing.

Syntax options for arrays:

- "array()" - Saves the entire array
- "array(index)" - Saves only the element at the specified index

STRUCT LOAD

Reads structure data from an already-open file:

```
Struct Load #filenumber, variable
Struct Load #filenumber, array()
Struct Load #filenumber, array(index)
```

The file must be opened before using "STRUCT LOAD". The structure variable must already be declared.

Syntax options for arrays:

- "array()" - Loads the entire array
- "array(index)" - Loads only into the element at the specified index

Examples

Save and load a single structure:

```
Dim p As Point
p.x = 100
p.y = 200

' Save
Open "point.dat" For Output As #1
Struct Save #1, p
Close #1

' Load
Dim p2 As Point
Open "point.dat" For Input As #1
Struct Load #1, p2
Close #1

Print p2.x, p2.y   ' Output: 100    200
```

Save and load an array of structures:

```
Dim people(100) As Person
' ... populate array ...

' Save to file
Open "people.dat" For Output As #1
Struct Save #1, people()
Close #1

' Load from file
Dim loadedPeople(100) As Person
Open "people.dat" For Input As #1
Struct Load #1, loadedPeople()
Close #1
```

Save and load individual array elements:

```
Dim records(99) As Record
' ... populate records ...

' Save specific records to file
Open "selected.dat" For Output As #1
Struct Save #1, records(5)    ' Save element 5
Struct Save #1, records(10)   ' Append element 10
Struct Save #1, records(25)   ' Append element 25
Close #1

' Load records back (to different positions or variables)
Dim temp As Record
Open "selected.dat" For Input As #1
Struct Load #1, temp          ' Load first saved record
Print temp.id
Struct Load #1, records(50)   ' Load second saved record into element 50
Struct Load #1, records(51)   ' Load third saved record into element 51
Close #1
```

Notes

- Data is saved in binary format (not human-readable)
- The structure type must match when loading
- For whole array operations, array dimensions must match when loading
- Files should be opened in appropriate mode for the operation

- Multiple structures can be saved to the same file sequentially
- Error occurs if file is not open or is not a disk file
- **Array variables must use parentheses**: "array()" for whole array, "array(i)" for single element
- Using an array name without parentheses will cause an error

Structures in Subroutines and Functions

Passing Structures as Parameters

Structures are always passed **by reference**, meaning the subroutine or function can modify the original structure:

```
Sub subname(parametername As typename)
  ...
End Sub
```

Example - Read-only access:

```
Sub PrintPoint(pt As Point)
  Print "X:"; pt.x; " Y:"; pt.y
End Sub

Dim p As Point = (100, 200)
PrintPoint p
```

Example - Modifying the structure:

```
Sub DoublePoint(pt As Point)
  pt.x = pt.x * 2
  pt.y = pt.y * 2
End Sub

Dim p As Point = (10, 20)
DoublePoint p
Print p.x, p.y    ' Output: 20    40
```

Passing Array Elements

You can pass a single element of a structure array:

```
Dim points(10) As Point
points(5).x = 100
points(5).y = 200

PrintPoint points(5)
DoublePoint points(5)
```

Passing Structure Arrays

Use empty parentheses to pass an entire array of structures:

```
Sub ProcessPoints(pts() As Point)
  ' Access pts(0), pts(1), etc.
End Sub

Dim myPoints(10) As Point
ProcessPoints myPoints()
```

Example:

```
Sub SumAllPoints(pts() As Point, count%)
  Local total_x% = 0, total_y% = 0
  Local i%
  For i% = 0 To count% - 1
    total_x% = total_x% + pts(i%).x
    total_y% = total_y% + pts(i%).y
  Next i%
```

```
  Print "Total X:"; total_x%; " Total Y:"; total_y%
End Sub

Dim data(5) As Point
' ... initialize data ...
SumAllPoints data(), 6
```

Functions with Structure Parameters

```
Function Distance(pt As Point) As FLOAT
  Distance = Sqr(pt.x * pt.x + pt.y * pt.y)
End Function

Dim p As Point = (3, 4)
Print Distance(p)    ' Output: 5
```

Functions Returning Structures

Functions can return structure values using "As typename" in the function declaration:

```
Function functionname(parameters) As typename
  functionname.member1 = value1
  functionname.member2 = value2
End Function
```

The function name acts as a local structure variable that is returned when the function exits.

Example - Creating a Point:

```
Function MakePoint(x%, y%) As Point
  MakePoint.x = x%
  MakePoint.y = y%
End Function

Dim p As Point
p = MakePoint(100, 200)
Print p.x, p.y    ' Output: 100    200
```

Example - Structure with multiple types:

```
Function CreatePerson(n$, a%, h!) As Person
  CreatePerson.name = n$
  CreatePerson.age = a%
  CreatePerson.height = h!
End Function

Dim employee As Person
employee = CreatePerson("Alice", 30, 1.68)
Print employee.name; " is"; employee.age; " years old"
```

Example - Factory function:

```
Function Origin() As Point
  Origin.x = 0
  Origin.y = 0
End Function

Dim startPoint As Point
startPoint = Origin()
```

Local Structures

Use "LOCAL" to declare structures that exist only within a subroutine or function:

```
Sub Example
  Local pt As Point
  pt.x = 100
```

```
  pt.y = 200
  ' pt is automatically freed when the sub exits
End Sub
```

Local Structure Arrays

```
Sub ProcessData
  Local tempPoints(10) As Point
  ' Use tempPoints...
End Sub
```

Local Structures with Initialization

```
Sub Example
  Local pt As Point = (50, 75)
  Print pt.x, pt.y
End Sub
```

Static Structures

Use "STATIC" to declare structures that persist between calls to a subroutine or function. Unlike LOCAL structures which are destroyed when the subroutine exits, STATIC structures retain their values across multiple calls.

Basic Static Structure Usage

```
Sub Counter
  Static data As Point
  data.x = data.x + 1
  data.y = data.y + 10
  Print "Call count:"; data.x; " Total:"; data.y
End Sub

' Each call increments the values
Counter    ' Prints: Call count: 1  Total: 10
Counter    ' Prints: Call count: 2  Total: 20
Counter    ' Prints: Call count: 3  Total: 30
```

Static Structure Arrays

You can also create static arrays of structures:

```
Sub RecordReading(temp!, humidity!)
  Static readings(100) As SensorData
  Static idx% = 0

  readings(idx%).temperature = temp!
  readings(idx%).humidity = humidity!
  idx% = idx% + 1

  If idx% > 100 Then idx% = 0  ' Wrap around
End Sub
```

Static Structures with Initialization

Static structures can be initialized. The initialization only occurs on the first call:

```
Sub GetConfig() As Config
  Static cfg As Config = (100, 200, "default")
  GetConfig = cfg
End Sub
```

Function Returning Static Structure

A common pattern is using a static structure to accumulate or track state across function calls:

```
Type Statistics
  count As INTEGER
  sum As FLOAT
  min As FLOAT
  max As FLOAT
End Type

Function UpdateStats(value!) As Statistics
  Static stats As Statistics

  stats.count = stats.count + 1
  stats.sum = stats.sum + value!

  If stats.count = 1 Then
    stats.min = value!
    stats.max = value!
  Else
    If value! < stats.min Then stats.min = value!
    If value! > stats.max Then stats.max = value!
  End If

  UpdateStats = stats
End Function

' Usage
Dim s As Statistics
s = UpdateStats(10.5)
s = UpdateStats(5.2)
s = UpdateStats(15.8)
Print "Count:"; s.count; " Sum:"; s.sum; " Min:"; s.min; " Max:"; s.max
' Prints: Count: 3  Sum: 31.5  Min: 5.2  Max: 15.8
```

How Static Structures Work

When you declare a STATIC structure inside a subroutine or function:

1. **First call**: MMBasic creates a persistent storage location for the structure in global memory. The structure is initialized to zeros (or to the specified initialization values).
2. **Subsequent calls**: The same memory location is reused, preserving values from previous calls.
3. **Name scoping**: The static structure is only accessible by name within the declaring subroutine/function. It cannot conflict with global variables or static variables in other subroutines with the same name.

Notes on Static Structures

- Static structures are allocated once and persist for the lifetime of the program
- Memory for static structures is not released until the program ends or is edited
- Static structures inside functions can be returned as function results
- Initialization values (if provided) are only applied on the first call
- Use "STRUCT CLEAR" inside the subroutine if you need to reset a static structure

Nested Structures

Structures can contain other structures as members. The nested structure type must be defined before it is used in another structure.

Defining Nested Structures

```
' Define inner structure first
Type Point
  x As INTEGER
```

```
  y As INTEGER
End Type

' Now define structure that contains Point
Type Line
  start As Point    ' Nested structure member
  finish As Point   ' Another nested member
  color As INTEGER
End Type
```

Accessing Nested Members

Use chained dot notation to access nested members:

```
Dim myLine As Line
myLine.start.x = 10
myLine.start.y = 20
myLine.finish.x = 100
myLine.finish.y = 200
myLine.color = 255

Print myLine.start.x      ' Prints 10
Print myLine.finish.y     ' Prints 200
```

Multiple Levels of Nesting

Structures can be nested to multiple levels:

```
Type Point
  x As INTEGER
  y As INTEGER
End Type

Type Box
  topLeft As Point
  bottomRight As Point
End Type

Type Scene
  boundary As Box    ' Box contains Points - 3 levels
  name As STRING LENGTH 20
End Type

Dim myScene As Scene
myScene.boundary.topLeft.x = 0
myScene.boundary.bottomRight.x = 640
myScene.name = "MainScene"

Print myScene.boundary.topLeft.x     ' Prints 0
```

Arrays with Nested Structures

Arrays of structures containing nested structures work as expected:

```
Dim lines(10) As Line
lines(0).start.x = 1
lines(0).start.y = 2
lines(5).finish.x = 100
```

Arrays of Nested Structure Members

Structure members can be arrays of nested structures:

```
Type Point
  x As INTEGER
  y As INTEGER
```

```
  End Type

  Type Polygon
    vertices(9) As Point   ' Array of 10 nested Point structures
    color As INTEGER
  End Type

  Dim shape As Polygon
  shape.vertices(0).x = 0
  shape.vertices(0).y = 0
  shape.vertices(1).x = 100
  shape.vertices(1).y = 50
  shape.color = 255
```

Complex Nesting Example

The most complex supported syntax combines all features:

```
  Type InnerType
    values(9) As FLOAT    ' Array of floats
  End Type

  Type OuterType
    items(5) As InnerType  ' Array of nested structs, each with array
  End Type

  Dim data(3) As OuterType  ' Array of outer structs

  ' Access: array(i).array_member(j).array_member(k)
  data(2).items(1).values(4) = 3.14159
  Print data(2).items(1).values(4)   ' Prints 3.14159
```

This demonstrates:

- "data(2)" - Element 2 of the outer array
- ".items(1)" - Element 1 of the nested struct array member
- ".values(4)" - Element 4 of the innermost float array

LIST TYPE with Nested Structures

The "LIST TYPE" command shows nested structure types by name:

```
  >LIST TYPE Line
  TYPE LINE
      START AS Point  ' offset=0
      FINISH AS Point  ' offset=16
      COLOR AS INTEGER  ' offset=32
  END TYPE  ' size=40 bytes
```

Limitations

- The nested type must be defined BEFORE the containing type
- No self-referential structures (a type cannot contain itself)
- Maximum nesting depth: 8 levels (configurable via "MAX_STRUCT_NEST_DEPTH")

Multiple Structure Types

You can define multiple different structure types in your program:

```
  Type Point
    x As INTEGER
    y As INTEGER
  End Type

  Type Rectangle
```

```
    left As INTEGER
    top As INTEGER
    width As INTEGER
    height As INTEGER
  End Type

  Type Line
    start As Point    ' Nested structure
    finish As Point
  End Type

  Dim p As Point
  Dim r As Rectangle
  Dim ln As Line
```

Best Practices

1. **Define types at the start of your program** - Place all TYPE definitions near the beginning, before any executable code.
2. **Define nested types before containing types** - Inner structures must be defined first.
3. **Use meaningful names** - Choose descriptive names for both types and members:

```
    Type SensorReading
      timestamp As INTEGER
      temperature As FLOAT
      humidity As FLOAT
    End Type
```

4. **Initialize structures** - Always initialize structure members before use, either with the initialization syntax or by assignment.
5. **Use LOCAL for temporary structures** - When a structure is only needed within a subroutine, declare it as LOCAL to automatically free memory.
6. **Pass structures to subroutines** - Rather than passing many individual parameters, group related data into a structure.

Limitations

- Maximum structure types: 32
- Maximum members per structure: 16
- Member names follow standard MMBasic variable naming rules
- Maximum nesting depth: 8 levels (configurable via "MAX_STRUCT_NEST_DEPTH")

Error Messages

Error	Cause
"Structure type not found"	The structure type name in DIM AS doesn't match any...
"Unknown structure member"	Accessing a member name that doesn't exist in the s...
"Structure type mismatch"	Trying to copy or pass structures of different type...
"Expected a structure variable"	A subroutine expected a structure but received some...
"Source must be a structure variab...	STRUCT COPY source is not a structure
"Destination must be a structure v...	STRUCT COPY destination is not a structure
"Not enough initialisation values"	Initialization list has fewer values than required
"Expected '(' for structure initia...	Missing opening parenthesis in initialization
"Expected a structure array"	STRUCT.FIND requires a structure array, not a singl...
"Member not found in structure"	STRUCT.FIND or STRUCT SORT member name doesn't exis...
"Cannot search array members"	STRUCT.FIND cannot search members that are arrays
"Type mismatch: expected numeric v...	STRUCT.FIND search value type doesn't match member ...

"Type mismatch: expected string va...	STRUCT.FIND search value is not a string but member...
"Expected #filenumber"	STRUCT SAVE/LOAD requires a file number starting wi...
"Invalid file number"	File number is outside valid range
"File not open"	STRUCT SAVE/LOAD file is not open
"Not a disk file"	STRUCT SAVE/LOAD requires a disk file, not serial p...
"Cannot save/load a structure memb...	STRUCT SAVE/LOAD requires whole structure, not memb...
"Array variable requires () or (in...	STRUCT SAVE/LOAD array must use parentheses

Complete Example

```
' Define structure types
Type Point
  x As INTEGER
  y As INTEGER
End Type

Type Line
  name As STRING LENGTH 20
  startX As INTEGER
  startY As INTEGER
  endX As INTEGER
  endY As INTEGER
End Type

' Declare variables
Dim origin As Point = (0, 0)
Dim cursor As Point
Dim lines(10) As Line

' Initialize cursor
cursor.x = 100
cursor.y = 100

' Create some lines
lines(0).name = "Horizontal"
lines(0).startX = 0 : lines(0).startY = 50
lines(0).endX = 100 : lines(0).endY = 50

lines(1).name = "Vertical"
lines(1).startX = 50 : lines(1).startY = 0
lines(1).endX = 50 : lines(1).endY = 100

' Subroutine to calculate line length
Function LineLength(ln As Line) As FLOAT
  Local dx% = ln.endX - ln.startX
  Local dy% = ln.endY - ln.startY
  LineLength = Sqr(dx% * dx% + dy% * dy%)
End Function

' Print line information
Sub PrintLine(ln As Line)
  Print ln.name; ": ("; ln.startX; ","; ln.startY; ") to ("; ln.endX; ","; ln.endY; ")"
  Print "  Length: "; LineLength(ln)
End Sub

' Display all lines
For i% = 0 To 1
  PrintLine lines(i%)
Next i%
```

Output:

```
  Horizontal: (0,50) to (100,50)
    Length: 100
  Vertical: (50,0) to (50,100)
    Length: 100
```

Quick Reference

Commands

Command	Description
`Type...End Type`	Define a new structure type
`Dim var As typename`	Declare a structure variable
`Dim arr(n) As typename`	Declare an array of structures
`dst = src`	Copy structure using assignment
`arr(i) = arr(j)`	Copy array elements using assignment
`Struct Copy src To dst`	Copy structure contents
`Struct Copy src() To dst()`	Copy entire structure array
`Struct Sort arr().member [,flags]`	Sort array by member field
`Struct Extract arr().member, dest()`	Extract member values to simple array
`Struct Insert src(), arr().member`	Insert array values into structure member
`Struct Clear var`	Reset all members to defaults
`Struct Clear arr()`	Reset all array elements to defaults
`Struct Swap var1, var2`	Exchange contents of two structures
`Struct Print var`	Print structure contents for debugging
`Struct Print arr()`	Print all array elements
`Struct Save #n, var`	Save structure to open file
`Struct Save #n, arr()`	Save entire structure array to open file
`Struct Save #n, arr(i)`	Save single array element to open file
`Struct Load #n, var`	Load structure from open file
`Struct Load #n, arr()`	Load entire structure array from open file
`Struct Load #n, arr(i)`	Load single array element from open file

MATH Commands Supporting Structures

Command	Description
`MATH C_ADD a().m, b().m, c().m`	Element-wise addition
`MATH C_SUB a().m, b().m, c().m`	Element-wise subtraction
`MATH C_MUL a().m, b().m, c().m`	Element-wise multiplication
`MATH C_DIV a().m, b().m, c().m`	Element-wise division
`MATH SCALE a().m, k, b().m`	Multiply all elements by scalar k
`MATH ADD a().m, k, b().m`	Add scalar k to all elements
`MATH POWER a().m, k, b().m`	Raise all elements to power k
`MATH V_ROTATE cx,cy,ang,x().m,y().m,ox().m,oy().m`	Rotate coordinates
`MATH WINDOW a().m, lo, hi, b().m`	Normalize to range [lo, hi]
`MATH(MAX(a().m [,idx%]))`	Maximum value (and optional index)
`MATH(MIN(a().m [,idx%]))`	Minimum value (and optional index)
`MATH(MEAN(a().m))`	Arithmetic mean
`MATH(SUM(a().m))`	Sum of all elements

ARRAY Commands Supporting Structures

Command	Description
`ARRAY SET value, a().m`	Set all member elements to value
`ARRAY ADD a().m, k, b().m`	Add scalar k to all elements

Functions

Function	Description
`Struct(FIND arr().member, value [...	Find element with matching member value, returns in...
`Struct(OFFSET typename$, element$...	Get byte offset of member within structure type
`Struct(SIZEOF typename$)`	Get size in bytes of a structure type
`Bound(arr() [,dimension])`	Get upper bound of structure array dimension (stand...

Member Types

Type	Size	Alignment	Description
`INTEGER`	8 bytes	8-byte	64-bit signed integer
`FLOAT`	8 bytes	8-byte	64-bit floating point
`STRING`	256 bytes	None	Default string (255 char...
`STRING LENGTH n`	n+1 bytes	None	String with specified ma...
Nested struct	Varies	8-byte	Size of the nested struc...

This appendix provides technical details about how structures are implemented internally in MMBasic. This information is primarily useful for developers working on MMBasic itself or advanced users who need to understand the low-level behavior.

Overview

MMBasic structures (user-defined types) are implemented using a **single vartbl entry per structure variable**, regardless of how many members the structure contains. Structure members do NOT create individual entries in "g_vartbl". Instead, member metadata is stored in a separate **structure type definition table** ("g_structtbl"), and member access is resolved at runtime by calculating byte offsets into a contiguous memory block.

Architecture

Data Structures

1. Structure Type Definition (`s_structdef`)

```
typedef struct s_structdef {
    unsigned char name[MAXVARLEN];              // Structure type name (e.g., "POINT")
    int num_members;                            // Number of members in this type
    struct s_structmember members[MAX_STRUCT_MEMBERS];  // Member definitions
    int total_size;                             // Total size in bytes (with alignment)
} structdef_val;
```

2. Structure Member Definition (`s_structmember`)

```
typedef struct s_structmember {
    unsigned char name[MAXVARLEN];  // Member name (e.g., "X", "Y")
    unsigned char type;             // T_NBR, T_STR, T_INT, or T_STRUCT (nested)
    unsigned char size;             // String max length, or nested struct type index
    int offset;                     // Byte offset within structure
    short dims[MAXDIM];             // Array dimensions (0 = not an array)
} structmember_val;
```

3. Variable Table Entry (`s_vartbl`)

Structure variables use the standard variable table entry:

```
  typedef struct s_vartbl {
      unsigned char name[MAXVARLEN];  // Variable name (e.g., "PT", "POINTS")
      unsigned char type;             // T_STRUCT | T_IMPLIED (and possibly T_PTR)
      unsigned char level;            // Scope level (0 = global, >0 = local)
      unsigned char size;             // ** STRUCT TYPE INDEX ** (not string size)
      unsigned char namelen;          // Flags (NAMELEN_EXPLICIT, NAMELEN_STATIC)
      int/short dims[MAXDIM];         // Array dimensions (for struct arrays)
      union u_val {
          MMFLOAT f;
          long long int i;
          MMFLOAT *fa;
          long long int *ia;
          unsigned char *s;           // ** POINTER TO STRUCT DATA BLOCK **
      } val;
  } vartbl_val;
```

Global Tables

Table	Purpose	Size
`g_structtbl[MAX_STRUC...	Array of pointers to s...	32 pointers
`g_structcnt`	Count of defined struc...	int
`g_vartbl[]`	Variable table (struct...	MAXVARS entries

Configuration Constants

- "MAX_STRUCT_TYPES" = 32 (maximum distinct TYPE definitions)
- "MAX_STRUCT_MEMBERS" = 16 (maximum members per structure)
- "MAX_STRUCT_NEST_DEPTH" = 8 (maximum nesting depth)

Memory Layout

Single Structure Variable

For a structure definition:

```
  TYPE point
      x AS FLOAT
      y AS FLOAT
  END TYPE

  DIM pt AS point
```

Creates:

1. **One entry in "g_structtbl"** (type definition, allocated once)
2. **One entry in "g_vartbl"** for variable "pt"
3. **One contiguous memory block** (16 bytes) pointed to by "g_vartbl[idx].val.s"

Memory layout for "pt":

```
  Offset  Content         Size
  0       x (FLOAT)       8 bytes
  8       y (FLOAT)       8 bytes
  ---     Total           16 bytes
```

Structure Array

```
  DIM points(100) AS point
```

Creates:

1. **One entry in "g_vartbl"** for array "points"
2. **One contiguous memory block** (16 × 101 = 1616 bytes, accounting for OPTION BASE)

Memory layout for "points(0)" through "points(100)":

```
Offset    Content
0         points(0).x, points(0).y     (16 bytes)
16        points(1).x, points(1).y     (16 bytes)
32        points(2).x, points(2).y     (16 bytes)
...
1600      points(100).x, points(100).y (16 bytes)
```

How vartbl Fields Are Used for Structures

Field	Usage for Structures	
`name[]`	Variable name (e.g., "PT", "POINTS")	
`type`	`T_STRUCT \	T_IMP
`level`	Scope level (0 = global)	
`size`	**Structure type index** into `g_structtbl[]`	
`namelen`	Flags: `NAMELEN_STATIC` for static variables	
`dims[]`	Array dimensions (0 for simple struct, >0 for arrays)	
`val.s`	**Pointer to allocated struct data**	

Key Insight: `size` Field Repurposing

For regular variables, "size" holds string length. For structures:
- "size" holds the **index into "g_structtbl"** that defines this struct's type
- Retrieved via: "int struct_type = (int)g_vartbl[idx].size;"
- Then access type definition: "g_structtbl[struct_type]"

Structure Member Access Resolution

When code accesses "pt.x" or "points(5).y":

Step 1: Parse Variable Name

- Detect dot in name
- Split into base name ("pt") and member path ("x")

Step 2: Find Base Variable

- Hash-based lookup in local then global variable space
- Verify "type & T_STRUCT"
- Return struct type index from "size" field

Step 3: Resolve Member Path

- Look up member in "g_structtbl[type_idx]->members[]"
- Calculate byte offset from "member.offset"
- Handle nested structs recursively
- Handle array indexing for member arrays

Step 4: Return Pointer

- Final pointer = "base_ptr + total_offset + array_offset"
- Set "g_StructMemberType" for type checking
- Set "g_StructMemberOffset" and "g_StructMemberSize" for STRUCT operations

MMBasic Structures User Manual

STATIC Structure Variables

STATIC structures create TWO vartbl entries:

1. **Global entry** with mangled name (""funcname\x1evarname"")
- Holds actual struct data
- Persists across function calls
- The "\x1e" (ASCII Record Separator) character separates the function name from the variable name
2. **Local entry** with original name
- "val.s" points to global entry's data
- Destroyed when function exits

The "<RS>" character in the name prevents conflicts with structure member syntax (which uses ".").

Example Memory Trace

```
TYPE room
    temp AS FLOAT
    name AS STRING LENGTH 20
END TYPE

FUNCTION thisreading() AS room
    STATIC n%           ' Creates global "thisreading<RS>n"
    STATIC data AS room ' Creates global "thisreading<RS>data"
    n% = n% + 1
    data.temp = 22.5
    data.name = "Kitchen"
    thisreading = data
END FUNCTION
```

g_structtbl[0] (room type):

```
name = "ROOM"
num_members = 2
members[0] = { name="TEMP", type=T_NBR, offset=0, size=8 }
members[1] = { name="NAME", type=T_STR, offset=8, size=20 }
total_size = 32 (8 + 21, rounded up for alignment)
```

g_vartbl entries when function runs:

Index	Name	Type	Size	val.s	
G1	"THISREADING\x1EN"	T_INT	8	(integer value)	
G2	"THISREADING\x1EDATA"	T_STRUCT	0	-> 32-byte block	
L1	"N"	T_INT\	T_PTR	8	-> G1's
L2	"DATA"	T_STRUCT\	T_PTR	0	-> G2's

Summary: Key Design Decisions

1. **One vartbl entry per variable** - Members are NOT individual variables
2. **Type definitions separate from instances** - "g_structtbl" holds metadata
3. **Contiguous memory blocks** - Efficient for arrays and memory operations
4. **Offset-based member access** - Calculated at runtime, no pointer chasing
5. **Alignment enforced** - 8-byte boundary for numeric types
6. **Nested structures supported** - Via recursive type references
7. **STATIC uses mangled names** - With "\x1e" separator to avoid dot conflicts