rfoMM - Comprehensive Documentation (v0.20)

Version: 0.20 **Date:** 2025-05-06 **Library:** rfoMM.bas

1. Introduction

1.1. What is rfoMM?

rfoMM (rfo Memory Manager) is the foundational memory management library for the **Project Nebula Runtime Environment**, and this is the version 0.20. Its primary purpose is to provide a safe, reliable, and organized way to create, track, and manage RFO Basic's core data structures (Bundles, Lists) within applications.

It introduces a string-based **Memory Address** system (MM_ADDRESS$) as a robust alternative to RFO Basic's potentially ambiguous numeric pointers.
While designed as a core pillar for Project Nebula, rfoMM.bas can also be included and used as a standalone library in any RFO Basic project to improve data structure management.

1.2. Why Use rfoMM? The Problem Solved

RFO Basic provides powerful data structures like Bundles and Lists, accessed via numeric pointers (e.g., 1, 2, 3). However, a significant challenge arises in complex projects: **RFO Basic maintains separate internal counters for each data structure type.** This means you can simultaneously have a Bundle identified by pointer 1, a String List identified by pointer 1, and a Numeric List identified by pointer 1.

Knowing only the numeric pointer (1) tells you nothing definitive about the *type* of structure it represents. This inherent ambiguity can lead to:

- **Difficult Debugging:** Errors where operations intended for one type of structure are accidentally performed on another type that happens to share the same numeric ID.
- **Complex Tracking:** Requiring the developer to manually track not only the numeric pointer but also the intended type for every single structure instance.
- **Resource Issues:** Difficulty in reliably managing the lifecycle (clearing, potential reuse) of structures when their identifiers are ambiguous.

rfoMM solves these critical problems by providing an abstraction layer:

- **Safety:** It assigns a unique, unambiguous string **MM_ADDRESS$** to every structure it manages, inherently linking the identifier to both the specific instance and its data type.
- **Clarity:** Your code interacts with meaningful addresses instead of potentially conflicting numeric pointers.
- **Organization:** It provides functions to manage the lifecycle of these structures, including controlled recycling of underlying RFO Basic resources.

1.3. Target Audience

This library is primarily aimed at:

- RFO Basic developers working on medium-to-large scale applications where managing numerous data structures becomes complex.
- Developers intending to use or build upon the Project Nebula Runtime Environment.

2. Core Concepts

Understanding these concepts is key to using rfoMM effectively.

2.1. The MM_ADDRESS$ System

The heart of rfoMM is the **Memory Address** (MM_ADDRESS$). Instead of returning raw numeric pointers like 1 or 2, rfoMM's core creation function (MM_CREATE$) returns a string identifier, typically in the format "rfoMM(x)" where x is an internal counter tracking total managed allocations.

- **Uniqueness:** Each MM_ADDRESS$ is unique across *all* data structures managed by rfoMM, regardless of their underlying RFO Basic type or numeric pointer. "rfoMM(1)" will always refer to the exact same structure instance until it is deleted via MM_DELETE.
- **Unified Namespace:** It eliminates the ambiguity of RFO Basic's parallel numeric pointers. There is only one "rfoMM(1)", "rfoMM(2)", etc., simplifying tracking.
- **Type Association:** The rfoMM system internally tracks the specific RFO Basic data type (Bundle, String List, etc.) associated with each MM_ADDRESS$ using internal tracking bundles (MEMORY_ADDRESSES and MEMORY_TYPES). Functions like MM_TYPE$ retrieve this information reliably.
- **Abstraction:** You primarily interact with the MM_ADDRESS$. When you need the underlying numeric pointer (e.g., to use standard RFO Basic commands like BUNDLE.PUT), you use the MM_POINTER(MM_ADDRESS$) function to retrieve it safely.

2.2. Supported Data Types

In version 0.20, rfoMM can manage the following RFO Basic data structure types, identified by these type strings used in MM_CREATE$, MM_LOAD$, etc.:

- "B": Bundle
- "SL": String List
- "NL": Numeric List
- "C": Class Bundle (Intended for use with the rfOOP pillar, treated as a Bundle by rfoMM)
- "O": Object Bundle (Intended for use with the rfOOP pillar, treated as a Bundle by rfoMM)

2.3. Lifecycle Management: Recycling vs. Deletion

A common source of issues in languages without automatic garbage collection is managing

when resources are no longer needed. RFO Basic does not automatically destroy structures or reclaim their numeric pointers when they go out of scope.
rfoMM addresses this through **managed recycling**:

1. **MM_DELETE(MM_ADDRESS$) is Called:** You signal that the structure identified by MM_ADDRESS$ is no longer needed.
2. **Tracking Removed:** rfoMM removes the MM_ADDRESS$ from its internal tracking bundles (MEMORY_ADDRESSES, MEMORY_TYPES).
3. **Structure Cleared:** rfoMM clears the underlying RFO Basic structure using BUNDLE.CLEAR or LIST.CLEAR. This prevents lingering data.
4. **Pointer Recycled:** The raw numeric pointer associated with the cleared structure is added to an internal "open" list specific to its type (e.g., OPEN_BUNDLES).
5. **Reuse:** When MM_CREATE$ needs a new structure of that type, it first checks the corresponding "open" list (via the internal MM_OPEN/MM_RECYCLE functions). If a recycled pointer is available, rfoMM reuses that pointer for the new structure instead of asking RFO Basic to allocate a brand new one.

This recycling mechanism offers:

- **Safety:** Prevents accidental use of numeric pointers associated with structures that *should* have been deleted but were merely abandoned. Using MM_POINTER on a deleted MM_ADDRESS$ will return 0.
- **Efficiency:** Can reduce the overhead of RFO Basic constantly creating new structure identifiers, potentially reusing existing cleared "slots".

It's important to understand that MM_DELETE doesn't free memory in the C++ sense, but rather manages the lifecycle and safe reuse of RFO Basic's structure pointers within the rfoMM system.

3. Setup and Usage

3.1. Standalone Usage

To use rfoMM in a standalone RFO Basic project:

1. Ensure the library file (e.g., rfoMM.bas) is accessible to your main script (e.g., in the same directory or a known path).
2. Include the library at the **top** of your main script using the INCLUDE statement:
```
INCLUDE rfoMM.bas
```

3.2. Project Nebula Context

Within the planned Project Nebula environment (packaged as an APK), rfoMM.bas is intended to be a core "pillar" residing in a standard Libraries/ directory. The Nebula runtime environment itself would typically handle the inclusion or loading of such core libraries automatically for application scripts running within Nebula.

3.3. Initialization

This is the most critical step! Before calling *any* other rfoMM function, you **must** initialize the memory management environment once, typically near the very beginning of your script (after INCLUDE).

```
MM_INIT()
```

Failure to call MM_INIT() first will result in errors when other rfoMM functions attempt to access the internal tracking structures.

4. API Reference

This section details the functions provided by rfoMM v0.20.

4.1. Initialization

- **MM_INIT()**
  - **Parameters:** None.
  - **Returns:** None.
  - **Description:** Initializes the internal bundles and lists required by rfoMM to track memory addresses, types, and recycled pointers. Creates the main rfoMM management object (accessible via BUNDLE.GET 1, "rfoMM", ... though direct access is discouraged).
  - **Example:**
    ```
    INCLUDE rfoMM.bas
    MM_INIT()
    PRINT "rfoMM Initialized."
    % Ready to use other MM functions...
    ```
  - **Notes:** MUST be called once at the start of your application before any other rfoMM function. Calling it multiple times may lead to unexpected behavior or errors.

4.2. User Functions

These are the primary functions intended for developers to interact with the memory manager.

- **MM_CREATE$(type$)**
  - **Parameters:**
    - type$ (String): The type of structure to create. Supported values: "B", "SL", "NL", "C", "O".
  - **Returns:** MM_ADDRESS$ (String): The unique memory address for the newly created structure (e.g., "rfoMM(5)"), or potentially an error indicator/empty string if the type is invalid (though current implementation might END).
  - **Description:** Creates a new RFO Basic data structure of the specified type$, brings it under rfoMM management, and returns its unique MM_ADDRESS$. It checks for

available recycled pointers of that type before creating a new one.

  - **Example:**
    ```
    DIM bundleAddr$, listAddr$
    bundleAddr$ = MM_CREATE$("B")
    listAddr$ = MM_CREATE$("SL")
    PRINT "Created Bundle: " + bundleAddr$
    PRINT "Created String List: " + listAddr$
    ```
  - **Notes:** This is the preferred function for creating new managed structures when you want to work with the MM_ADDRESS$ system directly.

- **MM_CREATE(type$)**
  - **Parameters:**
    - type$ (String): The type of structure to create. Supported values: "B", "SL", "NL", "C", "O".
  - **Returns:** Pointer (Numeric): The raw RFO Basic numeric pointer for the newly created structure, or potentially 0 if creation fails.
  - **Description:** A wrapper function that calls MM_CREATE$(type$) internally and then uses MM_POINTER to return the underlying numeric pointer of the newly created structure. The structure is still managed by rfoMM.
  - **Example:**
    ```
    DIM bundlePtr
    bundlePtr = MM_CREATE("B")
    IF bundlePtr > 0 THEN PRINT "Created Bundle Pointer: " +
    bundlePtr
    ```
  - **Notes:** Useful if you immediately need the numeric pointer after creation for use with RFO Basic commands and prefer not to make a separate MM_POINTER call. However, using MM_CREATE$ and storing the MM_ADDRESS$ is generally recommended for consistency.

- **MM_DELETE(MM_ADDRESS$)**
  - **Parameters:**
    - MM_ADDRESS$ (String): The address of the managed structure to delete/recycle.
  - **Returns:** None.
  - **Description:** Removes the structure associated with MM_ADDRESS$ from rfoMM's active management. It clears the underlying RFO Basic structure (BUNDLE.CLEAR or LIST.CLEAR), removes the address from internal tracking bundles, and adds the numeric pointer to the appropriate internal recycling list.
  - **Example:**
    ```
    DIM addr$
    addr$ = MM_CREATE$("NL")
    PRINT "Exists before delete: " + MM_EXISTS(addr$) % Prints 1
    MM_DELETE(addr$)
    PRINT "Exists after delete: " + MM_EXISTS(addr$) % Prints 0
    ```
  - **Notes:** After calling MM_DELETE, the MM_ADDRESS$ is invalid. Attempting to

use it with functions like MM_POINTER or MM_TYPE$ will return 0 or "UNDEFINED". Does not return an error if the address doesn't exist (prints message).

- **MM_LOAD$(struct_id, type$)**
  - **Parameters:**
    - struct_id (Numeric): The RFO Basic numeric pointer (>0) of an *existing*, *unmanaged* structure.
    - type$ (String): The correct RFO Basic type of the structure being loaded ("B", "SL", "NL", "C", "O").
  - **Returns:** MM_ADDRESS$ (String): The newly assigned unique memory address for the loaded structure, or "" if struct_id is invalid (<1).
  - **Description:** Brings an existing RFO Basic structure (that was created *outside* of rfoMM, e.g., via a direct BUNDLE.CREATE or returned by an RFO command) under rfoMM management. Assigns it a new MM_ADDRESS$ and adds it to internal tracking.
  - **Example:**
    ```
    DIM rawListPtr, listAddr$
    LIST.CREATE S, rawListPtr % Create list outside rfoMM
    LIST.ADD rawListPtr, "item1", "item2"
    PRINT "Raw List Pointer: " + rawListPtr
    listAddr$ = MM_LOAD$(rawListPtr, "SL") % Bring under
    management
    PRINT "Managed Address: " + listAddr$
    IF MM_EXISTS(listAddr$) THEN PRINT "Successfully loaded."
    % Can now manage rawListPtr via listAddr$
    MM_DELETE(listAddr$) % Deletes it from MM and clears it
    ```
  - **Notes:** Crucial for integrating structures created by external means. Ensure the provided type$ matches the actual type of struct_id. Loading an already managed pointer might lead to issues (not explicitly prevented).

- **MM_POINTER(MM_ADDRESS$)**
  - **Parameters:**
    - MM_ADDRESS$ (String): The address of a managed structure.
  - **Returns:** Pointer (Numeric): The underlying RFO Basic numeric pointer associated with the address, or 0 if the address is invalid or not found.
  - **Description:** Retrieves the raw numeric pointer needed to interact with the managed structure using standard RFO Basic commands (e.g., BUNDLE.PUT, LIST.ADD, GR.GET.TEXT).
  - **Example:**
    ```
    DIM bundleAddr$, bundlePtr
    bundleAddr$ = MM_CREATE$("B")
    bundlePtr = MM_POINTER(bundleAddr$)
    IF bundlePtr > 0 THEN
      BUNDLE.PUT bundlePtr, "myKey", "myValue"
      BUNDLE.GET bundlePtr, "myKey", value$
      PRINT "Value from bundle: " + value$
    ```

```
ELSE
  PRINT "Invalid Address: " + bundleAddr$
ENDIF
MM_DELETE(bundleAddr$)
```

  - **Notes:** This is the bridge between the rfoMM abstraction and RFO Basic's direct structure manipulation commands. Always check if the returned pointer is > 0 before using it.

- **MM_TYPE$(MM_ADDRESS$)**
  - **Parameters:**
    - MM_ADDRESS$ (String): The address of a managed structure.
  - **Returns:** TypeString$ (String): The type string ("B", "SL", "NL", "C", "O") associated with the address, or "UNDEFINED" if the address is invalid or not found.
  - **Description:** Returns the data type of the structure managed at the given address.
  - **Example:**

```
DIM addr1$, addr2$, type1$, type2$
addr1$ = MM_CREATE$("B")
addr2$ = MM_CREATE$("NL")
type1$ = MM_TYPE$(addr1$)
type2$ = MM_TYPE$(addr2$)
PRINT addr1$ + " is type: " + type1$ % Prints B
PRINT addr2$ + " is type: " + type2$ % Prints NL
MM_DELETE(addr1$)
MM_DELETE(addr2$)
```

- **MM_EXISTS(MM_ADDRESS$)**
  - **Parameters:**
    - MM_ADDRESS$ (String): The address to check.
  - **Returns:** Exists (Numeric): 1 if the address corresponds to a currently managed structure, 0 otherwise.
  - **Description:** Checks if a given MM_ADDRESS$ is valid and currently tracked by the memory manager.
  - **Example:**

```
DIM addr$
addr$ = MM_CREATE$("SL")
PRINT "Exists? " + MM_EXISTS(addr$) % Prints 1
MM_DELETE(addr$)
PRINT "Exists? " + MM_EXISTS(addr$) % Prints 0
PRINT "Exists? " + MM_EXISTS("rfoMM(abc)") % Prints 0
```

- **MM_EXISTS$(MM_ADDRESS$)**
  - **Parameters:**
    - MM_ADDRESS$ (String): The address to check.
  - **Returns:** Exists$ (String): "TRUE" if the address corresponds to a currently managed structure, "FALSE" otherwise.
  - **Description:** String version of MM_EXISTS. Checks if a given MM_ADDRESS$ is valid and currently tracked.

- **Example:**
  ```
  DIM addr$
  addr$ = MM_CREATE$("B")
  PRINT "Exists? " + MM_EXISTS$(addr$) % Prints TRUE
  MM_DELETE(addr$)
  PRINT "Exists? " + MM_EXISTS$(addr$) % Prints FALSE
  ```

- **MM_ADDRESS$(struct_id, type$)**
  - **Parameters:**
    - struct_id (Numeric): The RFO Basic numeric pointer (>0) of a structure.
    - type$ (String): The expected type of the structure ("B", "SL", "NL", "C", "O").
  - **Returns:** MM_ADDRESS$ (String): The corresponding memory address if the pointer/type pair is found within the managed structures, otherwise "" (empty string).
  - **Description:** Looks up an *already managed* structure using its raw pointer and type to find its associated MM_ADDRESS$. Useful if you have a pointer (e.g., stored previously or obtained externally) and need to find its rfoMM handle.
  - **Example:**
    ```
    DIM addr1$, ptr1, foundAddr$
    addr1$ = MM_CREATE$("NL")
    ptr1 = MM_POINTER(addr1$)
    foundAddr$ = MM_ADDRESS$(ptr1, "NL")
    PRINT "Found Address: " + foundAddr$ % Prints original addr1$
    PRINT "Lookup wrong type: " + MM_ADDRESS$(ptr1, "SL") %
    Prints ""
    MM_DELETE(addr1$)
    ```
  - **Notes:** Only works for structures already managed by rfoMM. Use MM_LOAD$ to bring unmanaged structures into the system.

- **MM_KEYS$(MM_ADDRESS$)**
  - **Parameters:**
    - MM_ADDRESS$ (String): The address of a managed **Bundle**.
  - **Returns:** KeysListAddress$ (String): The MM_ADDRESS$ of a *newly created and managed String List* containing the keys of the bundle, or potentially "" if input address is invalid or not a bundle.
  - **Description:** Retrieves the keys from a managed bundle. Importantly, the list containing the keys is itself automatically created and managed by rfoMM, and this function returns the address *of that new list*.
  - **Example:**
    ```
    DIM bAddr$, bPtr, kAddr$, kPtr
    bAddr$ = MM_CREATE$("B")
    bPtr = MM_POINTER(bAddr$)
    BUNDLE.PUT bPtr, "Name", "JJ"
    BUNDLE.PUT bPtr, "Value", 123

    kAddr$ = MM_KEYS$(bAddr$)
    PRINT "Address of keys list: " + kAddr$
    IF MM_EXISTS(kAddr$)
    ```

```
  PRINT "Keys list type: " + MM_TYPE$(kAddr$) % Prints SL
  kPtr = MM_POINTER(kAddr$)
  LIST.SIZE kPtr, kSize
  PRINT "Number of keys: " + kSize
  LIST.TOSTRING kPtr, kStr$
  PRINT "Keys: " + kStr$
  MM_DELETE(kAddr$) % IMPORTANT: Delete the keys list when
done
ELSE
  PRINT "Failed to get keys list address."
ENDIF
MM_DELETE(bAddr$) % Delete original bundle
```

- **Notes:** Because MM_KEYS$ creates a *new* managed list, you are responsible for calling MM_DELETE on the returned KeysListAddress$ when you are finished with the keys list to allow it to be recycled. Failure to do so will leave the keys list managed indefinitely.

- **MM_KEYS(MM_ADDRESS$)**
  - **Parameters:**
    - MM_ADDRESS$ (String): The address of a managed **Bundle**.
  - **Returns:** KeysListPointer (Numeric): The raw numeric pointer of a *newly created and managed String List* containing the keys, or 0 if an error occurs.
  - **Description:** Wrapper function for MM_KEYS$. It calls MM_KEYS$, then uses MM_POINTER on the resulting keys list address to return the raw numeric pointer of the keys list.
  - **Example:**

```
DIM bAddr$, bPtr, kPtr
bAddr$ = MM_CREATE$("B")
bPtr = MM_POINTER(bAddr$)
BUNDLE.PUT bPtr, "Alpha", 1

kPtr = MM_KEYS(bAddr$)
PRINT "Pointer of keys list: " + kPtr
IF kPtr > 0
  LIST.SIZE kPtr, kSize
  PRINT "Key count: " + kSize
  % IMPORTANT: Need to delete the managed keys list later
  % Find its address first if needed: kAddr$ =
MM_ADDRESS$(kPtr, "SL")
  % MM_DELETE(kAddr$)
ENDIF
MM_DELETE(bAddr$)
```

  - **Notes:** While convenient for getting the pointer directly, you still need to manage the lifecycle of the created keys list. It's often clearer to use MM_KEYS$ and explicitly manage the list via its address. Remember the keys list is managed by rfoMM even though you only get the pointer here.

- **MM_VALIDATE$(MM_ADDRESS$)**
  - **Parameters:**
    - MM_ADDRESS$ (String): The address of a managed structure (typically a Bundle intended to represent an rfOOP Class or Object).
  - **Returns:** Status$ (String): "TRUE" if the structure exists and its underlying bundle contains expected rfOOP keys (NAME, INDEX, TYPE, SUB TYPE, TYPINGS, CLASS VARIABLES, OBJECT VARIABLES), otherwise "FALSE".
  - **Description:** Performs a structural validation check, primarily intended for internal use within the planned rfOOP framework or for debugging rfOOP-like structures. It verifies the presence of specific keys expected in bundles representing classes/objects in that system.
  - **Example:**

```
DIM objAddr$, objPtr, status$
objAddr$ = MM_CREATE$("O") % Create "Object" type bundle
status$ = MM_VALIDATE$(objAddr$)
PRINT "Validation status (empty): " + status$ % Prints FALSE

objPtr = MM_POINTER(objAddr$)
BUNDLE.PUT objPtr, "NAME", "MyObj"
BUNDLE.PUT objPtr, "INDEX", objPtr
BUNDLE.PUT objPtr, "TYPE", "O"
BUNDLE.PUT objPtr, "SUB TYPE", "MySubType"
BUNDLE.PUT objPtr, "TYPINGS", 0
BUNDLE.PUT objPtr, "CLASS VARIABLES", 0
BUNDLE.PUT objPtr, "OBJECT VARIABLES", 0
status$ = MM_VALIDATE$(objAddr$)
PRINT "Validation status (filled): " + status$ % Prints TRUE
MM_DELETE(objAddr$)
```

  - **Notes:** This function's utility depends heavily on adherence to the specific rfOOP structure conventions. For general bundles, it will likely return "FALSE". And the Example above is pre rfOOP commands, so we manually give the "Properties" to objPtr.

4.3. Using Standard RFO Basic Commands

rfoMM manages the *identity*, *type*, and *lifecycle* of structures. To manipulate the *data inside* those structures, you use standard RFO Basic commands (BUNDLE.PUT, BUNDLE.GET, LIST.ADD, LIST.GET, LIST.SIZE, etc.) combined with the numeric pointer obtained via MM_POINTER.

General Pattern:

1. Get the MM_ADDRESS$ (e.g., from MM_CREATE$).
2. Get the numeric pointer using MM_POINTER(MM_ADDRESS$).
3. Check if pointer > 0.
4. Use the pointer with standard RFO Basic commands (e.g., BUNDLE.PUT pointer, key$, value$).

See examples under MM_POINTER and MM_KEYS$ for illustrations.

4.4. Internal Functions (Brief Overview)

These functions are used by rfoMM internally. Direct calls by user code are typically unnecessary and not recommended as their behavior might change in future versions.

- **MM_OPEN(type$):** Checks internal lists for a recyclable numeric pointer of the specified type.
- **MM_RECYCLE(type$):** Retrieves a recycled pointer from the internal list (removing it) or triggers the creation of a new structure via MM_CREATE if no recycled pointers are available.

5. Examples / Use Cases

Example 1: Basic Bundle Management

```
INCLUDE rfoMM.bas
MM_INIT()

FN.DEF TEST_PRINT(TestDesc$, Condition)
 IF Condition THEN PRINT TestDesc$ + "... PASSED" ELSE PRINT TestDesc$
+ "... FAILED"
FN.END


DIM playerAddr$, playerPtr, name$, score

PRINT "Creating player data bundle..."
playerAddr$ = MM_CREATE$("B")
TEST_PRINT("Bundle created?", playerAddr$ <> "")

playerPtr = MM_POINTER(playerAddr$)
IF playerPtr > 0
  PRINT "Setting player data using pointer " + playerPtr
  BUNDLE.PUT playerPtr, "Name", "Hero"
  BUNDLE.PUT playerPtr, "Score", 1000

  BUNDLE.GET playerPtr, "Name", name$
  BUNDLE.GET playerPtr, "Score", score
  PRINT "Player Name: " + name$ + ", Score: " + score
ELSE
  PRINT "Failed to get pointer for " + playerAddr$
ENDIF

PRINT "Deleting player data bundle..."
MM_DELETE(playerAddr$)
```

```
TEST_PRINT("Bundle deleted?", !MM_EXISTS(playerAddr$))

END
```

Example 2: Managing a List and Its Keys

```
INCLUDE rfoMM.bas
MM_INIT()
DIM settingsAddr$, settingsPtr, keysAddr$, keysPtr, i, keyCount, key$

PRINT "Creating settings bundle..."
settingsAddr$ = MM_CREATE$("B")
settingsPtr = MM_POINTER(settingsAddr$)
IF settingsPtr > 0
  BUNDLE.PUT settingsPtr, "Volume", 8
  BUNDLE.PUT settingsPtr, "Difficulty", "Hard"
  BUNDLE.PUT settingsPtr, "User", "JJ"

  PRINT "Getting keys list..."
  keysAddr$ = MM_KEYS$(settingsAddr$)
  IF MM_EXISTS(keysAddr$)
    keysPtr = MM_POINTER(keysAddr$)
    PRINT "Keys list address: " + keysAddr$ + ", Pointer: " + keysPtr
    IF keysPtr > 0
      LIST.SIZE keysPtr, keyCount
      PRINT "Settings contains " + keyCount + " keys:"
      FOR i = 1 TO keyCount
        LIST.GET keysPtr, i, key$
        PRINT " - " + key$
      NEXT i
    ENDIF
    PRINT "Deleting keys list..."
    MM_DELETE(keysAddr$) % Clean up the keys list
  ELSE
    PRINT "Failed to get keys list."
  ENDIF
ELSE
  PRINT "Failed to create settings bundle."
ENDIF

PRINT "Deleting settings bundle..."
MM_DELETE(settingsAddr$) % Clean up the settings bundle

END
```

Example 3: Loading and Managing External Data

```
INCLUDE rfoMM.bas
MM_INIT()
DIM rawDataPtr, dataAddr$, dataPtr, value

PRINT "Simulating external data creation..."
LIST.CREATE N, rawDataPtr
LIST.ADD rawDataPtr, 10, 20, 30
PRINT "Raw data pointer: " + rawDataPtr

PRINT "Loading raw data into rfoMM..."
dataAddr$ = MM_LOAD$(rawDataPtr, "NL")
IF dataAddr$ <> ""
  PRINT "Loaded. Managed Address: " + dataAddr$
  dataPtr = MM_POINTER(dataAddr$)
  IF dataPtr = rawDataPtr % Verify pointer is the same
    LIST.GET dataPtr, 2, value
    PRINT "Value at index 2: " + value % Should be 20
  ELSE
    PRINT "Pointer mismatch after load!"
  ENDIF

  PRINT "Deleting managed external data..."
  MM_DELETE(dataAddr$)
  PRINT "Exists after delete? " + MM_EXISTS(dataAddr$) % Should be 0
ELSE
  PRINT "Failed to load raw data."
ENDIF

END
```

6. Known Issues & Limitations (v0.20)

- **Performance:** While designed to be efficient, managing extremely large numbers (tens of thousands) of concurrent structures might incur performance overhead due to internal bundle lookups (MM_POINTER, MM_TYPE$, MM_DELETE). Consider architectural choices if dealing with such scales. But Let's be honest, if you are creating APK's of that size, are you really using a language like RFO Basic?
- **Error Handling:** Error handling is basic. Invalid MM_ADDRESS$ inputs typically result in functions returning 0, "", or "UNDEFINED". Critical errors (e.g., invalid type in MM_CREATE$) may PRINT a message and END. A more robust error/exception system is planned for future integration with rfOOP.

- **Concurrency:** Not designed for multi-threaded scenarios (if ever applicable in RFO Basic contexts).

7. Future Plans

rfoMM is a foundational piece. Future development within Project Nebula may include:

- **Data Persistence:** Integration with a future data persistence/serialization pillar, likely using MM_ADDRESS$ as the key identifier for saving/loading complex data graphs.
- **Stack Support:** Potential addition of "SS" and “NS” as a supported type.
- **Advanced Error Handling:** Implementing a more structured error reporting mechanism.

8. License

rfoMM is distributed under the terms of the **GNU General Public License version 3 (GPLv3)**. Please refer to the LICENSE.txt file included in the package distribution for the full license text.