This is part of **Win16 API** which allow to create versions of program from one source code to run under OS/2 and Win16. Under OS/2 program can be running under Win-OS/2 if program is Windows NE executable, and with help on Windows Libraries for OS/2, if it is OS/2 NE executable. Here is a WLO to OS/2 API mapping draft

2021/09/01 04:23 · prokushev · 0 Comments

DGROUP, Local Heap and Atom Table

Overview

In 16-bit versions of Windows (1.x, 2.x, 3.x), each module (application or DLL) typically has its own data segment (DGROUP) limited to 64 KB. This segment contains the stack, the local heap, and the atom table. Local heap functions manage memory within this segment using near pointers (offsets). They are exported by the KERNEL module. Internally, the local heap is managed through a set of data structures that reside inside the segment itself.

A key part of this management is the Instance Data (also called the NULL segment) located at the very beginning of DGROUP, which holds pointers to the heap, atom table, and stack information. The field at offset 6 (pLocalHeap) contains a near pointer to the HeapInfo structure that heads the local heap. This pointer is set by LocalInit() and must be validated by checking the signature word (li_sig) at offset 22h (KRNL286) or 28h (KRNL386) within the presumed heap.

The local heap itself is organized as a series of arenas (headers) that precede each block. The two low bits of the la_prev field in an arena are used as flags: bit 0 indicates whether the block is in use (1) or free (0); bit 1 indicates whether the block is MOVEABLE (1) or FIXED (0). Free blocks are linked via la_free_prev and la_free_next. For MOVEABLE blocks, a separate handle table (pointed to by hi_htable in HeapInfo) stores the actual address, lock count, and flags; each handle is an offset into this table.

Internal Structures

Instance Data (NULL Segment)

The first 16 (10h) bytes of the default data segment (DGROUP) are reserved for system use and are referred to as the Instance Data or NULL segment. This area is present only if the first WORD at offset 0 is zero; otherwise, the structure is not present. The layout is as follows:

Offset	Type	Field	Description
00h	WORD	wMustBeZero	Must be zero for the NULL segment structure to be considered present.
02h	DWORD	dwOldSSSP	When SwitchStackTo() is called, the current SS:SP is stored here. At other times, may contain the value 5 (number of reserved pointers).

Offset	Type	Field	Description
06h	WORD	pLocalHeap	Near pointer to the Local Heap information structure (i.e., the HeapInfo structure). This field is set by LocalInit() and points to the beginning of the local heap management structures. If no local heap exists, this field may be stale (non-zero but invalid). Always verify the heap by checking the signature at offset 22h (KRNL286) or 28h (KRNL386) – see li_sig below.
08h	WORD	pAtomTable	Near pointer to the atom table structure. Set by InitAtomTable(). Zero until atoms are used.
0Ah	WORD	pStackTop	Near pointer to the end (top) of the stack. For DLLs, this is zero.
0Ch	WORD	pStackMin	High-water mark of stack usage. For DLLs, zero.
0Eh	WORD	pStackBottom	Near pointer to the beginning (bottom) of the stack. For DLLs, zero.

Important Notes:

- The field at offset 6 (pLocalHeap) is the primary way to locate the local heap structures given only the DGROUP selector.
- When LocalInit() is called on a globally allocated block (non-DGROUP), the WORD at offset 6 of that block is also set to point to the local heap information structure for that block.
- Similarly, if InitAtomTable() is called on a global block, offset 8 points to the atom table, and offset 6 will point to the associated local heap (since atoms are stored in the local heap).

HeapInfo and LocalInfo

Every local heap begins with an instance of the HeapInfo structure, which is identical to the one used by the global heap and is defined in WINKERN.INC. Its location is given by the pLocalHeap field at offset 6 of the Instance Data. Immediately following the HeapInfo structure are additional fields that, together with HeapInfo, form the LocalInfo structure.

HeapInfo Structure (386)

Under KRNL386, the HeapInfo structure occupies **1Eh** bytes and has the following format (according to “Windows Internals” documentation, Table 2-2):

Offset	Type	Field	Description
00h	WORD	hi_check	If this value is nonzero, the debug version of KERNEL verifies the heap. This field appears to be used only for the local heap, not for the global heap.
02h	WORD	hi_freeze	If this is nonzero, KERNEL should not compact the heap. For the global heap, this value appears to be set only while inside the INT 24h handler. The local heap is frozen during LocalAlloc and LocalRealloc. LocalFreeze and LocalMelt also changes this field.
04h	WORD	hi_count	The total number of blocks in the heap.
06h	DWORD	hi_first	A far pointer to the arena header for the first block in the heap. The first block is always a sentinel and points to itself.
0Ah	DWORD	hi_last	A far pointer to the arena header for the last block in the heap. The last block is always a sentinel and points to itself.

Offset	Type	Field	Description
0Eh	BYTE	hi_ncompact	The number of compactions that have been performed on the heap to try and free up memory for a particular allocation. Some code appears to use this field as a count, while other code seems to use it as bitfields.
0Fh	BYTE	hi_dislevel	According to WINKERN.INC, it is the current discard level. Both the local and global heaps use it. The global heap treats the value as a bitfield, using it with flags such as GA_NODISCARD.
10h	DWORD	hi_distotal	Only used by the global heap. When discarding begins, this field contains the number of bytes that need to be discarded. As the global heap discards blocks, it subtracts their sizes from this field. When the field reaches zero or below, discarding stops.
14h	WORD	hi_htable	This field contains a near pointer to a handle table for the heap. Only the local heap uses this field.
16h	WORD	hi_hfree	Near pointer to the free handle table. Only local heap uses it.
18h	WORD	hi_hdelta	When the local heap needs to increase the number of handles it has, it allocates the number of handles specified in this field. The default value is 20h.
1Ah	WORD	hi_hexpand	A near pointer to the function that KERNEL uses to increase the number of handles for the local heap. Because it's a near pointer, the function must reside in the KERNEL code segment. Thus, there's no way to hook this function. The Global heap does not use it.
1Ch	WORD	hi_pstats	A near pointer to a LocalStats structure which the local heap uses in the debug KERNEL. As the local heap does various things, such as search for free blocks, it increments fields in the structure. The structure is defined in WINKERN.INC.

HeapInfo Structure (286)

Under KRNL286, the HeapInfo structure occupies 18h bytes. The layout is as follows:

Offset	Type	Field	Description
00h	WORD	hi_check	If nonzero, debug kernel verifies the heap. Used only for the local heap.
02h	WORD	hi_freeze	If nonzero, heap compaction is disabled.
04h	WORD	hi_count	Total number of blocks in the heap.
06h	WORD	hi_first	Near pointer to the arena header of the first block (always a sentinel pointing to itself).
08h	WORD	hi_last	Near pointer to the arena header of the last block (always a sentinel pointing to itself).
0Ah	BYTE	hi_ncompact	Number of compactions performed (or bitfield).
0Bh	BYTE	hi_dislevel	Current discard level.
0Ch	WORD	hi_distotal	When discarding begins, this field contains the number of bytes that need to be discarded. As discarding proceeds, the sizes of discarded blocks are subtracted until the value reaches zero or below.
0Eh	WORD	hi_htable	Near pointer to the handle table for moveable blocks. Used only by the local heap.
10h	WORD	hi_hfree	Near pointer to the free handle table entry list.
12h	WORD	hi_hdelta	Number of handle table entries allocated when the table needs to grow. Default is 20h.
14h	WORD	hi_hexpand	Near pointer to the function that expands the handle table. This function must reside in the KERNEL code segment.

Offset	Type	Field	Description
16h	WORD	hi_pstats	Near pointer to a LocalStats structure (used in debug kernel).

LocalInfo Structure

For both **KRNL386** and **KRNL286**, the LocalInfo structure immediately follows the HeapInfo structure and has the following layout:

Offset	Type	Field	Description
00h	DWORD	li_notify	Far pointer to a routine called either when a heap block is about to be moved or discarded, or when the heap is out of memory. Initialized to point at LocalNotifyDefault().
04h	WORD	li_lock	Lock count of the local heap. A non-zero value prevents blocks from moving or being discarded.
06h	WORD	li_extra	Minimum amount in bytes by which the local heap should be grown when expanded. Default is 200h.
08h	WORD	li_minsize	Minimum size of the local heap, as specified by the HEAPSIZE line in the .DEF file.
0Ah	WORD	li_sig	Signature word set to 484Ch ('LH' in a hex dump). Used by various Windows routines to verify heap integrity. This signature should be checked when validating a potential heap pointer from offset 6 of the Instance Data.

Important:

- In KRNL286, the li_sig field is located at offset 22h from the beginning of the combined HeapInfo + LocalInfo structure.
- In KRNL386, the li_sig field is located at offset 28h from the beginning of the combined HeapInfo + LocalInfo structure.

Arena Formats

Every block in the local heap is preceded by an arena (header) that contains management information. Arenas always start on a 4-byte boundary, so the two low bits of every arena address are zero. These bits are reused as flags in the la_prev field of each arena. The two low bits of la_prev have the following meaning:

- Bit 0 (least significant): Set if the block is in use (FIXED or MOVEABLE); cleared if the block is free.
- Bit 1: Set if the block is MOVEABLE; cleared if the block is FIXED (only meaningful when bit 0 is set).

Thus, to obtain the real address of the previous arena, the two low bits must be masked off.

There are three arena formats, corresponding to the three possible block states:

FIXED Block Arena

Offset	Type	Field	Description
00h	WORD	la_prev	Near pointer to the preceding arena, with flags in the low two bits.
02h	WORD	la_next	Near pointer to the next arena.

For a FIXED block, the handle is the address of the block itself. The arena can be found by subtracting 4 from the block address.

MOVEABLE Block Arena

Offset	Type	Field	Description
00h	WORD	la_prev	Near pointer to the preceding arena, with flags.
02h	WORD	la_next	Near pointer to the next arena.
04h	WORD	la_handle	Offset of the handle table entry for this block.

The la_handle field provides two-way mapping between a MOVEABLE block and its handle table entry. Given the block address, subtract 6 to get the arena; the la_handle field gives the offset of the handle entry. Given a handle entry, the first field (lhe_address) gives the block address.

Free Block Arena

Offset	Type	Field	Description
00h	WORD	la_prev	Near pointer to the preceding arena, with flags.
02h	WORD	la_next	Near pointer to the next arena.
04h	WORD	la_size	Size of the block, including the arena.
06h	WORD	la_free_prev	Offset of the previous free arena.
08h	WORD	la_free_next	Offset of the next free arena.

Free blocks are threaded in a doubly linked list using la_free_prev and la_free_next. The start of this free list is stored in the la_free_next field of the first block in the heap (see below).

The First Local Heap Block

The first block in the local heap is special. It resides in memory before the LocalInfo structure. Although its la_prev field has the low bit set (indicating a FIXED, in-use arena), the local heap routines treat this arena as if it were free. The la_free_next field of this first block points to the first truly free block in the heap.

Handle Table

For MOVEABLE blocks, handles are offsets into a handle table that resides in its own local heap blocks. The first handle table is located via the hi_htable field in the HeapInfo structure. Each handle table begins with a WORD specifying how many handle entries follow. After all entries, the last WORD is the offset of the next handle table (or 0 if none). Free handle entries are linked together for quick

allocation.

Format of Handle Table Entries (in-use):

Offset	Type	Field	Description
00h	WORD	lhe_address	Address of the memory block referenced by the handle.
02h	BYTE	lhe_flags	Flags: 0Fh = LHE_DISCARDABLE (discard level), 1Fh = LHE_USERFLAGS (reserved for programmer?), 40h = LHE_DISCARDED (block has been discarded).
03h	BYTE	lhe_count	Lock count of the block. Non-zero prevents moving or discarding.

Format of Handle Table Entries (free):

Offset	Type	Field	Description
00h	WORD	lhe_link	Offset of the next free handle table entry.
02h	WORD	lhe_free	Always FFFFh to indicate a free entry.

Heap Operations

- **Allocation (LocalAlloc)** walks the free list, splitting blocks if necessary, and sets up the appropriate arena. For MOVEABLE blocks, it also allocates a handle table entry.
- **Compaction (LocalCompact)** coalesces adjacent free blocks and may move or discard unlocked MOVEABLE blocks. When a block is moved, its lhe_address is updated.
- **Locking (LocalLock/LocalUnlock)** manipulates the lhe_count field of the handle entry for MOVEABLE blocks; for FIXED blocks, no count is maintained.
- **Discarding (LocalDiscard)** frees the memory of a MOVEABLE block but keeps the handle entry alive with the LHE_DISCARDED flag set.

Atom Tables

Atom tables provide a mechanism for efficiently storing and sharing strings (atoms) in 16-bit Windows. Each module may have its own local atom table located in its local heap. In addition, there is a single global atom table that serves the entire system.

The field at offset **08h** in the Instance Data (`pAtomTable`) contains a near pointer to the atom table header for the current module. This pointer is initialized by calling `InitAtomTable()`. If no atom table exists, `pAtomTable` is zero.

Relationship with the Local Heap

Physically, an atom table resides **inside** the local heap of some data segment. Therefore, before creating an atom table, the segment must be initialized as a local heap by calling LocalInit.

ATOMENTRY Structure

Each string atom is stored as an `ATOMENTRY` structure in the local heap. The structure has the following form:

Offset	Type	Field	Description
00h	WORD	`next`	Next entry in the same hash bucket (0 if last).
02h	WORD	`usage`	Reference count.
04h	BYTE	`len`	Length of the string (1–255).
05h	BYTE[]	`name`	ASCIIZ string (length `len` + 1).

Types of Atoms

Windows supports two fundamentally different types of atoms: **string atoms** and **integer atoms**. Their handling is completely distinct.

String Atoms

String atoms are created by passing an ordinary string to AddAtom. They are stored in the atom table as `ATOMENTRY` structures.

- **Range**: `0xC000` to `0xFFFF` (encoded pointer).
- **Storage**: Allocated in the local heap as `ATOMENTRY`, inserted into the hash table.
- **Reference count**: Yes (`usage` field).
- **String representation**: The original string.
- **Creation**: `AddAtom("MyString")`.

Encoding: A string atom value is derived from the near pointer to its `ATOMENTRY`. Since the pointer is 4-byte aligned, the low two bits are zero. The atom is formed by shifting the pointer right by 2 bits and ORing with `0xC000`. This guarantees the range `0xC000–0xFFFF`.

```
#define HANDLETOATOM(handle) ((ATOM)(0xc000 | ((handle) >> 2)))
#define ATOMTOHANDLE(atom)   ((HANDLE16)(atom) << 2)
```

Integer Atoms

Integer atoms are created by passing a string of the form `"#dddd"` (or by using `MAKEINTATOM` with a value ≤ 0xBFFF). They are **not stored in the atom table** and have no associated `ATOMENTRY` structure.

- **Range**: `0x0001` to `0xBFFF` .
- **Storage**: None; the value is used directly as the atom.
- **Reference count**: Not applicable.
- **String representation**: Generated on the fly as `"#dddd"` when `GetAtomName` is called.
- **Creation**: `AddAtom("#1234")` or `AddAtom(MAKEINTATOM(0x04D2))`.

How it works: When a string of the form '#dddd' is passed, the function parses the decimal number and, if it is less than 0xC000, returns it directly without accessing the atom table. Similarly, `FindAtom` for such a string or for a `MAKEINTATOM` value simply returns the number without any lookup. Integer atoms are always considered “found” because any value in the range is valid.

MAKEINTATOM Macro

The `**MAKEINTATOM**` macro is defined as:

```
#define MAKEINTATOM(i)  (LPTSTR)((DWORD)((WORD)(i)))
```

This macro casts a 16-bit integer value to a pointer type. When this “pointer” is passed to atom functions, it is interpreted as an integer atom (if the value is ≤ `0xBFFF`) or as a string atom (if ≥ `0xC000`).

- For values ≤ `0xBFFF`, the function treats it as an integer atom and returns the value directly.
- For values ≥ `0xC000`, the function assumes it is an encoded pointer to an `ATOMENTRY` and will dereference it (after shifting left by 2 bits) to access the atom entry.

Important: `MAKEINTATOM` does not create a string or allocate any memory; it is simply a type-punning convenience to pass integer atoms to functions that formally expect a string pointer.

Creating Custom Atom Tables (outside DGROUP)

Since all atom operations work with the current segment pointed to by DS, you can create and use an atom table in any arbitrary data segment by following three steps:

1. **Create a local heap** in the target segment using `LocalInit(Selector, Start, End)`.
2. **Switch the DS register** to that segment.
3. Call `InitAtomTable(size)` to initialize the atom table in the newly created heap.

After that, any subsequent call to `AddAtom`, `FindAtom`, etc., will operate on the custom table if DS is temporarily set to the correct segment.

Summary of Atom Type Differences

Feature	String Atoms	Integer Atoms
Range	`0xC000` – `0xFFFF`	`0x0001` – `0xBFFF`
Stored in atom table	Yes, as `ATOMENTRY` in hash buckets	No
Memory allocated	`ATOMENTRY` structure in local heap	None
Reference count	Yes (`usage`)	No
String representation	Original string	Generated as `“#dddd”` on the fly
Creation	`AddAtom(“MyString”)`	`AddAtom(“#1234”)` or `AddAtom(MAKEINTATOM(0x04D2))`

Feature	String Atoms	Integer Atoms
Find behavior	Searches hash table	Always returns the value (always “found”)
Delete behavior	Decrements refcount, frees if zero	No operation (returns 0)

Custom Local Heaps

Although each module typically has a default local heap in its DGROUP, it is possible to create additional local heaps in other segments or in globally allocated memory. This is useful for managing private memory pools within a large global block, or for creating atom tables in separate memory areas.

Using LocalInit() to Create a Custom Local Heap

The `LocalInit()` function initializes a local heap within a specified segment. Its prototype is:

```
WORD LocalInit(WORD wSegment, WORD pStart, WORD pEnd);
```

- `wSegment` – Selector of the segment where the heap will be created.
- `pStart` – Offset of the first byte of the heap area (must be paragraph-aligned, i.e., a multiple of 16).
- `pEnd` – Offset of the last byte of the heap area (inclusive). The heap will manage memory from `pStart` to `pEnd`.

If successful, `LocalInit()` returns a non-zero value. It sets up the `HeapInfo` and `LocalInfo` structures at the beginning of the heap area (starting at `pStart`) and updates the segment’s instance data at offset **06h** (`pLocalHeap`) to point to that `HeapInfo` structure. However, if the segment is not a default data segment (i.e., not DGROUP), the instance data at offset 0 must also contain a zero word to indicate that the NULL segment structure is present; otherwise, the heap may not be recognized by some routines.

Example: Creating a local heap in a globally allocated block of memory (64 KB):

```
; 1. Allocate a 64KB global memory block
GlobalAlloc GMEM_FIXED, 0x10000
mov dx, ax          ; DX = selector of allocated block

; 2. Temporarily set DS to that segment to access its instance data
push ds
push dx
pop ds

; 3. Initialize the NULL segment (Instance Data) at offset 0.
;    The first word must be zero (wMustBeZero = 0).
xor ax, ax
mov word ptr [0], ax
; The other fields (pLocalHeap, etc.) will be filled by LocalInit.

; 4. Define the heap area: start at offset 16 (0x0010) to preserve
```

```
;    the 16-byte Instance Data, end at 0xFFFF (the last byte of the
segment).
mov bx, 16          ; pStart = 16
mov cx, 0xFFFF      ; pEnd = 0xFFFF

; 5. Restore DS (if no longer needed)
pop ds

; 6. Call LocalInit to create the heap in segment dx, from pStart to pEnd.
push dx
push bx
push cx
call LocalInit      ; returns non-zero on success

; After the call, the Instance Data at offset 6 in segment dx
; contains a valid near pointer to the HeapInfo structure located at offset
16.
```

After this call, the global block can be used with local heap functions (`LocalAlloc`, `LocalFree`, etc.) by using the selector in `DS` and near pointers (offsets) within that segment.

Important Considerations:

- The heap structures themselves occupy space at the beginning of the heap area. The first block (sentinel) resides at `pStart + size of (LocalInfo)`.
- The segment's instance data (at offset 0) must be properly set up, especially the zero word at offset 0, to avoid confusion with other structures.
- Custom local heaps are not automatically enlarged if they run out of space; they are limited to the range specified in `LocalInit`.
- The `HEAPSIZE` setting in the module's .DEF file only affects the default DGROUP heap.

Creating Atom Tables Outside DGROUP

As already described, to create an atom table in an arbitrary memory area, you must first initialize a local heap there (as above) and then, after switching DS, call `InitAtomTable`. This technique allows fully isolated atom tables for special purposes.

Summary of Custom Heap and Atom Table Creation

- Use LocalInit on a segment to establish a local heap anywhere in memory.
- The segment must have a valid NULL segment structure (zero word at offset 0) for the heap to be recognized.
- After `LocalInit`, you can use `LocalAlloc`, `LocalLock`, etc., with near pointers within that segment.
- To create an atom table in a custom heap, switch DS to that segment and call InitAtomTable.
- All subsequent atom operations must be performed with DS set appropriately (or via wrapper functions).
- Custom heaps and atom tables are useful for isolating memory pools, implementing resource

managers, or working with large data structures without polluting the default DGROUP.

References

1. Schulman, A., Maxey, D., Pietrek, M. *Undocumented Windows*. Addison-Wesley, 1992.
2. Pietrek, M. *Windows Internals*. Addison-Wesley, 1993.
3. Chen, R. *The Old New Thing* (blog). Microsoft Developer Blogs.
4. Microsoft OS/2 Version 1.1 Programmer's Reference, Volume 1.