Ultima VI

Design Notes Collection - Technical Design

OBJECT DESIGN

This section discusses those features about objects that will be implemented in ***Ultima VI*** . The primary author of this section is Herman Miller. Any questions should be addressed to him.

DEFINITIONS

Object: An object is any animate or inanimate thing that can be manipulated or changed by a player of ***Ultima VI*** . In the inanimate case, doors, bottles, chairs, carts, chests, etc. are considered to be objects of the simple kind. These objects are described by a similar structure listed in further pages of this document. In the animate case, an object may be considered to be a Non-Player character, an enchanted, dancing chair, or a monster. In general, an animate object could also be called a monster or NPC, In a few rare cases will an object be animate that does not take on living characteristics. As in an inanimate object, an animate object has the same Object Structure to define its basic characteristics, but it also contains additional information to define its more living qualities

Shape Type The Shape Type of a tile is the Number of the first tile in the group of tiles or it may be a reference number base for a tile. The first case would apply to objects like doors. The shape type would be the door tile number. Whether it was open or not (a different tile), the shapetype for any door found would be the same. The second case would make the shape type the first tile in an animation sequence (contiguous tiles).

Safe Macro: A macro that does not cause mysterious effects to occur when a parameter uses increment ++ and decrement -- operators, or other potentially destructive expressions.

Unsafe Macro: A Macro that causes mysterious effects to occur when the parameter(s) passed contain complex expressions and/or increment ++ or decrement -- operators.

RESTRICTIONS AND USES

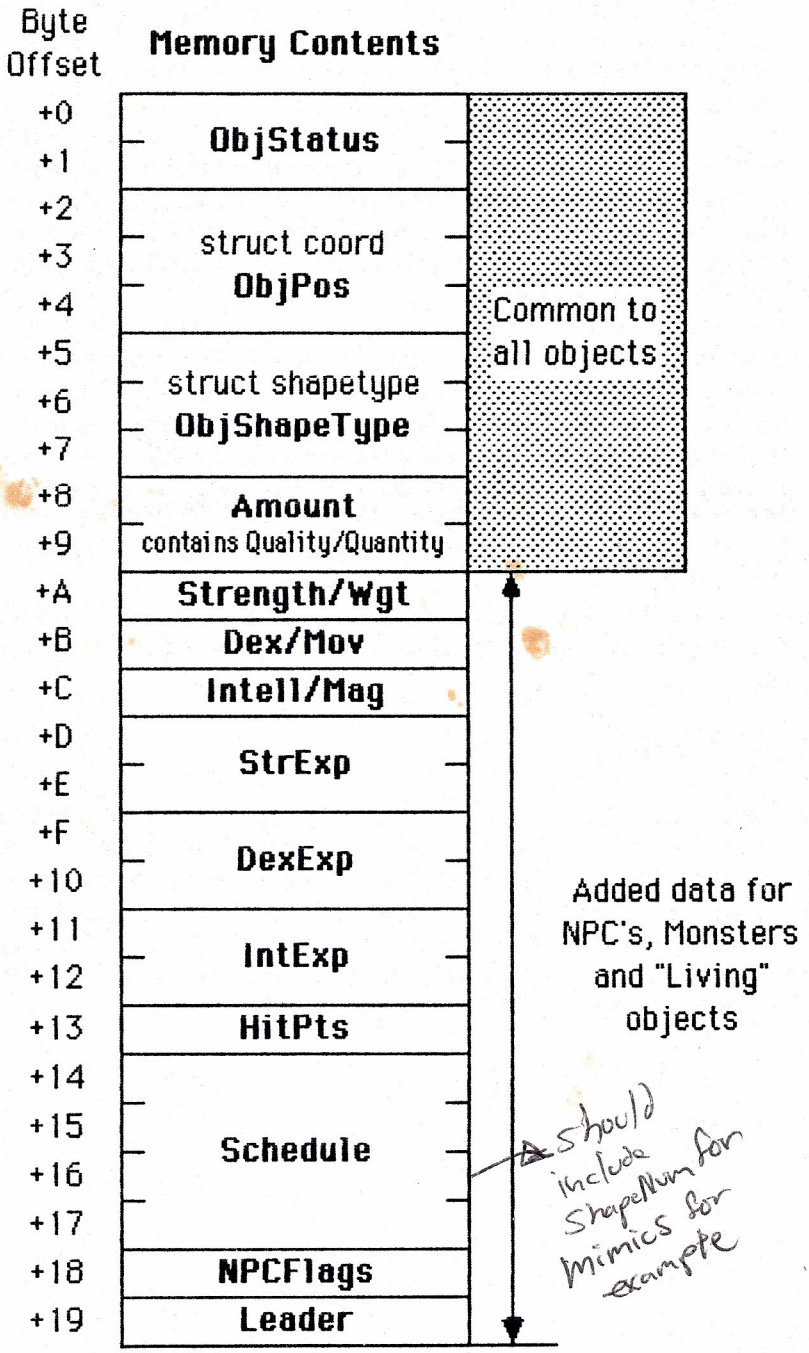
Since every thing that can be manipulated by an ***Ultima VI*** player is considered to be an object, there are very few restrictions as to what may potentially be done by the player. It is concievable to turn a statue of a tiger into a real tiger. It is possible to reduce a person to wood shavings. A door can be turned into a window, or a wizard. There is a practical limit to the number of more complex objects in the game. Because of the increase in size of the structure, a limit of 256 of these objects is allotted. Most of these will be the NPC's that can be found in the game. There is a possibility that another 32 or so will be added to act as monsters to be generated for combat.

Ultima VI

Design Notes Collection - Technical Design

DATA STRUCTURES

The Object list is a compilation of similar data structures. For simplicity and speed, the first 256 of these structures are the extended type needed for characters and monsters. The next 256 Objects are the party inventory. The rest of the Object List is divided into 16 regions, each region being part of the map. When characters cross from one map region to another, the Object Lists for the new regions are loaded and the old ones are saved. In the IBM implementation the structures are saved in memory as parallel arrays. For purposes of discussion, they will be viewed as follows:

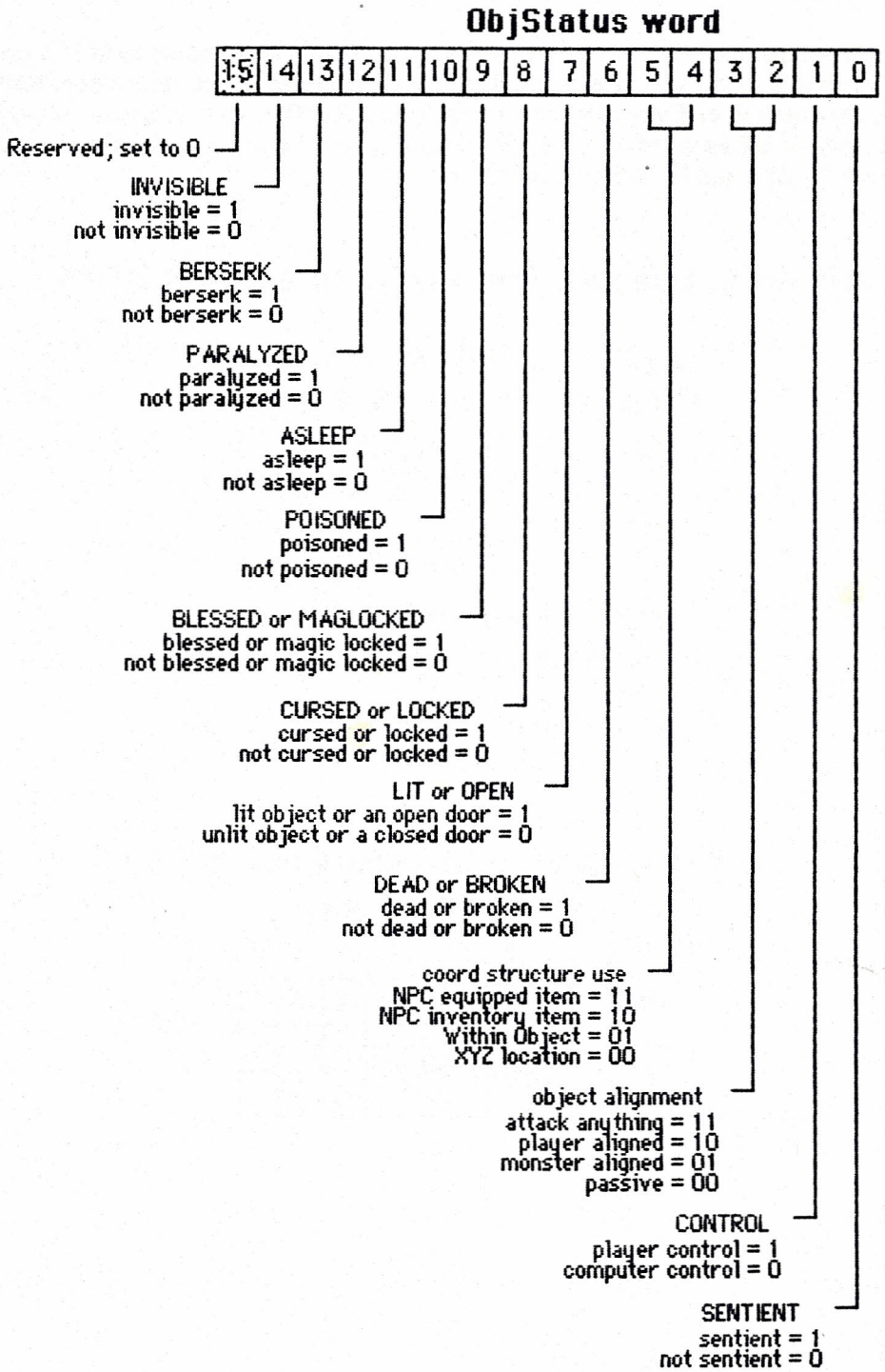


Ultima VI

Design Notes Collection - Technical Design

ABOUT **ObjStatus**

ObjStatus is a series of flags, with each bit or combination of bits describing the condition of an object. The diagram describes their locations and functions.



Ultima VI

Design Notes Collection - Technical Design

ABOUT **struct coord ObjPos**

ObjPos is actually 3 bytes that act as if they were a union of four structures.They are not a union in the C Language sense because they are not declared as such. A diagram of its four uses is detailed below.

ObjPos has 4 functions, In most cases it will register the **X-Y-Z location** of an object. But in other cases, the object will be associated with another object. These associations may be **Within**, **Inventory**, or **Equipped**. In such instances, **ObjPos** uses its two lower bytes to act as a pointer to the associated object. Theoretically, such a system would allow nesting of associations. In ***Ultima VI*** this will not occur.

Inventory, Equipped, and Within type struct coord

Byte Offset	Bit Positions 7	6	5	4	3	2	1	0
+0								
+1								
+2								

Pointer to Associated Object

Unused Bits

X-Y-Z Location structure coord

Byte Offset	Bit Positions 7	6	5	4	3	2	1	0
+0	x	x	x	x	x	x	x	x
+1	y	y	y	y	y	y	x	x
+2	z	z	z	z	y	y	y	y

Ultima VI

Design Notes Collection - Technical Design

ABOUT **struct shapetype ObjShapeType**

ObjShapeType contains information about the type of object there is and it current on-screen shape. Both of these pieces of information take 12 bits to describe, so three bytes have been allocated to their use.

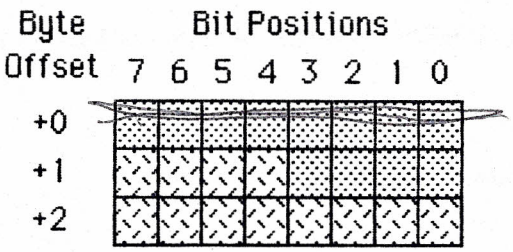
~~GetShapeType~~

GetObjType

GetFrameNum

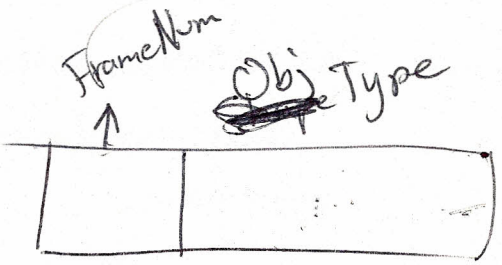
GetShapeNum

struct shapetype



Shape Number Showing

Shape Type in Actuality



ABOUT **Amount**

Amount is a 2-byte value that serves a double purpose. In cases where the **Quality** of an item is not of particular use, ie. Food and Gold, both bytes are used as a counter. In most other cases the bytes are divided up into **Quality** and **Quantity** bytes.

Quantity indicates how many of the given object is available. The range is 0-255.

Quality serves a unique purpose for each type of Object. For example, the **Quality** of a key could refer to what door it will open. A simple test of the **Quality** of a door against the **Quality** of a key will determine if that key opens the door. A **Quality** associated with an Object called 'Magic Spell' would indicate which magic spell it was.

ABOUT **Strength/Wgt**

Strength/Wgt is a 1-byte value that is composed of two nibbles of data. The low nibble contains the actual strength value of the Object, The high nibble contains the weight it may carry.

ABOUT **Dex/Mov**

Dex/Mov is a 1-byte value that is composed of two nibbles of data. The low nibble contains the actual dexterity value of the Object, The high nibble contains the movement points it has left.

Ultima VI

Design Notes Collection - Technical Design

ABOUT **Intell/Mag**

Intell/Mag is a 1-byte value that is composed of two nibbles of data. The low nibble contains the actual intelligence value of the Object, The high nibble contains the Magic points available for use.

ABOUT **StrExp, DexExp, and IntExp**

These a 2-byte values keep track of the amount of experience gained in each statistic.

ABOUT **HitPts**

Contains a record of how many hit points the Object has left.

ABOUT **Schedule**

Schedule is still not developed at this time.

ABOUT **NPCFlags**

NPCFlags contains extra information regarding an NPC. Here it will be determined what kind of movement mode the Object is in. Follow mode can now be defined as **Adjacent** an **Nearby** to help the NPC Tracker move multi-tile objects. **Drunk** and **Fleeing** conditions can also be noted here.

ABOUT **Leader**

Leader is a pointer to the Leader Object. This is especially useful to the NPC Tracker which must move Objects in a particular direction based on the Following Mode. In a multi-tile monster, the Leader will be considered to be the head. All Objects associated with the creature will have this location filled with the Number of the head.

Ultima VI

Design Notes Collection - Technical Design

IBM OBJECT DATA ORGANIZATION

As mentioned above, the data structure organization in IBM memory is actually a series of parallel arrays. Some of these arrays are to be accessed only by macros, others by direct manipulation of the arrays. A complete Listing of those arrays and their description and accessing methods are listed below.

FUNCTIONS AND MACROS AVAILABLE

The following list of functions and macros are designed to properly interface with the Object Data Structure. These routines will be the only interface method accepted. Since Macros are also discussed here, it should be pointed out that Macros will be divided into **safe** and **unsafe** categories. All macros are to be considered **unsafe** unless stated otherwise.

ARRAYS AVAILABLE FOR GENERAL ACCESSING

These arrays may be manipulated directly in the standard fashions. **DO NOT** access the other arrays in any other fashion, other than the macros given.

unsigned int	**Amount**[ListSize];	same memory as **Quality/Quantity** bytes. Use this **ONLY** when **Quality** and **Quantity** are not important. Otherwise, use the macros to access this data array. At this writing you may use **Amount** when you are manipulating **Food** and **Gold**.
unsigned int	**StrExp**[256];	found in extended structure
unsigned int	**DexExp**[256];	found in extended structure
unsigned int	**IntExp**[256];	found in extended structure
unsigned char	**HitPts**[256];	found in extended structure
unsigned char	**Leader**[256];	found in extended structure

MACROS WHICH MANIPULATE **Amount** IN **Quality/Quantity** FORMAT

These macros place values into each byte as defined in the structure of **Amount**.

char	**GetQual**(ObjectNum);	
char	**GetQuan**(ObjectNum);	
char	**SetQual**(ObjectNum, Value);	returns Value given
char	**SetQuan**(ObjectNum, Value);	returns Value given
char	**AddQual**(ObjectNum, Value);	returns new Quality amount
char	**AddQuan**(ObjectNum, Value);	returns new Quantity amount
char	**SubQual**(ObjectNum, Value);	returns new Quality amount
char	**SubQuan**(ObjectNum, Value);	returns new Quantity amount
char	**QualQuan**(Quality, Quantity);	Compresses Quality and Quantity into an integer to place into array **Amount**.

Ultima VI

Design Notes Collection - Technical Design

MACROS WHICH TEST **ObjStatus** BITS

These macros are considered **safe**. Each macro returns a TRUE or FALSE result depending on whether the bit is set or clear. In general, if a bit is set, the result is TRUE. See fig. 4 for a complete listing of the bit fields being tested. Note that 4 bit fields have overlapping functions. The macro list below places those macros which operate on the same bits on the same line.

These macros test the simple Bit fields in **ObjStatus**.

int **IsPlrControl**(ObjectNum);	player or computer controlled object?	
int **IsSentient**(ObjectNum);		
int **IsBroken**(ObjectNum);	int **IsDead**(ObjectNum);	
int **IsLit**(ObjectNum);	int **IsOpen**(ObjectNum);	
int **IsCursed**(ObjectNum);	int **IsLocked**(ObjectNum);	
int **IsBlessed**(ObjectNum);	int **IsMagLocked**(ObjectNum);	
int **IsPoisoned**(ObjectNum);		
int **IsAsleep**(ObjectNum);		
int **IsParalyzed**(ObjectNum);		
int **IsBerserk**(ObjectNum);		
int **IsInvisible**(ObjectNum);		

These macros test the alignment of an object

int **IsNeutral**(ObjectNum);	is alignment passive?
int **IsGood**(ObjectNum);	is it player aligned?
int **IsEvil**(ObjectNum);	is it monster aligned
int **IsChaotic**(ObjectNum);	is it an enemy to everything?

These macros determine how struct coord is to be used. Also determines if an item is equipped or not.

int **IsXYZ**(ObjectNum);	used as an XYZ locator?
int **IsInObj**(ObjectNum);	used as an association to an object?
int **IsEquipped**(ObjectNum);	used to tell to which NPC it is equipped?
int **IsUnEquip**(ObjectNum);	used as an NPC inventory but not equipped?
int **IsInven**(ObjectNum);	used as an NPC inventory equipped or not?

Ultima VI

Design Notes Collection - Technical Design

MACROS WHICH MANIPULATE **ObjStatus** BITS

The following macros are **safe** to use. **Set**...() sets a status bit to 1, **Clr**...() returns a bit to 0. Those macros which operate on the same status bits but have differing names are preceeded by similar markers.

```
   void SetSentient(ObjectNum);        void ClrSentient(ObjectNum);
   void SetPlrControl(ObjectNum);      void ClrPlrControl(ObjectNum);
1  void SetBroken(ObjectNum);          void ClrBroken(ObjectNum);
1  void SetDead(ObjectNum);            void ClrDead(ObjectNum);
2  void SetLit(ObjectNum);             void ClrLit(ObjectNum);
2  void SetOpen(ObjectNum);            void ClrOpen(ObjectNum);
3  void SetCursed(ObjectNum);          void ClrCursed(ObjectNum);
3  void SetLocked(ObjectNum);          void ClrLocked(ObjectNum);
4  void SetBlessed(ObjectNum);         void ClrBlessed(ObjectNum);
4  void SetMagLocked(ObjectNum);       void ClrMagLocked(ObjectNum);
   void SetPoisoned(ObjectNum);        void ClrPoisoned(ObjectNum);
   void SetAsleep(ObjectNum);          void ClrAsleep(ObjectNum);
   void SetParalyzed(ObjectNum);       void ClrParalyzed(ObjectNum);
   void SetBerserk(ObjectNum);         void ClrBerserk(ObjectNum);
   void SetInvisible(ObjectNum);       void ClrInvisible(ObjectNum);
```

SPECIAL MACROS FOR **ObjStatus**

The two **Get**...() macros return the full values of their combination bits. The **Set**...() macro must be passed the comparisons listed to work correctly. The **Get**...() macros are **safe**. The **Set**...() macro is **unsafe**.

```
void  SetAlignment(ObjectNum, Alignment);
int   GetAlignment(ObjectNum);
int   GetCoordUse(ObjectNum);   Macro to determine how struct coord is used.
```

Return Values from **GetCoordUse**()	LOCXYZ	acts as XYZ location
	CONTAINED	pointer to object it is in
	INVEN	pointer to NPC it belongs to
	EQUIP	pointer to NPC it is equipped on.
Alignments are:	NEUTRAL	passive alignment
	EVIL	monster aligned
	GOOD	player aligned
	CHAOTIC	attack everything

Ultima VI

Design Notes Collection - Technical Design

MACROS WHICH MANIPULATE STRUCT **coord ObjPos**

The **Get**...() macros are **safe**.

int	**GetX**(ObjectNum);	returns X coordinate of an object
int	**GetY**(ObjectNum);	returns Y coordinate of an object
int	**GetZ**(ObjectNum);	returns Z coordinate of an object
int	**GetAssoc**(ObjectNum);	returns pointer number when struct coord is used as an inventory, equipped, or within type.

MACROS WHICH MANIPULATE STRUCT **coord**

Macros used on struct **coord** defined in a function:

int	**GetCoordX**(StructName);	returns X coordinate of an object
int	**GetCoordY**(StructName);	returns Y coordinate of an object
int	**GetCoordZ**(StructName);	returns Z coordinate of an object
int	**GetCoordAssoc**(StructName);	returns pointer number when struct coord is used as an inventory, equipped, or within type.

void	**SetCoordX**(StructName, X);	sets X coordinate of an object
void	**SetCoordY**(StructName, Y);	sets Y coordinate of an object
void	**SetCoordZ**(StructName, Z);	sets Z coordinate of an object
void	**SetCoordXY**(StructName, X, Y);	sets X and Y coordinates of an object
void	**SetCoordXYZ**(StructName, X, Y, Z);	sets X, Y, and Z coordinates of an object
void	**SetCoordAssoc**(StructName, Value);	sets pointer number when struct coord is used as an inventory, equipped, or within type.

Ultima VI

Design Notes Collection - Technical Design

MACROS THAT OPERATE ON STRUCT **shapetype ObjShapeType**

These macros operate with struct shapetype in the object list. **Get**...() macros are **safe**. **Set**...() macros are **unsafe**.

int	**GetShape**(ObjectNum);	returns Shape Number of Object
int	**GetType**(ObjectNum);	returns Shape Type of Object
void	**SetShape**(ObjectNum, ShapeNum);	sets the Shape Number of an Object.
void	**SetType**(ObjectNum, ShapeType);	sets the Shape Type of an Object.

MACROS WHICH MANIPULATE THE EXTENDED PORTION OF OBJECTS

The first 256 Objects are slightly different than other objects in that their structure is the extended type found on page 3. The following macros deal with the extra fields shown.

int	**GetStr**(ObjectNum);	returns Object's Strength;
int	**GetDex**(ObjectNum);	returns Object's Dexterity;
int	**GetInt**(ObjectNum);	returns Object's Intelligence;
int	**GetWgt**(ObjectNum);	returns Object's Wgt;
int	**GetMov**(ObjectNum);	returns Object's movement value;
int	**GetMag**(ObjectNum);	returns Object's Magic Points;
void	**SetStr**(ObjectNum, NewStr);	sets Object's Strength;
void	**SetDex**(ObjectNum, NewDex);	sets Object's Dexterity;
void	**SetInt**(ObjectNum, NewInt);	sets Object's Intelligence;
void	**SetWgt**(ObjectNum, NewWgt);	sets Object's Wgt;
void	**SetMov**(ObjectNum, NewMov);	sets Object's movement value;
void	**SetMag**(ObjectNum, NewMag);	sets Object's Magic Points;

These Macros do not do bounds checking at all. They wrap around

void	**AddStr**(ObjectNum, add);	adds to Object's Strength;
void	**AddDex**(ObjectNum, add);	adds to Object's Dexterity;
void	**AddInt**(ObjectNum, add);	adds to Object's Intelligence;
void	**AddWgt**(ObjectNum, add);	adds to Object's Wgt;
void	**AddMov**(ObjectNum, add);	adds to Object's movement value;
void	**AddMag**(ObjectNum, add);	adds to Object's Magic Points;

These Macros do not do bounds checking at all. They wrap around

void	**SubStr**(ObjectNum, sub);	subtracts from Object's Strength;
void	**SubDex**(ObjectNum, sub);	subtracts from Object's Dexterity;
void	**SubInt**(ObjectNum, sub);	subtracts from Object's Intelligence;
void	**SubWgt**(ObjectNum, sub);	subtracts from Object's Wgt;
void	**SubMov**(ObjectNum, sub);	subtracts from Object's movement value;
void	**SubMag**(ObjectNum, sub);	subtracts from Object's Magic Points;

FUNCTIONS THAT ARE ASSOCIATED WITH THE OBJECT LIST

```
int     MaxHP(NPCNum)
int     NPCNum;
```

Returns the maximum hit points an Object can have. This is useful in determining how many hit points to heal, or possibly, how much damage can be given.

```
int     Level(NPCNum)
int     NPCNum;
```

Returns the current level of the NPC in Question. This function does not check to see that it is accessing a proper NPC (ie. accessing the tail of a multi-tile creature).

```
int     StrLevel(NPCNum)
int     NPCNum;
```

Returns Current Level based on Strength Experience points.

```
int     DexLevel(NPCNum)
int     NPCNum;
```

Returns Current Level based on Dexterity Experience points.

```
int     IntLevel(NPCNum)
int     NPCNum;
```

Returns Current Level based on Intelligence Experience points.

```
int     FindLoc(x, y, z)
int     x, y, z;
```

This is a scan routine that looks through the object list to find the first Object of type LOCXYZ at Location X,Y,Z. If no Object is found returns a negative result, otherwise, the value is the index into the Object List for the given object.

```
int     NextLoc();
```

Continues a search started by **FindLoc()**. Returns the same information.

```
int     FindInv(ObjectNum)
int     ObjectNum;
```

Searches for any objects of type CONTAINED, INVEN, or EQUIP associated with **ObjectNum**. Returns the first Object found, or a negative value if no object is associated with the given **ObjectNum**.

```
int     NextInv();
```

Continues a search started by **FindInv**(). Returns the same information.

```
int     SearchArea(X1, Y1, X2, Y2)
int     X1, Y1, X2, Y2;
```

This is a scan routine that looks through the object list to find the first Object located within the region bounded by the rectange. No Z coordinate is needed since this routine does all searches by using global variable **mapz** If no Object is found, a negative result is returned, otherwise, the value is the index into the Object List for the given object. This routine saves some internal variables for use by other functions.

```
int     NextArea();
```

Continues a search started by **SearchArea**(). Returns the same information.

```
int     FindInvType(ObjNum, ObjType, Quality)
int     ObjNum, ObjType, Quality;
```

Searches for an Object of **Quality** and **ObjType** associated with **ObjNum**. **ObjType** is the actual type of item such as a SWORD, KEY, ARROW, FOOD, etc. An associated object is of type CONTAINED, INVEN, or EQUIP. A negative **Quality** parameterpassed ignores quality checking. If no such object is found, a negative result is returned.

```
int     AddObj(Status, Location, ShapeType, ShapeNum, Amount)
int     Status;
struct  coord Location;
int     ShapeType, ShapeNum, Amount;
```

Adds an Object to the Object List. returns a negative result if placement failed. **Status** is the Status Bits sent. **Location** is the structure **coord** which may contain X,Y,Z coordinates or a pointer to an object. **ShapeType** is the Object Type and **ShapeNum** is the tile number to display. **Amount** corresponds to the number of that item available or its **Quality/Quantity** values. To place **Quality/Quantity** values, convert them to **Amount** values with the **QualQuan**() macro.

```
int     DeleteObj(ObjectNum)
int     ObjectNum;
```

Deletes an Object from the Object List.

```
int     MoveObj(ObjectNum, x, y, z)
int     ObjectNum, x, y, z;
```

Moves **ObjectNum** from anywhere (either in an inventory or on the map) to a location on the map. This could be particularly useful for dropping items, moving things around, and teleporting. At this writing, teleporting can only happen to Player's characters. Later versions might support more sophisticated handling of Objects. A negative return value indicates failure.

```
int     InsertObj(ObjNum, DestObj, Placement)
int     ObjNum, DestObj, Placement;
```

Places **ObjNum** into **DestObj**. **Placement** is a modifier which determines **ObjNum**'s association to **DestObj**. **Placement**'s values are CONTAINED, INVEN, or EQUIP. A negative return value indicates a failure to insert.

```
int     GiveObj(Owner, ObjType, Amount)
int     Owner, ObjType, Amount;
```

Add to the **Quantity** of **ObjType** (owned or contained by **Owner**) by the **Amount** given. **Owner** may be a monster, a player, or even a chest. A failure returns a negative result. If the Object does not exist in **Owner**'s inventory, then it is added. Objects that use **Amount**, rather than **Quality/Quantity** will be handled properly. If **Quality** and **Quantity** need to be passed, compress them using the **QualQuan**() macro.

```
int     TakeObj(Owner, ObjType, Amount)
int     Owner, ObjType, Amount;
```

Subtract **Quantity** of **ObjType** (owned or contained by **Owner**) by the **Amount** given. **Owner** may be a monster, a player, or even a chest. A failure returns a negative result. If the quantity of an existing Object is reduced to a negative level, it will be deleted. Objects that use **Amount**, rather than **Quality/Quantity** will be handled properly. If **Quality** and **Quantity** need to be passed, compress them using the **QualQuan**() macro.

```
int     TransferObj(OldOwner, NewOwner, ObjType, Amount)
int     OldOwner, NewOwner, ObjType, Amount;
```

Transfer the **Quantity** of **ObjType** (owned or contained by **OldOwner**) by **Amount** to **NewOwner**. Owners may be monsters, players, or even chests. A failure returns a negative result. Objects are created and deleted as necessary. Objects that use **Amount**, rather than **Quality/Quantity** will be handled properly. If **Quality** and **Quantity** need to be passed, compress them using the **QualQuan**() macro.

```
int     Encumbrance(ObjNum)
int     ObjNum;
```

Find out how much weight is contained within **ObjNum**. It does not include the weight of **ObjNum**. Thus, with this routine, a person can find out how much weight is in a backpack or sack, or how much weight a man is lifting.

```
int     ClearSpace(Number,x,y)
int     Number, x, y;
```

Clears out space in the Object List for **Number** of Objects at location **x,y**. This routine returns a negative result if operation failed. Otherwise, the index to the first clear space is returned.

```
int     NextSpace();
```

returns index of the next Object Slot created by the **ClearSpace()** Function. returns a negative result if the end of the list is reached.

```
int     CreateMonster(MonsterType, x, y, z)
int     MonsterType, x, y, z;
```

Creates a new monster of the given type and adds it into the NPC portion of the Object List. The number returned is the index into this list where the monster was placed. A negative value is returned if the creation was a failure.

```
int     LoadNewRegions(x,y)
int     x,y;
```

Loads new Object Regions into the Object List. DO NOT CALL THIS FUNCTION. This is written only to inform the staff of its use. This function will produce changes to the object list which can produce bugs in your code. At this writing, a call to **NextSpace()**, **NextLoc()**, or any other **Next...()** type function. will be invalid after a player has walked off into a new region. In addition, **ScratchBuff[]** is used by this function. Be careful when using this area of memory, especially when possible Object rearrangement can happen.