

## **Halo Player and Object Management**

Made By: Oxide aka Genocide, urbanyoung.

## **A Bit of Background Information:**

Some of you may not be familiar with how a computer works (from a software side). It is obviously very complex and so I will just give you the basics (as I understand them).

Computers have an operating system which interfaces with the hardware. Software developers don't really need to worry about this. The operating system manages all programs that run on the computer. Each program is assigned some memory for it to use. A program has two main sections of memory. There is the code section that contains all the instructions the program uses, and then there's the data section. From the point of view of a Phasor script, only the data section is relevant. Memory, of course, needs to be referenced in a way that allows access. This is done through a *memory address*. A memory address refers to a specific part of memory within the program.

When Halo loads it reads data from its files and stores it in memory. This data is used to control the game. It specifies where players spawn, where the guns, vehicles and scenery are. You get the idea. Now, what happens if we modify this memory?

## How Halo Structures Data:

Halo loads data from various map files; it loads textures, shaders, object properties, spawns, vehicles etc. The data is referenced using “tags”. A “tag” has two basic parts: the type and the name. Let’s look at a shotgun for example. Its tag type is ‘weap’ which, as you can guess, specifies that it’s a weapon. Its tag name is ‘weapons\shotgun\shotgun’ which identifies the unique item within the tag type. When Halo searches for something it first locates the type data (in this case the data for ‘weap’) and then searches it for the specific data ‘weapons\shotgun\shotgun’. Every object can be specified by a tag type and name.

Here’s the structure of a tag entry:

```
// Structure of the tag header
struct hTagHeader
{
    DWORD tagType; // ie weap
    DWORD unk[2]; // I don't know
    DWORD mapid; // unique id
    char* tagName; // name of tag
    LPBYTE metaData; // data for this tag
    DWORD unk1[2]; // I don't know
};
```

As you can see a tag entry has a bit more data than just the name and its type. What I want to mention here is the id. Within a map each tag is assigned a unique id, nothing else within that map has the same id. However, this id isn’t consistent across different maps; this is why we need the tag type/name too. Internally, Halo generally operates with the mapid when referencing an object. Because of this, Phasor does too. Phasor lets you find the mapid of an object with the `lookupTag` function. It takes the tag type, and its name, and returns the mapid of that tag. This can be used for specifying which weapon to assign a player.

Every shotgun in a map has the same map id. That’s good for generically identifying all shotguns, but how would we identify a particular shotgun? Halo obviously has a system for this. It uses what I call *object ids*. When loading data from a map it finds the generic information (tag) that contains all the data needed to create an object. When creating the actual object it assigns them a unique object id. Much like how a mapid is for one tag, an object id is for one object. Given an object id we can find that object’s specific data (or memory). Again, Phasor handles this for you with the `getObject` function. *Given an object id, getObject will return the location of that object’s memory.*

## How Halo Manages Players:

That's the basics of how objects are managed, but what about players? How do they work? Well, as you know Halo has a hardcoded 16 player limit. This means that every possible player can be identified by a number, 1 – 16 (in reality 0-15 but we'll get into that later). Each player has a 0x200 (512) byte region of memory that stores player specific data. This includes things like the player's name, their team, location and speed. If we want to modify this data we, again, need to find the memory address. Phasor can handle this for you with the *getplayer* function. Given a *player's memory id* *getplayer* will return the location of that player's memory.

It's important to note that Halo has two very similar ways of identifying a player. The first and most common way is through the use of *memory ids*. A memory id is a number ranging from 0 to 15 that identifies a player. This is the method Phasor uses internally. The second method is through the use of *player or rcon ids*. This is the id that's displayed when you use *sv\_players*. You can think of it as the *user id* as it's really only associated with user inputs (ie using server commands). The *memory id* is usually the same as the *player id*, but not always. As such Phasor needs to provide a way to convert from one to the other. To convert from a *memory id* to a *player id* you can use *resolveplayer* and for player to memory *rresolveplayer*. This is important when processing user inputs like server commands.

Now that you know how players are identified we can dig a bit deeper. I explained what an object is in the previous section; make sure you're familiar with it. *A player owns an object but an object does not own a player*. This means that every player also has an object. This object is identified, as you'd expect, through *memory ids*. The player's object id is contained within the player structure (the 0x200 byte allocation of memory) and is at offset 0x34. *An offset is just the difference between the memory base (ie player structure) and the data we want*. As we know where the player's memory id is located, we can use *getobject* to find the player's object.

The player's object contains a lot of information (some of it duplicated in the player structure). It has things like the player's weapons, shields, location, walk speed (and much more). Before I go further I should make something clear. The player structure isn't always the real data; the object data is. For example, the x/y/z locations in the player structure are where the client thinks they are. The x/y/z locations in the object data is where they actually are. As such you can think of the object data as completely server controlled, whereas the player structure isn't. This isn't too important until we get into wanting to move an object.

## Practical Uses of this Information:

By now I hope you have a basic understanding of how Halo manages data. This information is the basis of some of the stuff Phasor does. Things like changing weapons, getting weapon info, making a player invisible, 'teleporting' objects, modifying damage, changing a player's speed all derive from this basic understanding.

Let's analyze one of the functions in my Infection script. I'll use one of the more complicated ones, `OnObjectInteraction`. I won't paste the whole function in one go as it's quite large. You should have a reference copy open though so you can see it all together.

```
-- regardless of the team, stop people taking the flag
if tagType == "weap" then
    if tagName == "weapons\\flag\\flag" then
        response = 0
    end
end
```

The above code stops a player from taking the flag in Capture the Flag. It should seem quite familiar to you now. First, it checks the tag type to see if it's a weapon. It then checks to see if the tag name matches that of the flag. If it does, it sets response to 0. If you've read the scripting guide/the rest of the function, you'll see that making response 0 blocks the object interaction. As such, by checking the tag data I was able to stop people picking up flags. Moving on.

```
elseif tagName == "powerups\\active camouflage" then
    -- check if the picking player is already invisible
    local m_player = getplayer(player)

    if m_player ~= 0 then

        local m_playerObjId = readdword(m_player, 0x34)

        local m_object = getobject(m_playerObjId)
```

This code again makes use of the tag name, checking to see if we're interacting with active camouflage. If so it will find the player's memory using `getplayer`. It then uses the `readdword` function (see Phasor Scripting Guide) to read the data at `m_player + 0x34`. As I mentioned earlier, this is the player's object's memory id. This is then used in a `getobject` call to get the memory address of the object. If we keep looking we see:

```
if m_object ~= 0 then
    local camoFlag = readdword(m_object, 0x204)

    if camoFlag ~= 0x51 then
        ...
    end
```

It then reads the data at `m_object + 0x204`. This is the offset for the `camoFlag`, which is a value specifying whether or not the player is currently camouflaged. If it doesn't equal 0x51 the player isn't camouflaged. I've had many people ask what this bit of code does, but when you understand the basics of it it's really quite, well, basic. The scripts I've made use a bit of

memory modification, which is all this is. I find the object and then I find the data. You can also see this with the OnWeaponCreation function.

```
writeword(m_weapon, 0x2B6, 600)  
--writeword(m_weapon, 0x2B8, 1600)
```

I simply write a value (600) to memory and tada, the weapon's ammunition has changed. The hardest part about this is finding the offsets.

<http://haloapps.wordpress.com>

## Offsets:

Finding the right offsets for the right data is the most difficult part of using the scripting system well. I don't expect many of you to be able to find them. I had to analyze how Halo runs to find many of them. However, quite a few people have messed around with Halo before. You can use what they found out.

You may or may not be aware of map modding. Basically you change some data within the map files and that changes how the map appears (weapons can have different properties; you can have new objects etc). This is easy enough to do as there are two main programs for it. HMT (Halo Map Tools) and Eschaton. Both of these programs use plugins which are in the XML format, which you can view in a text editor like Notepad. These plugins include the offsets the data's out. Unfortunately, the lowest unit Phasor can edit is a byte (which is 8 bits) and so Phasor can't edit all of the data that map tools can. However, it can do a fair bit. I'll give you an example of finding the offset.

```
<type>float</type>
  <offset>0x1D4</offset>
  <name>Maximum Damage (Min)</name>
  <info>Maximum damage caused (minimum of the max range)</info>
</value>
```

This is an excerpt from HMT's weap plugin. As you can see, the offset is surrounded by <offset></offset>. There is also a description of what it does below, and its type above. If you look at my Infection script, OnDamageLookup you'll see:

```
local max_dmg_min = readfloat(tagdata, 0x1D4)
```

As you can see, I use the same offset in my code and use the float data type. You can do this for other tags and values, there are many plugins (look at Eschaton ones too!).

Here's a list of the offsets I've found. I had many more but can't find where they got saved.

```
struct vect3d
{
    float x;
    float y;
    float z;
};

// Some structure issues were clarified thanks to the code at:
// http://code.google.com/p/halo-
// devkit/source/browse/trunk/halo_sdk/Engine/objects.h
struct G_Object // generic object header
{
    DWORD mapId; // 0x0000
    long unk[3]; // 0x0004
    char unkBits : 2; // 0x0010
    bool ignoreGravity : 1;
    char unk1 : 4;
    bool noCollision : 1; // has no collision box, projectiles etc pass
    right through
    char unkBits1[3];
    unsigned long timer; // 0x0014
    char empty[0x44]; // 0x0018
    vect3d location; // 0x005c
    vect3d velocity; // 0x0068
    vect3d rotation; // 0x0074 (not sure why this is used, doesn't yaw do
    orientation?)
    vect3d axial; // 0x0080 (yaw, pitch, roll)
    vect3d unkVector; // 0x008C (not sure, i let server deal with it)
    char unkChunk[0x28]; // 0x0098
    unsigned long ownerPlayer; // 0x00c0 (index of owner (if has one))
    unsigned long ownerObject; // 0x00c4 (object id of owner, if
    projectile is player id)
    char unkChunk1[0x18]; // 0x00c8
    float health; // 0x00e0
    float shield; // 0x00e4
    char unkChunk2[0x10]; // 0x00e8
    vect3d location1; // 0x00f8 set when in a vehicle unlike other one.
    best not to use tho (isnt always set)
    char unkChunk3[0x10]; // 0x0104
    unsigned long veh_weaponId; // 0x0114
    unsigned long player_curWeapon; // 0x0118
    unsigned long vehicleId; // 0x011c
    BYTE bGunner; // 0x0120
    short unkShort; // 0x0121
    BYTE bFlashlight; // 0x0123
    long unkLong; // 0x0124
    float shield1; // 0x0128 (same as other shield)
    float flashlightCharge; // 0x012C (1.0 when on)
    long unkLong1; // 0x0130
    float flashlightCharge1; // 0x0134
    long unkChunk4[0x2f]; // 0x0138
};

struct G_Biped
{
    char unkChunk[0x10]; // 0x1F4
    long invisible; // 0x204 (0x41 inactive, 0x51 active. probably
    bitfield but never seen anything else referenced)
```



```

    struct aFlags // these are action flags, basically client button
    presses
    { //these don't actually control whether or not an event occurs
        bool crouching : 1; // 0x0208 (a few of these bit flags are
thanks to halo devkit)
        bool jumping : 1; // 2
        char UnknownBit : 2; // 3
        bool Flashlight : 1; // 5
        bool UnknownBit2 : 1; // 6
        bool actionPress : 1; // 7 think this is just when they
initially press the action button
        bool melee : 1; // 8
        char UnknownBit3 : 2; // 9
        bool reload : 1; // 11
        bool primaryWeaponFire : 1; // 12 right mouse
        bool secondaryWeaponFire : 1; // 13 left mouse
        bool secondaryWeaponFire1 : 1; // 14
        bool actionHold : 1; // 15 holding action button
        char UnknownBit4 : 1; // 16
    } actionFlags;
    char unkChunk1[0x26]; // 0x020A
    unsigned long cameraX; // 0x0230
    unsigned long cameraY; // 0x0234
    char unkChunk2[0x6f]; // 0x238
    BYTE bodyState; // 0x2A7 (2 = standing, 3 = crouching, 0xff =
invalid, like in vehicle)
    char unkChunk3[0x50]; // 0x2A8
    unsigned long primaryWeaponId; // 0x2F8
    unsigned long secondaryWeaponId; // 0x2FC
    unsigned long tertiaryWeaponId; // 0x300
    unsigned long ternaryWeaponId; // 0x304
    char unkChunk4[0x18]; // 0x308
    BYTE zoomLevel; // 0x320 (0xff - no zoom, 0 - 1 zoom, 1 - 2 zoom etc)
    BYTE zoomLevel1; // 0x321
    char unkChunk5[0x1AA]; // 0x322
    BYTE isAirbourne; // 0x4CC (only when not in vehicle)
};

// Partial structure of the player structure
struct hPlayerStructure
{
    WORD playerJoinCount; // 0x0000
    WORD localClient; // 0x0002 always FF FF on a dedi in Halo is
00 00 if its your player
    wchar_t playerName[12]; //0x0004
    DWORD unk; // 0x001C only seen FF FF FF FF
    DWORD team; // 0x0020
    DWORD m_interactionObject; // 0x0024 ie Press E to enter THIS
vehicle
    WORD interactionType; // 0x0028 8 for vehicle, 7 for weapon
    WORD interactionSpecifier; // 0x002A which seat of car etc
    DWORD respawnTimer; // 0x002c
    DWORD unk2; // 0x0030 only seen empty
    DWORD m_playerObjectid; // 0x0034
    DWORD m_oldObjectid; // 0x0038
    DWORD unkCounter; // 0x003C sometimes changes as you move, fuck
idk
    DWORD empty; // 0x0040 always FF FF FF FF, never accessed
    DWORD bulletUnk; // 0x0044 changes when the player shoots
    wchar_t playerNameAgain[12]; // 0x0048

```

```

        WORD playerNum_NotUsed; // 0x0060 seems to be the player
number.. never seen it accessed tho
        WORD empty1; // 0x0062 byte allignment
        BYTE playerNum; // 0x0064 player number used for rcon etc (ofc
this is 0 based tho)
        BYTE unk_PlayerNumberHigh; // 0x0065 just a guess
        BYTE team_Again; // 0x0066
        BYTE unk3; // 0x0067 idk, probably something to do with player
numbers
        DWORD unk4; // 0x0068 only seen 0, it wont always be 0 tho
        float speed; // 0x006C
        DWORD unk5[11]; // 0x0070 16 bytes of FF? then 4 0, some data,
rest 0... meh idk about rest either
        WORD kills; // 0x009C
        WORD idk; // 0x009E byte allignment?
        DWORD unk6; // 0x00A0 maybe to do with scoring, not sure
        WORD assists; // 0x00A4
        WORD unk7; // 0x00A6 idk byte allignment?
        DWORD unk8; // 0x00A8
        WORD betrayals; // 0x00AC
        WORD deaths; // 0x00AE
        WORD suicides; // 0x00B0
        // cbf with the rest
        BYTE rest[0x14E]; // 0x00B2
};

```