Wog T1 – Map objects, game memory

By Jakub Grzana
latest update: 05.08.2020

About this document:

Here I will describe (in a over-simplified way) basics of computer memory and how you can access Heroes3 game and map data, as well as tools provided by WoG project. This is distributed in the hope that it will be useful, but without any warranty. Keep in mind I might have fucked up.

Basics of computer memory:

Memory computer is built from cells known as 'bits'. Each bit can take one of two states: 0 or 1. Not much, yeah? Well, this is why singular bits are rarely used.

Usually to store data we use bytes (8bit array) that can store 256 states. How? Well, lets build byte from scratch and see how each additional bit increase number of possible states.

Note: sequence matter
```
1bit - 2 states (0,1)
2bit - 4 states (00,01,10,11)
3bit - 8 states (000,001,010,011,100,101,110,111)
```
And so on.

Each bit multiplies possible combinations by 2. For n bits, you have 2^n sequences. Each sequence is unique, so it stores unique information. Some very smart ppl figured out you can store numerical values in bytes this way. This is known as 'binary numeral system'.
```
0b0001 = 1
0b0010 = 2
0b0011 = 3
0b0100 = 4
```
0b means it's binary number, 0x means it's hexadecimal number. If you don't understand what's going on, read more about these topics in internet. Also check out 'little endian' and 'big endian'.

So, we can store 2^n information in n-bits, thus 8-bits can store numerical value from 0 to 255 ($2^8 - 1$, one of combinations is 0) You can use 1 bit to determine sign (0 - positive, 1 negative) thus in 8 bits you could store signed values from -128 to 127.

Byte can be used to store not only numerical values though. You can use every bit to store separate flag. Lets say every object in the game has 1 byte in memory that stores information "who visited this object". Example: `0b01101010`
This would mean that player blue, green, purple and teal visited this object. Decimal value of data stored can be check but doesn't bother us. All we do here is to set and unset bits corresponding to specified players.

You can combine 2 bytes to store values 0 to over 65535. I mentioned that "bit is cell" of memory. That is correct from software perspective, but physically it's not truth and memory is build using bytes.

So you can see that bits can be gathered into groups (bytes) to store large amount of data, and this data can be interpreted in various ways. Same goes for whole memory.

Game memory is a giant (one-dimensional) array of bytes. Some part is executable code and functions, another map data, yet another is creature data loaded from .txt files. Everything is stored using bytes and using special tools you can access and modify it on will.

Note that you can use only parts of memory that were allocated by h3 exec and wog addition. Accessing unallocated memory, or memory allocated by other program (or operating system) will usually cause segmentation fault, a very hard to track down error.

You can find addresses of data and functions using debugger, for example 'x32dbg'. There will be separate document for reverse engineering h3.

Here is example of Heroes 3 resource archive (LOD file) opened with Hex Editor (HxD) Each 2 hex values stands for 1 byte. To the right you can see additional conversion to ASCII characters. Note the Offset – *"In computer science, an offset within an array or other data structure object is an integer indicating the distance (displacement) between the beginning of the object and a given element or point, presumably within the same object"*

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Tekst zdekodowany
0004E200	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0004E210	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0004E220	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0004E230	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0004E240	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0004E250	00	00	00	00	00	00	00	00	00	00	00	00	78	9C	ED	5D	............xśí]
0004E260	5B	73	DC	46	76	7E	4E	AA	F2	1F	10	E5	41	49	D5	88	[sÜFv~NŞň..ĺAIŐ.
0004E270	25	51	F2	AE	B4	2F	5B	94	6C	4B	8A	AF	65	6A	AD	72	%Qň®´/[”lKŠŻej.r
0004E280	DE	7A	80	9E	19	98	00	1A	8B	06	38	1A	A7	F2	DF	F3	Ţz€ž....‹.8.§ňßó
0004E290	7D	E7	9C	06	30	24	65	D9	A6	60	39	59	55	65	23	99	}çś.0$eŮ¦`9YUe#™
0004E2A0	9C	E9	CB	E9	73	F9	CE	55	77	7E	08	43	D6	77	65	9B	śéËésůÎUw~.CÖwe›
0004E2B0	85	4B	DF	65	FD	CE	67	E7	AD	AF	AA	EC	69	08	17	AB	…KßeýÎgç.ŻŞěi..«
0004E2C0	AC	18	62	9F	95	7D	16	36	9B	55	E6	9A	22	8B	7D	99	¬.bź•}.6›Ućš"‹}™
0004E2D0	5F	F0	27	65	93	1D	C2	D0	65	AD	CB	2F	4E	EE	FC	D3	_đ'e“.ÂĐe.Ë/NîüÓ
0004E2E0	BF	FC	F3	1D	AE	B4	29	F1	19	D7	64	BE	72	EB	D0	B9	żüó.®´)ń.×dIrëĐą

Note about multi-dimensional arrays.

In many programming languages you can use multi-dim arrays, but for computers those are unnatural. Thus compiler changes them to 1dim array on his own. It goes, for example, like this:

```
Tab[MaxX][MaxY][MaxZ]
```

translates into

```
Tab1[MaxX*MaxY*MaxZ]
```

You can access (x,y,z) coords like this

```
Tab1[x + y*MaxX + z*MaxX*MaxY]
```

Byte, Word, Dword?

Word - Natural unit of data used by a particular processor design. A word is a fixed-sized piece of data handled as a unit by the instruction set or the hardware of processor. In WoG project:

```
typedef unsigned char  Byte;
typedef unsigned short Word;
typedef unsigned long  Dword;
```

Struct.h

Most of editable (via erm) objects are modelled (declared) in structs.h. This includes heroes, adventure map objects, spells and much more. The important part to understand here is: you never create objects using those structs. You use them to create pointers, and then assign them address in game memory.

Quick note about pointers: it's variable that stores the memory address as its value. Brief example:

```
int a, *b;
b = &a;
*b = 5;
```

Here I created int-variable 'a' and pointer-to-int-variable 'b'. Using & operator I assigned 'b' pointer with **address** of 'a' var. Using * operator on pointer 'b' I assigned value '5' to place that 'b' points at – in this case, 'a' variable. So by operating on 'b' you can apply changes to 'a'. It's really simple, once you grasp it. If you want to learn more, or need a more in-depth explanation, check this.

Here is example of Campfire - adventure map object – model in structs.h.

```
// первые 4 байта для костра, "first 4 bytes for bonfire"
struct _CFire_{ // type 0x0C
    unsigned  ResType :  4;
    unsigned  ResVal  : 28;
};
```

Note the colon (:) in here. It allows you to use custom number of bits for your variable. In this case, `ResType` is 4bit unsigned integer variable, and `ResVal` is 28bit unsigned integer variable.

Let's discuss adventure map objects, for example: campfire. Coords are taken from receiver arguments (`FR(998)` f.e.) stored in `_ToDo_*`. Game loads data from memory via `GetMapItem(…)` into `_MapItem_*` struct. It is universal struct for adv map objects. Next, this pointer is assigned to `_CFire_*`. Because pointer is assigned directly to object data in game memory, all changes to this struct are reflected in-game without any synchronization.

```
int ERM_SetFire(char Cmd,int Num,_ToDo_*sp,Mes *Mp)
{
    STARTNA(__LINE__,&Mp->m.s[Mp->i])
    int v;
    Dword MixPos=GetDinMixPos(sp);
    _MapItem_ *mip=GetMapItem2(MixPos);
    _CFire_ *stp=(_CFire_ *)mip;
    if(mip->OType!=0xC){ MError("\"!!FR:\"-not a campfire."); RETURN(0) }
    switch(Cmd){
        case 'B': // B#1/#2 (4)(>5)
            CHECK_ParamsMin(2);
            v=stp->ResType; Apply(&v,4,Mp,0); stp->ResType=(Word)v;
            v=stp->ResVal;  Apply(&v,4,Mp,1); stp->ResVal=(Word)v;
            break;
        default: EWrongCommand(); RETURN(0)
    }
    RETURN(1)
}
```

Most important part for us is inside orange rectangle. You probably have questions here and don't worry, I will explain everything to ya. We cannot really analyse it line by line, cuz you cannot understand `GetDinMixPos(..)` before you know what `_MapItem_` is. Thus, let me guide you.

Let's start with the second line: acquiring `_MapItem_`. It's representation of every adventure map tile (square) consisting of terrain, road, object type and more. In comments you can see offset (in bytes) for every component. Take a brief look. Most important ones for us are: `OType/OSType` (object type/subtype) and `SetUp` stored in first 4 bytes.

```
struct _MapItem_{
    Dword   SetUp;      // +0,
    // ^^ most of object-specified data is stored here
    Byte    Land;       // +4
    Byte    LType;      // +5
    Byte    River;      // +6
    Byte    RType;      // +7
    Byte    Road;       // +8
    Byte    RdType;     // +9
    Word    _u1;        // +0A
//  Dword   Bits;
    Byte    Mirror;     // +0C тип отражения, "reflection type"

    Byte    Attrib;     // +0D ж или к клетки
    Word    _Bits;      // +0E
    Word    _u2;        // +10
    ODraw   *Draw;      // +12
    ODraw   *DrawEnd;   // +16
    ODraw   *DrawEnd2;  // +1A
    int     OType;      // +1E, type of object
    Word    OSType;     // +22, subtype of object
    Word    DrawNum;    // +24
};
```

It's mind-boggling for me that people recreated this struct just by looking into game memory. Anyway, this is actual representation of adventure map tile in memory. If you create pointer to this struct and assign correct part of memory – you can easily modify it at will, with human-readable interface. All structs in structs.h are such interfaces for memory, to make working with it easier – and safer.

To bring this back to Campfire – like I mentioned first 4 bytes, `SetUp`, are most important right now. This part of memory is object-dependent and every square has other data inside. Thus, `_CFire_` struct. Have you noticed it's 4 bytes in size? Most of structs for "Other Objects" in ERM are 4 bytes long and are used to work with `SetUp` component.

```
_CFire_ *stp=(_CFire_ *)mip;
```

This line does nothing, except assign memory stored in `_MapItem_*` to `_CFire_*` variable. This allows easy and safe access to `SetUp` component. It's first one in memory here, so no need for offset.

Now you should understand how you access parameters of object when you have pointer to map square representation in memory. But how you get address of square?

To access address of square, we need cords on map: X, Y, Z. Those are granted by ERM parser via arguments (_ToDo_ struct) to be extracted like this: `Dword MixPos=GetDinMixPos(sp);` They comes in compressed form, but don't worry about this. Just assume that `MixPos` are your coordinates.

```
Dword GetDinMixPos(_ToDo_ *sp)
{
    STARTNA(__LINE__, 0)
    Dword MixPos;
    int x,y,l,ind;
    switch(sp->ParSet){
        case 1:
            ind=GetVarVal(&sp->Par[0]);
            if((ind<1)||(ind>(VAR_COUNT_V-2))){ MError("Index of var for Dinamic position is out of range (1...9998)."); RETURN(0) }
            x=ERMVar2[ind-1]; y=ERMVar2[ind]; l=ERMVar2[ind+1];
            MixPos=PosMixed(x,y,l);
            break;
        case 3: // x/y/l
            x=GetVarVal(&sp->Par[0]);
            y=GetVarVal(&sp->Par[1]);
            l=GetVarVal(&sp->Par[2]);
            MixPos=PosMixed(x,y,l);
            break;
        default: // t/st
            EWrongParamsNum(); RETURN(0)
        }
    RETURN(MixPos)
}
```

Adventure map is 3 dimensional array, indexed X (horizontal), Y (vertical), Z (level), allocated on heap as contiguous block, thus in original source, it was probably made (somewhat) like this:
`_MapItem_*** GameMap = new _MapItem_[MaxX*MaxY*MaxZ]`
Accessing this array is similar to accessing array flattened by compiler, mentioned at the beginning: you need one index: `[x + y*MaxX+ z*MaxX*MaxY]` In H3 it's little easier cause MaxX=MaxY is the same.

We also need to find the beginning of adventure map array. I've no idea how, but they managed to find it. Look at code below. It decompress coordinates from `MixPos`

```
_MapItem_ *GetMapItem2(Dword MixPos)
{
    STARTNA(__LINE__, 0)
    int x,y,l;
    MixedPos(&x,&y,&l,MixPos);
    RETURN(GetMapItem(x,y,l))
}
```

then acquires `MapSize (MaxX/MaxY)` and pointer to (beginning of) adventure map array (*MIp0)

```
_MapItem_ *GetMapItem(int x,int y,int l) // back: Map2Coord
{
    STARTNA(__LINE__, 0)
    int MapSize;
    _MapItem_ *MIp0,*MIp;
    __asm{
        mov   ecx,BASE
        mov   ecx,[ecx]
        mov   eax,[ecx+0x1FC40]
        mov   MIp0,eax
        mov   eax,[ecx+0x1FC44]
        mov   MapSize,eax
    }
    if(x<0) x=0; if(x>=MapSize) x=MapSize-1;
    if(y<0) y=0; if(y>=MapSize) y=MapSize-1;
    if(l<0) l=0; if(l>1) l=1;
    MIp=&MIp0[x+(y+l*MapSize)*MapSize];
    RETURN(MIp)
}
```

Eventually, returns `&Mip0[x + y*MapSize + l*MapSize*MapSize]`

Summary

I've explained this topic from reversed perspective – started with last step and finished with first – because I considered it's easiest way to understand. Now let's sort out this knowledge.

```
Dword MixPos=GetDinMixPos(sp);
_MapItem_ *mip=GetMapItem2(MixPos);
_CFire_ *stp=(_CFire_ *)mip;
```

Tools made by WoG Team use `MixPos` instead of cords x/y/z, thus you need to extract them from ERM Parser using `GetDinMixPos(..)` function

Adventure map squares are stored in 1-dimensional array made of `_MapItem_` objects inside game memory. You can access every square by creating pointer and assigning it with address using `GetMapItem(..)` function.

Structs declared in structs.h are mostly interfaces. You can make pointers and assign in-game memory to them, for easier and safer editing.

First 4 bytes in `_MapItem_` are used to store object setup, individual for all types of object. Most of "other object" receivers edit that particular part of memory. You can assign address from `_MapItem_*` to `_CFire_*` for easier editing of `SetUp`.