Wog T1 – ERM Parser

by Jakub Grzana
Last update: 01.08.2020

About this document:

Here I will explain how you – as C/C++ programmer - might use ERM Parser. Thus I will describe every tool required to add new triggers and receivers, together with error management and support for various syntaxes. If you want to know how ERM Parser works internally – wrong address.

Macros, prerequisites and basics of WoG project:

Most of new source should be put in NewWog project, because it has enabled optimisation. Old project – t1b – has disabled optimisation for the sake of inline assembly that would be fucked up by compiler.

All header files (.h) should start with `#pragma once` for the sake of easy #including.

All functions must have declaration in header file, and definition in source (.cpp) file. With this, you can use function written in NewWog project (with optimisation) and call it inside t1b, from erm.cpp.

Every source (and only source) file must have defined `__FILENUM__` macro, like this.
`#define __FILENUM__ 8`
Every file must have UNIQUE number. Filenum is used with error display and logging. Additionally those features use `char* SourceFileList[]` from common.cpp. new files must be added here too. Neglecting this may lead to segfault. List of current filenums can be found inside mentioned char array, I will attach it at the end of document for your own convenience – note this one might be outdated.

Source for some macros. You are not expected to understand them yet (atleast I don't…)

```
// Slava recklessly modifies EBX in a lot of places, so LAStEBX should make things safer
#define START(x,y)    __asm{ pusha } int GErLeveL = GEr.Add(x,y); __asm{ popa }  int LAStEBX; __asm{ mov LAStEBX, ebx}
#define START(x)      __asm{ pusha } int GErLeveL = GEr.Add(x,0); __asm{ popa }  int LAStEBX; __asm{ mov LAStEBX, ebx}
#define STARTNA(x,y)  __asm{ pusha } int GErLeveL = GEr.AddN(__FILENUM__*1000000 + x,y); /*_CrtDumpMemoryLeaks();*/ __asm{ popa }  int
#define STOP          {__asm{ mov ebx, LAStEBX } __asm{ pusha }  PEr::Del(GErLeveL);  __asm{ popa }}
#define RETURN(x)     {__asm{ mov ebx, LAStEBX } __asm{ pusha }  PEr::Del(GErLeveL);  __asm{ popa } return(x);}
#define RETURNV       {__asm{ mov ebx, LAStEBX } __asm{ pusha }  PEr::Del(GErLeveL);  __asm{ popa } return;}
```

I've added new trigger and receiver: 'UU'. It's part of NewWog project, stored in NewWog folder. It's done purely for experiments, you can use it as well. In release version, this trigger and receiver will be disabled.

All information in this document are deduced from original source and (usually Russian and pretty enigmatic) comments. This is distributed in the hope that it will be useful, but without any warranty. Keep in mind I might have fucked up.

Triggers ERM

ERM Parser is well-written and does most of management job for us. Thus, adding trigger isn't very hard. All you have to do is add entry to global array ERM_Triggers

```
struct _ERM_Trigger_{ // triggers with 1 or no parameters
    Word Id; // name of trigger, for example 'BA'
    int Event; // the start of events range, UNIQUE identifier
    int paramMin; // minimum parameter value
    int paramMax; // maximum parameter value
    Byte post; // post trigger ?
} ERM_Triggers[]=
{
    {'IP',30330,0,3},          // IP#;
    {'BA',30300,0,1},          // BA#;
    {'BA',30350,50,54},        // BA#;
```

Id is name of trigger. 'Event' is start of unique identifiers range. ParamMin and ParamMax are min/max value of parameter, you can understand it easily by looking at !?BA. Parser will automatically detect your trigger in scripts, load it and store in savefile. Although this trigger will never be executed at this point - you still need caller.

Caller is function that calls (triggers) ERM trigger. Typically it is done using hook, intercepting call of internal function. For example, !?HD was created by hooking internal functions that display hint. While I won't discuss how to find internal function in this document, I will show how to add hook. Note: hooks will be further discussed in separate document.

```
__newCallers newCallers[]=
{
    {0x4F8171,long(newMainProc),H_ADRO|H_RADR},//Replace MainProc
    {0x601ACC,long(px_DirectDrawCreate),H_CALL,Q_Color32}, //DX Proxy Run
```

Just add entry to new_Callers global array, stored in global_hooks.h, following visible pattern; first internal function address, then function to be called. Using this global array, you can intercept function call at address 0x601ACC and call `px_DirectDrawCreate()`.

Inside this function you can initialise some variables - for example, v998/v999/v1000 in !?HD or !?OB - and finally you must call trigger. This is done by setting global variable `'pointer'` to trigger identifier, then calling function `ProcessERM();`

```
    pointer = 30372;
    ERM_GM_ai = -1;
    Map2Coord(HC_Map, &ERM_PosX, &ERM_PosY, &ERM_PosL);
    HC_MsgParams = MsgParams;
    HC_Map = 0;
    ProcessERM();
    HC_MsgParams = 0;
    RETURNV;
}
```

Note: ERM_PosX/Y/Z are automatically copied to v998/v999/v1000 inside `ProcessERM()` function. Additionally, you are required to start trigger function with `STARTNA(__LINE__,0)` macro, and end using `RETURNV` macro. Why? Not sure. It is used by several macros and error management.

```
void __fastcall HintTrigger(_HC_MsgParams_ *MsgParams)
{
    if (HC_Map == 0) return; // Uncharted Territory
    STARTNA(__LINE__, 0)
    ...
    RETURNV;
}
```

Receivers ERM

Adding new receivers is similar to adding new triggers: just add new entry to ERM_Addition global array in erm.cpp file.

```
struct _ERM_Addition_{
    Word   Id; // "name", like 'QW', 'UN'
    int   (*Fun)(char,int,_ToDo_ *,Mes *); // function to be called
    int    Type; // Type, represents arguments of receiver (NOT command!)
} ERM_Addition[]={
    {'CD',ERM_CasDem,0},
    {'MA',ERM_MonAtr,0},
    {'UN',ERM_Universal,0},
```

Id is name of receiver (`'OB'`), Fun is function to be called (`'ERM_SetObject'`), and Type is type of arguments for receiver (x/y/z for OB) List of all types will be attached at the end of document. All functions that are called by ERM Receivers should be named `'ERM_FunctionName'`, please stick to that rule.

Receiver function's header is standardised, as you can see in above struct.

```
int ERM_func(char Cmd,int Num,_ToDo_* sp,Mes *Mp)
// char Cmd - command, for example 'A', 'B', 'U' and so on. Use switch(Cmd) case 'A'...
// int Num - number of parameters passed
// _ToDo_* sp - struct containing various data, for example arguments of receiver: SY(v998,v999,v1000)
// Mes* Mp - struct containing parameters
{
    STARTNA(__LINE__, 0)
    switch(Cmd)
    {
        case 'A':
        {
            // here goes most of code
        } break;
    }
    RETURN(1)
}
```

Cmd is command to be executed, for example BA:A. Switch statement here is vastly superior to if-else for numerous reasons, including performance and readability, so please use switch. Additionally closing whole 'case' body inside bracket helps with managing local variables and code separation, so it's encouraged to use this style of coding.

All ERM receivers should manage wrong (unrecognised) commands. Easiest way to do so is to use switch statement with 'default'. Macro 'EWrongCommand()' will be discussed in Error Management.

```
default:
    { EWrongCommand(); RETURN(0); }
    break;
```

Remember to use `STARTNA(__LINE__,0)` and `RETURN(x)` similarly to triggers. `RETURN(0)` means error (parser will automatically create and show error message in this case) `RETURN(1)` means everything is fine.

You don't really need to know what exactly is inside Mes nor _ToDo_ structs, there are functions that manage them for you. Basically _ToDo_ stores all data regarding receiver arguments (OB10/12/1), and Mes stores parameters of command to be executed (:U?y1), including type of syntax (get/set). That being said, I will show to you code for those structs + google-translated comments.

```
struct Mes{
    // первым для быстрого доступа
    // "first for quick access"
    long  i;

    // Receiver might have 1 string in parameters
    _Mes_ m;

    // "up to 16 dependency flags | and &"
    VarNum Efl[2][16][2];      // до 16 флагов зависимости & и |

    // сами параметры
    // "parameters themselves"
    VarNum VarI[16];

    // вроде, используется только при парсинге
    //  "seems to be used only for parsing"
    char  c[16];

    // 1, если использован d синтаксис
    // "1 if d syntax is used"
    char  f[16];

    // строки ^^ обрабатываются не так как все. Они всегда стоят в конце и берутся по m.c[i] и далее
    // " ^^ lines are processed differently. They always stand at the end and are taken by m.c [i] and further"
    int   n[16];
};


struct _ToDo_{
    Word   Type;           // тип (id) ресивера, "type (id) of the receiver"
    Byte   Disabled;       // запрещен приемник, "prohibited receiver"
    Byte   DisabledPrev;   // пред состояние запрета, "before ban state"
    VarNum Efl[2][16][2];  // до 16 флагов зависимости & и | "up to 16 dependency flags & and |"
    Dword  Pointer;        // указатель на структуру, "pointer to structure"
    VarNum Par[16];        // указатель на структуру свойств этого объекта, "a pointer to the property structure of this object"
    int    ParSet;         // сколько Pointers установлено, "how many Pointers are installed"
    _Mes_  Self;           // НЕ#:
    _Mes_  Com;            // X#/#/#/#/#
};


struct VarNum {
      // номер флага, переменной  или число
      // "flag number, variable or number"
      int Num;

      // тип переменной
      // (в случае yv1 это - 6, т.е. y)
      // "variable type"
      // "(in case of yv1 this is 6, i.e. y)"
      // 0=число, 1=flag, 2=f...t, 3=v1...1000, 4=w1...100, 5=x1...100, 6=y1...100, 7=z-20...1000+,8=e1...e100
      Byte Type;

      // тип индиксирующей переменной
      // "type of the index variable"
      // indexed (в случае yv1 это - 3, т.е. v)
      // 0=нет, 1=флаг, 2=f...t, 3=v1...1000, 4=w1...100, 5=x1...100, 6=y1...100
      Byte IType;

      // тип проверки
      // "check type"
      Byte Check;  // 0=nothing, 1?, 2=, 3<>, 4>, 5<, 6>=, 7<=
};

// стандартный формат строки по указателю
// "standard string format by pointer"
struct _Mes_{
    char *s;
    long  l;
};
```

Gathering data for receiver: `Apply`, `StrMan::Apply`

While you can get parameter values and manage possible syntaxes on your own, it is easier – and much more safe – to use built-in tools. For numerical values it will be `Apply` function.

```
int Apply(void *dp,char size,Mes *mp,char ind)
// void* dp - address of variable to store data
// char size - size in bytes (fe. 4 means integer)
// Mes* Mp - pointer to Mes structure
// char ind - number of parameter (0..15)
```

`Apply` manage set, get and get-address (d?) syntax. Function returns 1 if value of *dp WASN'T changed (get syntax) and returns 0 if *dp WAS changed (set syntax). You can see now how to enforce get/set syntax for given parameter. Example:

```
// Call any trigger - note it is bad idea, cuz it doesn't make ANY other changes
case 'C':
{
    CHECK_ParamsNum(1);
    unsigned long ptr = 0;
    if(Apply(&ptr,sizeof(ptr),Mp,0)) { MError2("Cannot get parameter 1 - pointer"); RETURN(0); }
    pointer=ptr;
    ProcessERM();
} break;
```

Don't worry about CHECK_ParamsNum() nor MError2() macros, will be discussed later, in Error Management.

You can pass address of ANY variable to void* type - one of most cool things in C language in my opinion.

If you want to use get or **get-address (d?)** syntax, you should use `Apply` on global variables or directly on places in memory. Using local variables in this case might lead to segmentation fault. To use get-only syntax, backup data then use `Apply` – if returned 1, it's okay, otherwise restore value from backup and show error.

```
if(Apply(&output,4,Mp,3) == 0) { MError2("Cannot set parameter 4 - output value"); RETURN(0)}
```

For your convenience, examples of get-only and set-only syntaxes:
SET-SYNTAX ONLY: `if(Apply(&value,4,Mp,0)) { RETURN(0); }`
GET-SYNTAX ONLY: `if(Apply(&output,4,Mp,2) == 0) { RETURN(0) }`

`Apply` doesn't work with strings, fortunately we have `StrMan::Apply`.

```
int StrMan::Apply(char* destination, Mes* Mp, int index, int bufsize) // return 1 if z-var was changed
// destination - address of vessel for string
// mp - address of Mes struct, passed by ERM parser
// index - index in Mes (0..15, which parameter is string, f.e XX:B1/^Hello^ means ind=1)
// bufsize - Buffer size for string. Usually 255 is fine
```

Destination string must be initialised (memory must be allocated) and length must be equal/higher than bufsize. Works similarly to usual Apply. `StrMan::Apply` returns 0 if destination string was changed (set syntax) otherwise return 1 (get syntax)

Error management

There are two macros to show error message.

- `MError(char*)`
- `MError2(char*)`

After using either of those, you should use `RETURN`(0). Example:

```
if(Apply(&output,4,Mp,3) == 0) { MError2("Cannot set parameter 4 - output value"); RETURN(0)}
```

The difference between them is: `MError2()` automatically include receiver name and command, so there is no need to write it manually. You should use it ONLY inside ERM Receiver function! Everywhere else use `MError()`. Macros uses global variables that can have scrapped values outside ERM Receivers, which may lead to incorrect error messages or segfault.

To show values in ERM Error (it is highly advisable to show value of problematic parameter) use `sprintf_s` or `Format()` function. First one is part of Microsoft C++, second one is written in asm by WoG Team. Note it is probably safer to use `sprintf_s`:

```
int sprintf_s(
      char *buffer, // output string will be stored here
      size_t sizeOfBuffer, // length
      const char *format, // format string
      ... // variables
);
```

Also there are macros regarding common errors and number of parameters.

```
#define EWrongCommand() MError2("unknown command.")
#define EWrongSyntax()  MError2("wrong syntax.")
#define EWrongParam()   MError2("invalid parameter value.")
#define EWrongParamsNum() MError2("wrong number of parameters.")
#define CHECK_ParamsNum(n) if(Num!=n){EWrongParamsNum(); RETURN(0)}
#define CHECK_ParamsMax(n) if(Num>n) {EWrongParamsNum(); RETURN(0)}
#define CHECK_ParamsMin(n) if(Num<n) {EWrongParamsNum(); RETURN(0)}
```

Note MError2 - use it only within ERM receiver function for reasons mentioned before.

`char* Format()` function:

I have to admit, very cool function, written in assembler. Although it is advised to use `sprintf_s` instead, because it should be safer.

```
char* Format(const char*, ...)
```

Works like sprint, but returns char* instead of writing it into destination string.. You can use `Format("Hello from number %d", (int)i);` to insert value of 'i' into string. Don't know if you can use it with every dataformat. No memory leaks possible, `Format()` uses static char[30000] as vessel. Notes from comments:

// Format() must be called from main thread

// returned string can be used in subsequent call to Format

Attachment #1: Get/set syntax – old style, don't use it

Numerical variables: `Get/Set VarVal`

Another way of getting/setting value. You should use `Apply` and local variables to make setonly/getonly syntax, but I will still describe those shortly so you can understand original/old source.

```
int GetVarVal(VarNum *vnp)
int SetVarVal(VarNum *vnp,int Val) // 0 means error, 1 means ok
```

TODO don't understand how they works yet xD
Looks like it allows to get/set value. if it is numerical value, or get index of z-var. How does setting z-var work? Unknown yet

Strings: StrCopy

```
StrCopy( BFBackGrUDef, 255, ERM2String(&Mp->m.s[Mp->i],0,&vv));
int _fastcall StrCopy(char* destination, int length, char* source)
```

Simple function to copy char-array of "length" from "source" into "destination"

Though you need to 'cast' ERM parser parameters to ERM string. Thus, use ERM2String

```
char* ERM2String(char* message, int zstr, int* output_length)
```

As message, you must pass `&Mp->s[INSERT_INDEX]`
Not sure what zstr is, probably 1 if it's z-var, otherwise 0
output_length - function writes length of string INTO given variable (pass address!)

Attachment #2: ERM variables

```
int ERMVar[20]; - quick variables, f..t
char ERMFlags[1000]; - flags
int ERMVar2[VAR_COUNT_V]; - v-vars
int ERMVarX[16]; - x-vars
int *ERMVarY,*ERMVarYT; - normal/trigger y-vars
float *ERMVarF,*ERMVarFT; - normal/trigger e-vars
int ERMVarH[HERNUM][200]; - hero w-vars
char ERMString[1000][512]; - strings, z-vars
int PL_WoGOptions[2][PL_WONUM]; - WoG options. Dunno why it's 2-dim
array.
```

Every ERM variable has additional char-type variable 'USED', for example 'char ERMVar2Used[VAR_COUNT_V];' it probably stores 0/1 and is flag 'if_used', not sure for what purpose, maybe logging.

Remember that v998 is stored in ERMVar2[997]. ERM indexes from 1, and C indexes from 0.

int ERM_PosX, ERM_PosY, ERM_PosL; - x/y/l, you need to set them manually. v998/v999/v1000 are set to these values inside `ProcessERM()` function.
Map2Coord(_MapItem_*, int* x, int* y, int* l); can help you with that

Attachment #3: Triggers list

'Pointer' values for every trigger. Might be outdated. Always up-to-date version is in erm.cpp file, under ERM_Triggers[]

```
// 0-30000 functions
// 30000-30100 timers
// 30100-30300 heroes (!!! must change for new town)
// 30300 to battle
// 30301 from battle
// 30302 next battle turn
// 30303 monster's action
// 30304 monster's after action
// 30305 WM reach dest
// 30306 WM killed
// 30307 MR Magic Resistance in Battle (pre-)
// 30308 MR Magic Resistance in Battle (post-)
// 30309 MR Dwarf Magic Resistance
// 30310 mouse click на карте (правый) !?CM0
// 30311 mouse click в Городе !?CM1
// 30312 mouse click in hero screen !?CM2
// 30313 mouse click it two heroes screen !?CM3
// 30314 mouse click in battlefield !?CM4
// 30315 игрок снял артифакт
// 30316 игрок одел артифакт
// 30317 mouse move over battlefield
// 30318 mouse move over townscreen
// 30319 mouse click на карте (левый) !?CM5
// 30320 MP3 music
// 30321 WAV and M82 Sound 3.58
// 30322 Magic cast (adv. map) (pre-)
// 30323 Magic cast (adv. map) (post-)
// 30324 Go to Town Hall Screen 3.58 !?TH0
// 30325 Leave Town Hall Screen 3.58 !?TH1
// 30330 MP before sending data before battle
// 30331 MP after receiving data before battle
// 30332 MP before sending data after battle
// 30333 MP after receiving data after battle
// 30334 MP received data during a battle
// 30340 CO Перед открытием диалога Командира
// 30341 CO После закрытия диалога Командира
// 30342 CO После покупки Командира в городе
// 30343 CO После воскрешения Командира в городе
// 30350 to battle
// 30351 from battle
// 30352 universal to battle
// 30353 universal from battle
// 30354 on show battlefield (!?BA54)
// 30360 Load Game
// 30361 Save Game
// 30370 Post Instruction call (только при старте новой карты, но не загрузки)
// 30371 Dlg CallBack
```

```
// 30372 Hint text callback (!?HD)
// 30373 Castle Income (!?CI0)
// 30374 Castle Monsters Growth (!?CI1)
// 30375 Flag Color (!?FC)
// 30376 Dig Grail (!?DG)
// 30377 Get Map Position Importance for AI (!?AI)
// 30378 Testing trigger ~!!!
// 30400-30600 hero every movement (!!! must change for new town)
// 30600-30800 hero gain level (!!! must change for new town)
// 30800 setup battle field
// 30801 MFCall(0); !?MF0; - Defence coefficient
// 30802 MFCall(1); !?MF1; - Block ability
// 30803 MFCall(2); !?MF2; - Hate trigger - control hate and basic
damage done to a creature (Defence not included, any effects that
depend on chance not included)
// ...
// 30900-30904 ; TL - timer #-seconds to call
// 31000-31099 - 100 local functions
// 31100-31199 - autotimers (!?TM#1/#2/#3/#4)
// 31200-31400 - hero gain level post-trigger (!!! must change for
new town)
// ...
// 0x04000000 ... макс используемый тип - уровень
// 0x10000000+ OB position enter
// 0x20000000+ LE position
// 0x40000000+ OB type/subtype enter
// 0x08000000+ OB leave
//      +  0x10000000+ OB position leave
//      +  0x40000000+ OB type/subtype leavestruct _ERM_Trigger_{ //
triggers with 1 or no parameters

!?OBType/Subtype trigger pointer.
// !?OB - dec 1073741824, hex
// !?OB98; - dec 1074143232, hex 0x40062000,
 0x40062000 - 0x40000000 = 0x62000 = 401408
 401408/98 = 4096 = 0x1000
// !?OB98/6; - dec 1074143239, hex 0x40062007,
0x40062007 - 0x40062000 = 0x7 = 7
Thus i think it is calculated this way:
pointer = 0x40000000 + Type*0x1000 + 1 + Subtype
EDIT: Yup, proof: pointer = lpointer + (o_t<<12); (inside function
ERM2Object)
```

Attachment #4: Filenum list

Might be outdated. Always up-to-date might be found in common.cpp, near `char* SourceFileList[]`

```
erm.cpp: __FILENUM__ 1
herospec.cpp: __FILENUM__ 2
service.cpp: __FILENUM__ 3
common.cpp: __FILENUM__ 4
casdem.cpp: __FILENUM__ 5
monsters.cpp: __FILENUM__ 6
womo.cpp: __FILENUM__ 7
_B1.cpp: __FILENUM__ 8
anim.cpp: __FILENUM__ 9
sound.cpp: __FILENUM__ 10
ai.cpp: __FILENUM__ 11
npc.cpp: __FILENUM__ 12
-- 13 is free appearantly
string.cpp: __FILENUM__ 14
crexpo.cpp: __FILENUM__ 15
artifact.cpp: __FILENUM__ 16
txtfile.cpp: __FILENUM__ 17
timer.cpp: __FILENUM__ 18
lod.cpp: __FILENUM__ 19
dlg.cpp: __FILENUM__ 20
spell.cpp: __FILENUM__ 21
global.cpp: __FILENUM__ 22
-- 23 is free appearantly
luaW_dlg.h: __FILENUM__ 24 // Header file?
erm_lua.cpp: __FILENUM__ 25
dxproxy.cpp: __FILENUM__ 26 // this one is commented
-- Thus, i've added two more files with numbers
TestingERM.cpp: __FILENUM__ 30
RepeatableBattle.cpp: __FILENUM__ 31
```