M&B Module System Documentation

Table of Contents

PART 1

1.1 What is the Module System?

The Mount&Blade Module System is a set of python scripts that lets you create and/or modify content for Mount&Blade. This is actually the system we are using for working on the content of the official version. Using the module system, you can do things like adding new troop types, new characters, new quests, new dialogs, etc., or you can edit the existing content.

It is important to note that Mount&Blade does not use Python and does not read the Module System python scripts directly. Rather, the python scripts are executed to create the text files that Mount&Blade reads.

Mount&Blade actually reads its content from text files under the Mount&Blade/Modules folder. Thus, *in theory* you can do all the modifications you'd like to do by editing these text files. (Indeed, some modders had worked out how to use these files and were able to create amazing mods by themselves.) However the text files are not really human-readable and are very impractical to work with. There are currently two options for writing new modules. The first is the official module system described in this document. The other is Effidian's unofficial editor which is currently discontinued and unusable for the current version - v.1.011, but works for older versions of M&B, such as v.7.51 .

1.2 Requirements for using the Module System

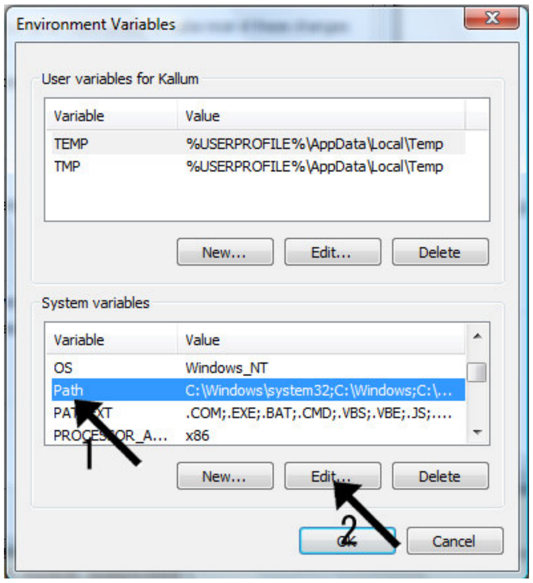
The module system consists of Python scripts, and as such, you need to have Python installed on your system to be able to work with them. You can download Python from Python.org's download page:
http://www.python.org/download/
There are more than a few downloads on that page. However you'll only need the **Windows version 2.6.1.**

NOTE: ANY OTHER VERSION OF PYTHON WILL NOT WORK. ONLY USE 2.6

After you download and install Python, you'll also need to add Python to your path enviroment variable. This is important, so try to be exact when you make these changes.

For Windows 9x systems, you can edit autoexec.bat file and add your python folder to the Path. For example if Python is installed under C:\Python24, add the following line:
set PATH=C:\Python24;%PATH%

If you have a Windows XP or a Windows Vista system, this operation is slightly different: Right-click on My Computer (Computer in the start menu for Vista users) select 'Properties', click on the 'Advanced' tab and then click on 'Enviroment Variables':



User variables for Kallum

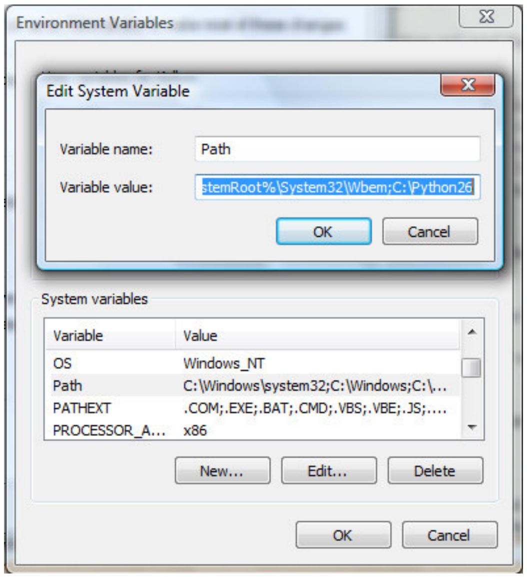
Variable	Value
TEMP	%USERPROFILE%\AppData\Local\Temp
TMP	%USERPROFILE%\AppData\Local\Temp

System variables

Variable	Value
OS	Windows_NT
Path	C:\Windows\system32;C:\Windows;C:\...
PATHEXT	.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;....
PROCESSOR_A...	x86

1. Scroll down on the 'System variables' until you find the 'Path' variable.

2. Click on the 'Edit...' button, a new window should pop up:



Variable name: Path

Variable value: stemRoot%\System32\Wbem;C:\Python26

System variables

Variable	Value
OS	Windows_NT
Path	C:\Windows\system32;C:\Windows;C:\...
PATHEXT	.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;....
PROCESSOR_A...	x86

Scroll to the end of the 'Variable value' and add ";C:\Python26", where “C:\Python26” is the name of the folder python is installed to.

Click ok on this box, then ok again on the next box.

1.3 Obtaining the Module System
The latest version of the Module System can be downloaded from Module System's official webpage at:
http://www.taleworlds.com/download/mb_module_system_1010_0.zip

You'll need to download the zip file for the module system and extract it (requring a program such as WinRAR, or 7-zip). Extract the Module System wherever it is easy to locate, e.g, the Desktop or My Documents.

1.4 Module Sytem Files
Now, let's have a look at the files in the module system. When we look at the actual Python files (files that end with .py) we see that there are actually four kinds of files:

* files that start with header_
* files that start with process_
* files that start with ID_
* files that start with module_

The first two kinds of files are necessary for running the module system. You shouldn't modify these at all. The third kind of files (ID_) are temporary files created while building the module. You can delete them if you like and the module system will generate them again. The final kind of files (module_) are actually the files that contain the content data. These are the files you will be modifying.

1.5 Creating a new module

Before going any further, let's first create a folder for your new module. For this, we need to go to the Mount&Blade/Modules folder (by default this is at "c:/Program Files/Mount&Blade/Modules") Now, under the Modules folder there should be a folder named Native. This is, so to speak, the official module. For your own module, you must create a new folder here, and copy the files from Native to the new folder. This new folder will be your own module folder so name it as you like. For the sake of simplicity, I am assuming it is named MyNewModule

You can test if you have done this right by launching Mount&Blade. Now, Mount&Blade's launch window should show a combo box, which lets you select the module you'd like to play. Now, try selecting your new module and starting a new game. Since we copied the contents of the native folder for our new module, the game we play now will be identical to the native game.

Next, we must make the Module System use the new folder as its target. To do that, open the file module_info.py for editing (Right click on the file and select "Edit with IDLE"; or open the file with Notepad or your preferred text editor) and change export_dir to point to your new folder. For example, if the folder for your module is: c:\Program Files\Mount&Blade\Modules\MyNewModule You should change this line as follows:

export_dir = "C:**/**Program Files**/**Mount&Blade**/**Modules**/**MyNewModule**/**"

NOTE: Within the Module System, directories are separated by **forward slashes(/)** not backslashes(\) that windows uses. Also note that you need the forward slash at the end of the declaration. Now our module system setup should be ready. To try it out, remove the file conversation.txt inside our new module folder, and then double click on **build_module.bat**. You should see a command prompt with some output like this:

Code: [Select]
```
Initializing...
Compiling all global variables...
Exporting strings...
Exporting skills...
Exporting tracks...
Exporting animations...
Exporting meshes...
Exporting sounds...
Exporting skins...
Exporting map icons...
Creating new tag_uses.txt file...
Creating new quick_strings.txt file...
Exporting faction data...
Exporting item data...
Exporting scene data...
Exporting troops data
Exporting particle data...
Exporting scene props...
Exporting tableau materials data...
Exporting presentations...
Exporting party_template data...
Exporting parties
Exporting quest data...
Exporting scripts...
Exporting mission_template data...
Exporting game menus data...
exporting simple triggers...
exporting triggers...
exporting dialogs...
Checking global variable usages...

______________________________

Script processing has ended.
Press any key to exit. . .
```

If you ran into an error, make sure you've followed all the steps of this tutorial exactly, and if you think you have, please use the Search function on the forums; chances are someone has already run into the same problem and an easy solution has already been posted.

If not, you should see the conversation.txt file re-created. Congratulations! You're all set up to create your own mod with the M&B module system.

PART 2

As mentioned in the previous chapter, you work with the Module system as follows:
1) Edit one or more of the module files (those starting with module_ and ending with .py) and make any changes you like. (Usually you need to right click and select 'Edit with Idle' to do that using the phython editor)

2) After that, double click on the file build_module.bat . This action will attempt to build your module (and report errors if there are any)
3) If there are no errors, you may launch Mount&Blade and test the changes you have made. Sometimes you may need to start a new game for the changes to take effect.

NOTE: Though PYTHON has its own editor, I have found that using NOTEPAD++ is of great help. You can have multiple files open (in tabs), as well as easily customize the look a feel of the interface. I have found this very helpful since I am used to working with other code editors (such as BYOND) with a specific layout.

2.1 – Personalizing Your Mount&Blade Mod

Before we begin, we will take the first step in personalizing your mod. Let's change the mod selection picture. This can be done with MS Paint, or anything that will edit BMP files. The file (main.bmp) is located in your Mount&Blade modules folder (ex. C:\Program Files\Mount&Blade\Modules\MyMod). Editing this will change the picture shown when you select your mod. This is often a good idea if you jump between working on a mod and playing others.

If you have the ability to edit DDS image files, you can make copies of the various DDS background pictures to your mod's texture folder (ex. from C:\Program Files\Mount&Blade\Textures to C:\Program Files\Mount&Blade\Modules\MyMod\Textures). If you change them in your mod's texture folder, they will be changed in the game when you play your mod. The file bg2.dds is the standard background for most menu pictures, and pic_mercenary.dds is the picture for the main menu. Don't spend too much time with this now, but know that you can make your mod look a lot slicker when it's done.

2.2 -- Editing the Module Files

The module system uses Python lists to represent collections of game objects. (A python lists starts with a square bracket '[', includes a list of objects seperated by commas, and ends with a closing square bracket ']') If you open and view any of the module files you'll see that it contains such a list. If you open up **module_map_icons.py**, for example, it contains:

```
map_icons = [
  ("player",0,"player", avatar_scale, snd_footstep_grass, 0.15, 0.173, 0),
  ("player_horseman",0,"player_horseman", avatar_scale, snd_gallop, 0.15, 0.173, 0),
  ("gray_knight",0,"knight_a", avatar_scale, snd_gallop, 0.15, 0.173, 0),
  ("vaegir_knight",0,"knight_b", avatar_scale, snd_gallop, 0.15, 0.173, 0),
  ("flagbearer_a",0,"flagbearer_a", avatar_scale, snd_gallop, 0.15, 0.173, 0),
  ("flagbearer_b",0,"flagbearer_b", avatar_scale, snd_gallop, 0.15, 0.173, 0),
  ("peasant",0,"peasant_a", avatar_scale,snd_footstep_grass, 0.15, 0.173, 0),
  ("khergit",0,"khergit_horseman", avatar_scale,snd_gallop, 0.15, 0.173, 0),
  ("khergit_horseman_b",0,"khergit_horseman_b", avatar_scale,snd_gallop, 0.15, 0.173, 0),
  ("axeman",0,"bandit_a", avatar_scale,snd_footstep_grass, 0.15, 0.173, 0),
  ("woman",0,"woman_a", avatar_scale,snd_footstep_grass, 0.15, 0.173, 0),
  ("woman_b",0,"woman_b", avatar_scale,snd_footstep_grass, 0.15, 0.173, 0),
  ("town",mcn_no_shadow,"map_town_a", 0.35,0),
  ("town_steppe",mcn_no_shadow,"map_town_steppe_a", 0.35,0),
  ("village_a",mcn_no_shadow,"map_village_a", 0.45, 0),
  ("village_burnt_a",mcn_no_shadow,"map_village_burnt_a", 0.45, 0),
  ("village_deserted_a",mcn_no_shadow,"map_village_deserted_a", 0.45, 0),

###more in the actual code, but we'll stop here###
```

Here map_icons is declared as a Python list and every element in the list is the declaration for a specific map icon object. In this example, ("player",0,"player", avatar_scale, snd_footstep_grass, 0.15, 0.173, 0), is such an object. We call such objects tuples. Tuples, like lists, contain elements seperated by commas (but they start and end with parentheses). The structure of each tuple object is documented at the beginning of the module file. For map icons, each tuple object contains:

1) name of the icon, The prefix icon_ is automatically added before each map icon id.
2) icon flags, See **header_map icons.py** for a list of available flags
3) Mesh name, can be found in the BRF files map_icon_meshes, map_icons_b, and map_icons_c
4) Mesh scale,
5) sound id.
6) Offset x position for the flag icon.
7) Offset y position for the flag icon.
8) Offset z position for the flag icon.

So, for the first tuple:
("player",0,"player", avatar_scale, snd_footstep_grass, 0.15, 0.173, 0),
1) name of the icon = "player"
2) icon flags = 0
3) Mesh name = "player"
4) Mesh scale = 0.2
5) sound id = snd_footstep_grass
6) Flag offset x = 0.15
7) Flag offset y = 0.173
6) Flag offset z = 0

You can work out the structure of game objects for each module system file in this way, by reading the documentation at the beginning and matching that with the contents of the list.

2.3 -- Adding New Game Objects

Knowing the structure of the map icon tuples, we can now begin to add our own map icons. Let us take another look at the list.

```
map_icons = [
  ("player",0,"player", avatar_scale, snd_footstep_grass, 0.15, 0.173, 0),
  ("player_horseman",0,"player_horseman", avatar_scale, snd_gallop, 0.15, 0.173, 0),
  ("gray_knight",0,"knight_a", avatar_scale, snd_gallop, 0.15, 0.173, 0),
  ("vaegir_knight",0,"knight_b", avatar_scale, snd_gallop, 0.15, 0.173, 0),
  ("flagbearer_a",0,"flagbearer_a", avatar_scale, snd_gallop, 0.15, 0.173, 0),
.
.
.
  ("banner_125",0,"map_flag_f20", banner_scale,0),
  ("banner_126",0,"map_flag_15", banner_scale,0),
]
###more in the actual code, but we'll stop here###
```

New game objects in any module file must be added *inside* the list. You can see, the list for module_map_icons ends just below ("banner_126",0,"map_flag_15", banner_scale,0)). In order to make room for our new game object, we have to move the bracket down by one line.

Having done that, we can now begin to add a new object. The easiest way to do this is to copy and paste a pre-existing object and then editing its contents. For example, copy the "town" tuple just below "banner_126":

```
.
.
.
  ("banner_126",0,"map_flag_15", banner_scale,0),
   ("town",mcn_no_shadow,"map_town_a", 0.35,0),
]
```

In this example, we have copied ("town",mcn_no_shadow,"map_town_a", 0.35,0). Now give it a new icon name; "new_icon". This new icon has a *flag* on it. Flags are settings that can be turned on and off by including or removing them in the appropriate field. For example, the flag mcn_no_shadow on our new icon will set this icon to cast no shadow on the ground.

We will now remove mcn_no_shadow from our new icon. To do this, we replace mcn_no_shadow with 0, telling the module system there are no flags for this icon.

```
.
.
.
  ("banner_126",0,"map_flag_15", banner_scale,0),
  ("new_icon",0,"map_town_a", 0.35, 0),
]
```

Both "town" and "new_icon" use the Mesh "map_town_a", which means they will both use the 3d model named map_town_a in the game's brf resource files. By changing this field, the icon can be set to use any 3d model from the resource files. Because both icons use the same Mesh, if we were to put "new_icon" into the game at this point, it would look exactly the same as "town".

Now let us give "new_icon" a bit of a different look. Let's change it to "**city**". This mesh is currently not used and it will stand out better in our mod.

```
.
.
.
  ("banner_126",0,"map_flag_15", banner_scale,0),
  ("new_icon",0,"city", 2,0),
]
```

In this example, we have also changed the icon's *scale* from 0.35 to **2**. This means the icon will be displayed two times as large as normal. That should help us tell it apart from "town" when we put it into the game.

Next we will create a new party in module_parties.py that uses our new icon. To do this, we will need to *reference* the icon from module_parties.py. Before continuing, we will run build_module.bat. This will let us know that we got the syntax right. It is a good idea to run the build to ensure that you aren't making errors that may be hard to track down later.

2.4 -- Referencing Game Objects

Open **module_parties.py** in your module system folder. You will see another Python list, parties = [, just below some constants declarations (pf_town = pf_is_static|pf_always_visible|pf_show_faction|pf_label_large). They can make repeated flags a lot easier to enter. More on constants later.

As you can see, the structure of tuples in module_parties.py is slightly different from module_icons. This holds true for many -- if not all -- of the module files. We'll take this opportunity to closely examine the parties structure. First, Understand that parties can be anything that you come in contact with on the map. These can be traveling groups or stationary villiages, towns or other locations of your own design. One key to parties is that an encounter is triggered when 2 or more parties meet.

Let's look at an example of a party. If you scroll down a bit you will come to:

```
  ("town_1","Sargoth",  icon_town|pf_town, no_menu, pt_none, fac_neutral,0,ai_bhvr_hold,0,(-1.55, 66.45),[], 170),
```

This tuple places the town of Sargoth on the map. Sargoth's various qualities are set in the appropriate fields -- quite similar to the fields we've seen in **module_map_icons.py**.

Breakdown of the tuple fields:

1) Party-id. This is used for referencing the party in other files.
2) Party name. This is the party's name as it will appear in-game. Can be as different from the party-id as you like.
3) Party flags. The first flag of every party object must be the icon that this party will use.
4) Menu. This field is *deprecated*, which means that it's outdated and no longer used. As of M&B version 0.730, this field has no effect whatsoever in the game.
5) Party-template. ID of the party template this party belongs to. Use pt_none as the default value.
6) Party faction. This can be any entry from module_factions.py.
7) Party personality. See header_parties.py for an explanation of personality flags.
8) AI-behaviour. How the AI party will act on the overland map.
9) AI-target party. The AI-behaviour's target.
10) Initial coordinates. The party's starting coordinates on the overland map; X, Y.
11) List of troop stacks. Each stack record is a triple that contains the following fields:
 11.1) Troop-id. This can be any regular or hero troop from module_troops.py.
 11.2) Number of troops in this stack; does not vary. The number you input here is the number of troops the town will have.
 11.3) Member flags. Optional. Use pmf_is_prisoner to note that a party member is a prisoner.
12) Party direction in degrees [optional]

Sargoth's tuple examination:

```
  ("town_1","Sargoth",  icon_town|pf_town, no_menu, pt_none, fac_neutral,0,ai_bhvr_hold,0,(-1.55, 66.45),[], 170),
```

1) Party-id = "town_1"
2) Party name = "Sargoth"
3) Party flags = icon_town|pf_town
4) Menu = no_menu
5) Party-template = pt_none
6) Party faction = fac_neutral
7) Party personality = 0
8) AI-behaviour = ai_bhvr_hold
9) AI-target party = 0
10) Initial coordinates = (-1.55, 66.45)
11) List of troop stacks = [] (None)
12) Party direction = 170

By looking at field 3, we can see that Sargoth references the icon "town" from module_icons.py, by adding the prefix *icon_* to it. This prefix is what points the system to the right module file. In order to reference module_icons, we use *icon_*; in order to reference module_factions, we use *fac_*; in order to reference module_parties, we use *p_*; and so on. There is an appropriate prefix for every module file -- you will find them all listed at the bottom of this segment.

Now that we know how parties are structured, we can begin adding our own. But before you do so, take note: In the case of module_parties.py and certain other module files, you should *not* add your new towns at the bottom of the list. There will be comments in these files warning you about doing this, as it can break operations in the native code. In module_parties.py, it is recommended that you add any new towns between "town_1" and "castle_1". This is defined in the module_constants.py. More on that later.

Now, copy the entry "town_1" and paste it in just after "town_18" but before "castle_1"..

```
.
.
.
  ("town_18","Narra",  icon_town_steppe|pf_town, no_menu, pt_none, fac_neutral,0,ai_bhvr_hold,0,(-22.6, -82),[], 135),

##JIK's Test Area
  ("town_1","Sargoth",  icon_town|pf_town, no_menu, pt_none, fac_neutral,0,ai_bhvr_hold,0,(-1.55, 66.45),[], 170),
##End of JIK's Test Area
```

```
#   Aztaq_Castle
#  Malabadi_Castle
  ("castle_1","Culmarr_Castle",icon_castle_a|pf_castle, no_menu, pt_none, fac_neutral,0,ai_bhvr_hold,0,(-69.2, 31.3),[],50),.
.
.
.
```

I will take this time to quickly talk about documenting and comments. As you can see I have commented out the section that I will be using and marked it with my name using a # at the front. This will make it easy to find the areas you are working in (using the FIND search in most editors). It is also a good idea to use comments to say what you are doing, or what it is in relation to if it's not apparent what you are doing. Anything after the # is not read by the compiler, so you can comment out whole lines, or add comments to end of lines. This makes is easier for you to edit to your mods, as well as to aid others whom may wish to learn from what you did. In this tutorial, I will start to include some basic commenting. Start making it a habit to do the same.

In this example, we will change the new party's identifier from "town_1" to "mod_town", and the party name from "Sargoth" to "Mod_Town".

We can now establish several things from looking at the tuple.

1) To reference this party from another file, we must use the identifier "mod_town" with the prefix **"p_"**, resulting in "p_mod_town".
2) In the game, we will only see the name "Mod_Town" to describe this party, never the identifier.
3) This party uses icon_town and the flag pf_town -- a flag that assigns common town settings. The flags field will be where our next few changes take place.
4) Mod Town is currently of the neutral faction.
5) If we were to put our new town into the game at this point, it would appear at exactly the same map coordinates as Sargoth. This, too, we will change next.

```
##JIK's Test Area
  ("mod_town","Mod_Town",  icon_new_icon|pf_town, no_menu, pt_none, fac_neutral,0,ai_bhvr_hold,0,(-1, -1),[], 45),
##End of JIK's Test Area
```

Here we have changed its map coordinates to (-1,-1), and changed it's rotation to 45, to make it look a bit different. The town is now set up to use our new icon, "new_icon", and has its own unique map coordinates, allowing it to show up without problems.

Save your progress, then click on build_module.bat. If everything went well, you should now be able to start up your mod and see the new town and icon near the centre of the map. Try it.

If everything did *not* go well, check carefully for spelling and syntax. Make very sure that all commas and brackets are in the right place. Bad syntax is the most common source of compiler errors in the official module system. After compiling, errors are usually pointed out at specific lines. Using a code editor like NOTEPAD++ makes it easier as lines are numbered, and it highlights (when you mouse over one) both the beginning and ending brackets.

In-game, travelling to the town now will trigger the town menu. You can try visiting the tavern, or walk around the town, but you may find that the scenes are wrong. We may have set up the party on the map, but we have yet to finish the guts. We will touch on this later, when we build up a tavern for a little quest. Let's not get a head of ourselves, one step at a time…

As you can see, the *interrelation* of the various module files can be extensive. Every part must be covered for your module to work properly. Fortunately, most changes only require the editing of one or two files at most.

Now that you have a thorough grasp of the modding basics, we can take an in-depth look into the various module files. It is important to know that to access objects from other module files, you will need to use the object prefixes. Some of them are listed here. Each object type's prefix can be found in the information section describing the tuples at the start of each **module_*<type>*.py** file.
List of module file prefixes:

fac_ -- module_factions.py
icon_ -- module_map_icons.py
itm_ -- module_items.py
mnu_ -- module_game_menus.py
~~mno_ -- module_game_menus.py -- References an individual menu option in module_game_menus.~~ **NOT SURE IF THIS IS STILL REAL**
mt_ -- module_mission_templates.py
psys_ -- module_particle_systems.py
p_ -- module_parties.py
pt_ -- module_party_templates.py
qst_ -- module_quests.py
script_ -- module_scripts.py
scn_ -- module_scenes.py
spr_ -- module_scene_props.py
str_ -- module_strings.py
trp_ -- module_troops.py
skl_ -- module_skills.py

module_dialogs.py is never directly referenced, so it has no prefix.

PART 3

In this chapter of the documentation, we cover **module_troops.py** and its functions. Module_troops is where all regular troops, Heroes, chests and town NPCs are defined, complete with faces, ability scores and inventory. Whenever you wish to make a new character or troop type, this is the file you'll be modding.

3.1 -- Breakdown of Module_Troops

The file begins with a small block of code that calculates weapon proficiencies and some other non-moddable code. Since this whole block falls outside the Python list and we will not be editing it, it needn't concern us yet. Skip ahead to the list troops = [.

Here we find tuples for the player and several other troops important to the game. Just below that are the various fighters we encounter in the arena fights. We'll study a few of these, as they are excellent examples of regular troops' level progression.

Observe:

```
  ["novice_fighter","Novice Fighter","Novice Fighters",tf_guarantee_boots|tf_guarantee_armor,no_scene,reserved,fac_commoners,
   [itm_hide_boots],
   str_6|agi_6|level(5),wp(60),knows_common,mercenary_face_1, mercenary_face_2],
```

This is a bog-standard troop called "novice fighter". "novice fighter" is low-level, not very good at fighting, has low ability scores, and is otherwise unremarkable.

Breakdown of the tuple fields:

1) Troop id. Used for referencing troops in other files.
2) Toop name.
3) Plural troop name.
4) Troop flags. *tf_guarantee_** flags must be set if you want to make sure a troop always gets equipped with a certain category of inventory. If you do not, the troop may appear without armour of that category. Only melee weapons are guaranteed to be equipped, if there are any in the troop's inventory.
5) Scene. This is only applicable to Heroes; it governs at which scene and entry point the Hero will appear. For example, scn_reyvadin_castle|entry(1) puts the troop at entry point 1 in Reyvadin Castle.
6) Reserved. Not currently used; must be either reserved or 0.
7) Faction. The troop's faction, used with the *fac_* prefix.
8) Inventory. A list of items in the troop's inventory. Regular troops will choose equipment from this list at random.
9) Attributes. The troop's attribute scores and character level. These work exactly as they do for the player.
10) Weapon proficiencies. The weapon proficiency scores of this troop. The function wp(x) will create random weapon proficiencies close to value x, but you can also add extra definitions to specifically designate certain proficiency scores. For example, to make an expert archer with other weapon proficiencies close to 60 you could use:

```
        wp_archery(160) | wp(60)
```

11) Skills. These are the same as the player's skills. Note that, in addition to the attributes and skills you've defined, the troop also gets 1 random attribute point and 1 random skill point per character level.
12) Face code. The game will generate a face according to this code. **You can export new face codes from the game by pressing CTRL+E in the face editor screen while Edit Mode is active.**
13) Face code 2. Only applicable to regular troops, can be omitted for Heroes. The game will create random faces between face code 1 and face code 2 for each individual of this troop type.

Novice_fighter tuple examination:

1) Troop id = "novice_fighter"
2) Toop name = "novice_fighter"
3) Plural troop name = "novice_fighters"
4) Troop flags = tf_guarantee_boots|tf_guarantee_armor
5) Scene = no_scene
6) Reserved = reserved
7) Faction = fac_commoners
8) Inventory = [itm_sword,itm_hide_boots]
9) Attributes = str_6|agi_6|level(5)
10) Weapon proficiencies = wp(60)
11) Skills = knows_common
12) Face code = swadian_face1
13) Face code 2 = swadian_face2

There are three things worth noting about this tuple.

Our "novice fighter" has tf_guarantee_armor, but no armour of his own. However, this does not make tf_guarantee_armor redundant; the troop will put on any armour he receives during the game.

To begin with (ie, at Level 1), "novice_fighter" has a STR of 6 and an AGI of 6. Upon start of the game, he is bumped up to Level 5, with all the usual stat gains that implies.

He has the skill knows_common. knows_common is a collection of skills that was defined at the beginning of module_troops; scroll up and observe this collection now.

```
knows_common = knows_riding_1|knows_trade_2|knows_inventory_management_2|knows_prisoner_management_1|knows_leadership_1
```

A troop that has knows_common will have every skill listed here; a Riding skill of 1, a Trade skill of 2, an Inventory Management skill of 2, a Prisoner Management skill of 1, and a Leadership skill of 1. knows_common is what is known as a *constant*; a phrase that represents something else, be it a number, an identifier, another constant, or any other valid object. A constant can represent any number of objects, as long as those objects are in the right order for the place where you intend to use this constant.

In this case, knows_common is defined as having everything on the right side of the = sign. So in effect, by putting knows_common in the Skills field, the module system will function just as if you'd typed out all of that in the Skills field.

Now let us look at the next entry in the list.

```
  ["regular_fighter","Regular Fighter","Regular Fighters",tf_guarantee_boots|tf_guarantee_armor,no_scene,reserved,fac_commoners,
   [itm_hide_boots],
str_8|agi_8|level(11),wp(90),knows_common|knows_ironflesh_1|knows_power_strike_1|knows_athletics_1|knows_riding_1|knows_shield_2,mercenary_face_1, mercenary_face_2],
```

In this example, you can see the slightly stronger "regular fighter"; this one has higher ability scores, is level 11, and knows some skills beyond knows_common. In-game, if some "novice fighters" in our party had reached sufficient experience to reach level 11, we might want to allow them to upgrade into "regular fighters". Next, we will see how this is done.

3.2 -- Upgrading Troops

The list of which troops can be upgraded into what is contained at the very bottom of module_troops. This defines which troops can be upgraded, including what they can upgrade into, when the experience conditions are met.

As you can see, each troop's upgrade choices must be defined here through the operation upgrade(troops). The first string is the ID of the troop to be upgraded, the second string is the ID of the resulting troop. For example, upgrade(troops,"farmer", "watchman") will allow a "farmer" to upgrade into a "watchman", when the "farmer" has accrued enough experience.

There are two types of upgrade operations.

upgrade(troops,"source_troop", "target_troop_1") offers only one upgrade choice; "source_troop" to "target_troop_1".

upgrade2(troops,"source_troop", "target_troop_1", "target_troop_2"), however, offers the player a choice to upgrade "source_troop" into either "target_troop_1" or "target_troop_2". Two is currently the maximum number of possible upgrade choices.

There is currently no entry for "novice_fighter" in this block, so let's make one. Copy upgrade(troops,"farmer", "watchman") and paste it to the bottom of the block. Notice that the upgrade section is outside of the main code block for troops (after the last '**]**'). It is important to watch for where aspects of the code are located, as it is important for the compiler to see things in a specific order. Also note that here they do not have commas (**,**) after each line. Always keep in mind where you are in or outside the code, and the conventions being used..

Then change "farmer" to "novice_fighter" and "watchman" to "regular_fighter". Any "novice fighters" in a party will now be able to upgrade to "regular fighters" as described in the last segment.

Next, we'll take it a bit further. Make another upgrade entry at the bottom of the list, with the source troop "new_troop" and the target troop "regular_fighter". "new_troop" doesn't exist just yet, but it will soon! Scroll up to the phrase: **# Add Extra Quest NPCs below this point**. New troops should be added before the "local_merchant" entry which is what we're going to do now.

3.3 -- Adding New Troops

Make a bit of space just below **# Add Extra Quest NPCs below this point**, then copy/paste the "novice_fighter" code into the empty space, and make the following changes:

```
##JIK - new troop entry
  ["new_troop","new_troop","new_troops",tf_guarantee_boots|tf_guarantee_armor,no_scene,reserved,fac_commoners,
   [itm_sword_medieval_a,itm_fighting_axe,itm_leather_jerkin,itm_skullcap,itm_hide_boots],
   str_6|agi_6|level(5),wp(60),knows_common,mercenary_face_1, mercenary_face_2],
```

From now on, every troop of the type "new_troop" will be wearing itm_leather_jerkin. They will also randomly wield either itm_sword_medieval_a or itm_fighting_axe. However, only *some* of them will have itm_skullcap, because of the entries in the Flags field; this troop only has guaranteed **armour** and **boots**. In order to make sure our new troops will all have helmets, we would have to add tf_guarantee_helmet to the Flags field. Let's do so.

```
##JIK - new troop entry
  ["new_troop","new_troop","new_troops",tf_guarantee_boots|tf_guarantee_armor|tf_guarantee_helmet,no_scene,reserved,fac_commoners,
   [itm_sword_medieval_a,itm_fighting_axe,itm_leather_jerkin,itm_skullcap,itm_hide_boots],
   str_6|agi_6|level(5),wp(60),knows_common,mercenary_face_1, mercenary_face_2],
```

For our next edit, we will make some changes the troop's stats. Put its STR to 9, and AGI to 9. Once that's done, change Level to 4 and weapon proficiency to 80.

Our "new troop" should now look like this:

```
##JIK - new troop entry
  ["new_troop","new_troop","new_troops",tf_guarantee_boots|tf_guarantee_armor|tf_guarantee_helmet,no_scene,reserved,fac_commoners,
   [itm_sword_medieval_a,itm_fighting_axe,itm_leather_jerkin,itm_skullcap,itm_hide_boots],
   str_9|agi_9|level(4),wp(80),knows_common,mercenary_face_1, mercenary_face_2],
```

It's now ready to be placed into the game, as an experiment. First let's compile our work to make sure there are no errors. Run **build_module.bat**.

3.4 – Mercenaries This section does not exist anymore. Will test adding new_troop to the player's stack. Maybe later I'll add hiring them from the Tavern scene.

Save your progress, then open **module_parties.py**. Let's give you a few friends to start out with you on your adventure. We'll give you 5 new_troop soldiers to your party stack. Edit "main_party" to look like this:

```
  ("main_party","Main Party",icon_player|pf_limit_members, no_menu, pt_none,fac_player_faction,0,ai_bhvr_hold,0,(17,
52.5),[(trp_player,1,0),(trp_new_troop,5,0)]),
```

We have added ,(trp_new_troop,5,0) to the party stack (the ',' is important!). Save your progress, close module_parties, and double-click on **build_module.bat**. If the build finishes without problems, you will now find your new troops as part of your party. *(You'll need to start a new game for the new troops to show up. You need to start a new game for changes to parties and most other objects to take effect.)*

Go out, fight some battles, and notice how you are able to upgrade the new troops to "regular fighters" when they accumulate enough XP.

Congratulations! You now know how to make and manipulate regular troops. We will cover Heroes, Merchants and other NPCs in the next segment.

3.5 -- NPCs

To look at the NPCs, open up module_troops.py. The various merchants and NPCs you see in the game are very similar to regular troops. The most significant element setting them apart is the flag *tf_hero*; this flag is what causes Marnid and Borcha to achieve their special status. Every unique NPC you encounter in the game is a Hero, even the merchants. The main differences between Heroes and regular troops are:

1) Heroes are unkillable. Their health is represented by a percentage value, and you can have only one of each Hero unless they are cloned by error or by design. Even by design, however, cloning Heroes is a bad idea.
2) Heroes each take up a full party stack.
3) Heroes show up properly in a scene when they are assigned to one in their troop tuple.
4) Heroes stay with the player when he is defeated by an enemy party -- the Heroes are not captured by the enemy -- but the player can capture enemy Heroes as normal.

Because there should be only one specimen of each Hero, they have no plural troop name. Field 3 of the Hero tuple is therefore identical to Field 2.

Example of a Hero tuple:

```
 ["npc2","Marnid","Marnid", tf_hero|tf_unmoveable_in_party_window, 0,reserved, fac_commoners,[itm_linen_tunic,itm_hide_boots,itm_club],
   str_7|agi_7|int_11|cha_6|level(1),wp(40),knows_merchant_npc|
   knows_trade_2|knows_weapon_master_1|knows_ironflesh_1|knows_wound_treatment_1|knows_athletics_2|knows_first_aid_1|knows_leadership_1,
   0x000000019d004001570b893712c8d28d00000000001dc8990000000000000000],
```

Here we have our friend Marnid, a faithful companion over the course of the game. He is marked as a Hero by the *tf_hero* in his Flags field. He also has the tf_unmoveable_in_party_window flag. This means that unless he is an enemy prisoner, you cannot garrison him or trade him to another party. We used to be able to find him at entry point 4 in the Happy Boar inn, but the new code has him randomly showing up at various taverns **(the code for this should be explained later : JIK)**. He's a bit of a pushover in combat, but his fairly good Trade skill comes in handy during our early adventures, and as a Hero he will never die unless removed by some scripted event. You will also note that Marnid has his own unique face code. It is possible to design faces using the in-game face editor and then retrieve their face codes for use in your module; this is covered in Part 7 of the documentation.

Another important thing to note is that Marnid's troop identifier in this file -- "npc2" -- . We must always reference the identifier with lowercase letters and without spaces. The module system will throw an error if you try to use an uppercase letter when referencing an identifier from another file. So, in order to reference "Marnid", we must use the identifier "trp_npc2".

We can now begin to make our own hero. Copy Marnid's tuple, and paste it at the top of the section marked off as "**#Add Extra Quest NPCs Below this point**". You can put it above the "new_troop" we created earlier, thus keeping the modifications that we have made together. Like stated earlier, it's a good idea to keep you code together ***where possible***. Some items of code have to be placed in a specific part of the module file.

As newer versions of M&B are released, Marnid might have different stats, but it will be similar to this below. Note that we have changed his identifier and his name to suit the new NPC we are going to add.

```
["npc17","Geoffrey","Geoffrey", tf_hero|tf_unmoveable_in_party_window, 0,reserved,
fac_commoners,[itm_courtly_outfit,itm_hide_boots,itm_club],def_attrib|level(6),wp(60),knows_trade_3|knows_inventory_management_2|knows_riding
_2, 0x000000019d004001570b893712c8d28d00000000001dc8990000000000000000],
```

In this example, we've changed the new hero's identifier to **npc17** (which is after ***npc16***, the last of the current list of native **npcs**) and names to "Geoffrey". Keeping with the convention of naming NPCs is a good idea to help you identify what they are for. If we were creating him to be a heroe that can join your party, like Marnid, it would be best to add him to the list of NPCs right after npc16. However, we'll be using him as an 'actor' in the quest we are going to be creating so he's fine where he is. In the code is fine. To make Geoffrey stand out more (and incase Marnid is present, they won't look like twins) let's change his itm_linen_tunic to itm_courtly_outfit. Please make this change for yourself now.

At this point, if we were to click on **build_module.bat**, the module system would compile without problems. Our new tuple does not conflict with anything that the module system can see. Yet if we did, we would have one major problem – It would be quite random as to where to find Geoffrey, as it is with the other NPCs. All heroes are found in random locations (with the flag 0). **(JIK NOTE: Cleaner explination needed, as you may not want your NPCs to spawn at taverns at all. Should find out how this is handled)**

In order to solve this, we will assign Geoffrey to entry point 1 in town_6 which is Praven instead. Praven is near the middle of the map, so for a random start, it's the easiest place to find.
Your tuple should now look like this:

```
  ["npc17","Geoffrey","Geoffrey", tf_hero, scn_town_6_tavern|entry(1),reserved,
   fac_commoners,[itm_courtly_outfit,itm_hide_boots,itm_club],
   def_attrib|level(6),wp(60),knows_trade_3|knows_inventory_management_2|knows_riding_2,
   0x000000019d004001570b893712c8d28d00000000001dc8990000000000000000],
```

Feel free to assign him some new equipment or stats, as we did with "new_troop". Then click on **build_module.bat**, open the game, and go to the tavern in Praven. If all went welll, you should now find Geoffrey standing in the tavern. Walk around and find him.

Attempting a dialogue with Geoffrey will cause him to give a stock response (mercenaries wanting to join your party), as he currently has no dialogue associated with him. This is another example of *interrelation*. He uses this dialog based on where he is in the **module_troops.py** file, which is defined in **module_constants.py**. We will talk about that later.

We will leave Geoffrey where is for now, and come back to him later as we explore making scenes, module_quests and module_dialogs. Before closing out this section we will add one more "actor" to the soon to be created quest. Search for constable_hareck. His tuple looks like this:

```
  ["constable_hareck","Constable Hareck","Constable Hareck",tf_hero, scn_zendar_center|entry(5),reserved,
   fac_commoners,[itm_leather_jacket,itm_hide_boots],def_attrib|level(5),wp(20),knows_common,0x00000000000c41c001fb15234eb6dd3f],
```

First we will copy him down to just below Geoffrey. This will make it easier for us to make changes if we need to. Since he is no longer used in the main game, we can mess around with him. Later we will give him a new face (the face code is old) and place him in a scene. For now it's just important to know that he will be part of our upcoming quest. We will be moving them both to the tavern in Mod Town, but before we can do that we would have to create the scene and add entry points. Let's just make him unique by changing his ID to "hareck", and his location to 0.

```
  ["hareck","Constable Hareck","Constable Hareck",tf_hero, 0,reserved,
   fac_commoners,[itm_leather_jacket,itm_hide_boots],def_attrib|level(5),wp(20),knows_common,0x00000000000c41c001fb15234eb6dd3f],
```

3.6 – Merchants – This has stayed the same from Winter's tutorial. Have yet to play with merchants

Merchants are a special type of Hero. In addition to *tf_hero*, they also have the flag *tf_is_merchant*. This flag causes them to not equip any of the items in their inventory, except what they are originally assigned in their troop tuple. In other words, these merchants can receive all sorts of items over the course of the game, but they will not wear or use the items, and the items will properly show up for sale.

Example of a merchant:

```
  ["zendar_weaponsmith","Dunga","Dunga",tf_hero|tf_is_merchant, scn_zendar_center|entry(3),0, fac_commoners,
   [itm_linen_tunic,itm_nomad_boots],
   def_attrib|level(2),wp(20),knows_inventory_management_10, 0x00000000000021c401f545a49b6eb2bc],
```

This was the weapon merchant in Zendar, named Dunga. He is almost identical to the other merchants in Native M&B. If you look closely, the only differences are their identifiers, names, scene placement, and faces.

NOTE: again, this is not tested. Would advise putting new merchants just before the "merchants_end" tuple.
To add a merchant, however, can be slightly complex. They are gathered into groups for a reason. There are scripts in M&B to update merchant inventories every day for each type of merchant -- to do this, these scripts use a *range*, a number of subsequent tuples between a starting point of choice (the lower bound) and a stopping point of choice (the upper bound). For example, the range of armour merchants includes everything from "zendar_armorer" (the lower bound) up to -- **but not including** -- "zendar_weaponsmith" (the upper bound). The upper bound of a range is not included in the range, so the upper bound needs to be set one entry further down (to "zendar_weaponsmith") if we want "town_14_armorer" to be included in the armour merchant range.

For this reason, new armour merchants must be added before "zendar_weaponsmith". New weapon merchants must be added before "zendar_tavernkeeper". New goods merchants must be added before "merchants_end".

3.7 -- Chests

Chest troops are special troops which serve as inventories for chests inside the game with which the player can interact. These chest troops are **not** the chests themselves, only their inventories. Chests as you see them in the game are part scene prop, part scene information, part troop and part hardcoded. New chests are somewhat complex to create and cross various module files; here we will cover only the information relevant to module_troops.

Example of a chest:

```
  ["zendar_chest","zendar_chest","zendar_chest",tf_hero|tf_inactive, 0,reserved,  fac_vaegirs,[],def_attrib|level(18),wp(60),knows_common, 0],
```

All chests must follow this example. The only things you should consider changing on a new chest troop are the troop's name and identifier, (possibly) troop level and troop skills, and the inventory. As mentioned, chest troops serve as inventory for in-game chests; therefore, any items you add to the chest troop's inventory will be inside the chest at the start of the game.

Chests require a chest troop to function. However, they also require several other modifications to different module files, which we will cover in the files' respective documentation.

Having learned this, you now know all there is to know about module_troops. There is a list of available flags in header_troops.py that you can use for the creation of further troops. Feel free to experiment, and when you're ready, please move on to the next part of this documentation.

PART 4

In Part 2 of this documentation, we learned how to make new *parties*; unique locations on the map. They are not to be confused with *party templates*, which we will be addressing here.

In the simplest terms, party templates are a set of guidelines from which parties on the map are spawned. This marks the most notable difference between *parties* and *party templates* -- parties are unique entities on the map, whereas templates do not physically exist in the game world. They serve only as a list of guidelines from which to spawn parties. Therefore, certain operations that use a party_id for input will not work when they are fed a party_template_id.

Parties that are spawned from a template do not have to be unique. There can be many parties of the same template; each will have a random number of troops depending on the player's level and the minimum/maximum troop limits defined in the template.

4.1 -- Breakdown of Module_Party_Templates

The file begins with the usual Python list: party_templates = [, followed by several templates that are hardwired into the game and should not be edited. You'll notice that the tuples in module_party_templates are very similar to those in module_parties, but the two are not interchangeable.

Example of a party template:

```
  ("village_farmers","Village Farmers",icon_peasant,0,fac_innocents,merchant_personality,[(trp_farmer,5,10),(trp_peasant_woman,3,8)]),
```

Here is a template we've all encountered in-game. Parties of this template will be called "farmers", they will automatically start dialogue when you encounter one, they will behave cowardly in-game, and they each have two troop stacks made up of farmers(5 to 10 in numbers) and peasant women(3 to 8 in numbers).
(***Seems that they have dropped the flag pf_auto_start_dialog, maybe due to the fact that this is for all parties?***)

Breakdown of the tuple fields:

1) Party-template id. Used for referencing party-templates in other files.
2) Party-template name. The name that parties of this template will use.
3) Party flags. See header_parties.py for a list of available flags.
4) Menu. Deprecated as in module_parties. Use the value 0 here.
5) Faction.
6) Personality. This field contains flags which determine party behaviour on the map.
7) List of stacks. Each stack record is a tuple that contains the following fields:
    7.1) Troop-id.
    7.2) Minimum number of troops in the stack.
    7.3) Maximum number of troops in the stack.
    7.4) Member flags(optional). You will have to add an extra field in order to set member flags. For example:
        (trp_swadian_crossbowman,5,14,pmf_is_prisoner)

There can be at most 6 stacks in a party template.Village Farmers tuple examination:

1) Party-template id = "village_farmers"
2) Party-template name = "Village Farmers"
3) Party flags = icon_peasant
4) Menu = 0
5) Faction = fac_innocents
6) Personality = merchant_personality
7) List of stacks:
    7.1) Troop-id = trp_farmer, trp_peasant_woman
    7.2) Minimum number of troops in the stack = 5, 10
    7.3) Maximum number of troops in the stack = 3, 8
    7.4) Member flags(optional) = None are set.

If you've followed the documentation since Part 1, you should be fairly adept at reading tuples by this point, and you will have noticed the one field in this tuple that's unlike any other field we've encountered before: Field number 6, the Personality flags field. We will cover this field and its functions in the next segment.

4.2 -- Personality

As mentioned in the tuple breakdown, the Personality field determines party behaviour on the map. Here you can assign custom scores for Courage and

Aggressiveness, or use one of the preset personalities such as merchant_personality. These presets are constants, each containing a Courage and an Aggressiveness score. The presets are all defined in **header_parties.py**, so open that file now and scroll to the bottom to see the constant definitions for yourself. There you will also notice the list of possible Courage and Aggressiveness settings.

The constant merchant_personality is used in many templates throughout the file. Parties with this personality will be friendly, they will not go out to attack the enemy or raid weaker parties. This is because merchant_personality sets the party's Aggressiveness to aggresiveness_0. A party with aggresiveness_0 will never attack another party, whereas normal combative parties with the preset soldier_personality will have aggresiveness_8. This will let them attack other parties if the attackers' faction is on bad terms with the defenders' faction, and if the would-be attackers aren't badly outnumbered.

Courage is the score that determines when parties will run away from another, larger party. Higher Courage means they will be less quick to turn away when the numbers aren't entirely favorable. merchant_personality parties have a Courage of 8, where soldier_personality have a Courage of 11.

These settings scale from 0 to 15, allowing you to precisely set the desired behavior for your party templates. New modders, however, are recommended to stick with the presets. They cover the full range of personalities you should need for your first mod.

Finally, for bandit templates, there is the flag banditness. This causes the bandit party to constantly consider other nearby parties as prey, and if the prey is carrying significant amounts of gold and/or trade goods, the bandit party will attack. Ideally, a bandit party should have low aggressiveness or low troop numbers so that it does not attack soldier parties. **NOTE:** You may not see the banditness flag here, but it is listed in **headers_parties.py** as part of **bandit_personality**.

4.3 -- Creating New Templates

Copy the tuple for "farmers" and paste it at the bottom of your file, before the closing bracket.

```
  ("new_template","new_template",icon_peasant,0,fac_innocents,merchant_personality,[(trp_farmer,5,10),(trp_peasant_woman,3,8)]),
```

In this example we've changed the identifier from "farmers" to "new_template", and we've done likewise for the name. Once we've done this, we can begin to edit the specifics of this template.

For our first tweak, let us change the template's faction to fac_neutral, and merchant_personality to soldier_personality.

```
  ("new_template","new_template",icon_peasant,0,fac_neutral,soldier_personality,[(trp_farmer,5,10),(trp_peasant_woman,3,8)]),
```

From now on, parties of this template will be of the Neutral faction, and they will attack an enemy party if they see one. Next, let's play with the troop composition.

```
  ("new_template","new_template",icon_peasant,0,fac_neutral,soldier_personality,[(trp_npc17,1,1),(trp_new_troop,2,9)]),
```

This example has a few notable changes -- most importantly, it's now being led by trp_npc17(*Geoffrey*), the Hero troop we created in Part 3 of this documentation. Since there can't be more than one of Geoffrey, we've changed his minimum and maximum troop numbers to 1.

As his followers, we've assigned a contingent of trp_new_troop, the regulars we also created in Part 3. There will be never be less than 2 "new troops" in this party, but as the player gains in level, the size will be gradually adjusted. Eventually this "new_template" can spawn with as many as 9 "new troops" -- but never more than 9.

Save the file and click on **build_module.bat**. If everything went well, you will now be able to use the new template in your module code, but parties of a party template must be spawned -- they do not just show up of their own volition. If we were to run the game at this point, we would not see any parties of our "new template" running around.

We will learn how to spawn parties of a template in an upcoming part of this documentation. For the moment, let us leave Geoffrey and his little band while we find out how to create new items in the next part of this documentation. Please move on to Part 5 now.

PART 5

In Parts 3 and 4, we learned how to create and equip new troops and party templates. With this background in place, we can now study how to make new items for our troops to use.

5.1 -- Breakdown of Module_Items

module_items.py begins with a number of constants that are used to govern *item modifiers*, which adjust the items we see in-game. Bent polearms, chipped swords, heavy axes, they are all created from a tuple in module_items and then given an appropriate item modifier to adjust their stats up or down.

The constants defined here consist of standard item modifiers. The items you find in merchants and in loot will draw their modifiers randomly from these constants. Modifiers that are not listed in an item's modifier constant will not be considered for merchant inventory or battle loot.

For more experienced modders, it is interesting to note that modifiers which aren't listed in the appropriate constant can still be used with the "troop_add_item" operation. For example, longbows usually come only in "plain", "bent" and "cracked" forms; but if we were to add a longbow to the player's inventory with the modifier "balanced" on it, the player will receive a balanced longbow.

After the constants, the list of tuples begins. The first one of interest is the weapon "practice_sword", which will function as a good example. Let’s look at this tuple:

["practice_sword","Practice Sword", [("practice_sword",0)],
itp_type_one_handed_wpn|itp_primary|itp_secondary|itp_wooden_parry|itp_wooden_attack, itc_longsword,
3,weight(1.5)|spd_rtng(103)|weapon_length(90)|swing_damage(16,blunt)|thrust_damage(10,blunt),imodbits_none],

This is a basic practice weapon, used in the arena fights and training fields. Breakdown of the tuple fields:

1) Item id. Used for referencing items in other files.
2) Item name. Name of item as it will appear in inventory window
3) List of meshes. Each mesh record is a tuple containing the following fields:
    3.1) Mesh name. The name of a 3d model in the game or module's resource files.
    3.2) Modifier bits that this mesh matches. A list of item modifiers that will use this mesh instead of the default. The first mesh in the list is the default.
4) Item flags.
5) Item capabilities. This field contains a list of animations that this item can use.
6) Item value. The base value in denars. Note, the actual value of the item will be much higher in-game unless the player has a trade skill of 10.
7) Item stats. This is where the statistics of the item are defined; weight, abundance, difficulty, armour ratings, etc.
8) Modifier bits. Modifiers that can be applied to this item. They are listed at the top of the module_items.py file
9) [Optional] Triggers. A list of simple triggers to be associated with this item.

Practice_sword tuple examination:

1) Item id = "practice_sword"
2) Item name = "practice_sword"
3) List of meshes:
    3.1) Mesh name = "practice_sword"
    3.2) Modifier bits = 0
4) Item flags = itp_type_one_handed_wpn|itp_primary|itp_secondary|itp_wooden_parry|itp_wooden_attack
5) Item capabilities = itc_longsword
6) Item value = 3
7) Item stats = weight(1.5)|spd_rtng(103)|weapon_length(90)|swing_damage(16,blunt)|thrust_damage(10,blunt)
8) Modifier bits = imodbits_none
9) Triggers = None.

We can now tell all of "practice_sword"'s details from its tuple:

- It uses the mesh "practice_sword" as its default mesh.
- It uses item flags that mark it as a one-handed melee weapon. Troops who are equipped with "practice_sword" will consider "practice_sword" when choosing a primary melee weapon from their inventory listing. They will also consider "practice_sword" when choosing a backup (secondary) weapon. (Note: The secondary weapon function is currently nebulous. It's unclear when it's used or if it's used at all. However, melee troops certainly will not switch to different melee weapons during combat.). *I have yet to find out what wooded_parry and wooden_attack mean.*
- It has all the animations defined in the constant itc_longsword, so "practice_sword" is able to be used as a long sword would be.
- It weights 1.5 kilograms. Its speed rating is 103, its weapon length is 90. It does a base 16 blunt damage while swinging, and a base 10 blunt damage while thrusting.
- It uses no item modifiers.

5.2 -- Damage Types

As we've observed in the tuple examination, "practice_sword" does blunt damage. Obviously this is because it's a practice weapon, but it provides a good incentive to look into the different damage types available in M&B.

First, there is ***cut damage***. Cut damage represents the slicing action of a sharp blade, such as a sword or an axe. Cut damage gets a bonus against unarmoured or lightly-armoured enemies, but conversely it has a large penalty against heavy armour. Cut damage will kill an enemy if it brings the enemy to 0 hit points.

Next we have ***blunt damage***. Blunt damage represents the bludgeoning effect of weapons without an edge, such as a mace or hammer. Blunt damage gets a 50% bonus against heavy armour, but blunt weapons are often shorter than cutting weapons and they do less damage overall. The largest advantage of blunt damage is that it knocks an enemy unconscious when the enemy is brought to 0 hit points, instead of killing him. Unconscious enemies can be captured and sold into slavery. Charging horses also do blunt damage.

Finally, ***pierce damage*** represents the penetrating tip of arrows, crossbow bolts and similar weapons. Pierce damage gets a 50% bonus against heavy armour, but piercing weapons usually do less damage overall in order to balance it with the other damage types. Pierce damage will kill an enemy if it brings the enemy to 0 hit points.

5.3 -- Creating An Item

Copy the "practice_sword" tuple and paste it at the bottom of the file, before the closing bracket. Once done, change the name and identifier of this new tuple to "new_mace".

["new_mace","New Mace", [("practice_sword",0)], itp_type_one_handed_wpn|itp_primary|itp_secondary|itp_wooden_parry|itp_wooden_attack,
 itc_longsword, 3,weight(1.5)|spd_rtng(103)|weapon_length(90)|swing_damage(16,blunt)|thrust_damage(10,blunt),imodbits_none],

The M&B items system is very flexible; it takes only a few small adjustments to turn a sword into a mace. In the case of "practice_sword", it was already set to do blunt damage. That makes our job even easier.

First, we change the item capabilities of our new mace from itc_longsword to itc_scimitar. This will cause our mace to lose the ability to thrust, because M&B's thrusting animation is not included in the itc_scimitar constant. Also change the item's mesh from "practice_sword" to "mace_pear". This is a mesh that is not used in the Native game, so by doing this we will be giving our new mace a fresh new look. Let's also drop "itp_wooden_attack", as other maces don't have this flag.

```
["new_mace","New Mace", [("mace_pear",0)], itp_type_one_handed_wpn|itp_primary|itp_secondary|itp_wooden_parry, itc_scimitar,
  3,weight(1.5)|spd_rtng(103)|weapon_length(90)|swing_damage(16,blunt)|thrust_damage(10,blunt),imodbits_mace],
```

In this example, we have changed the mesh as planned, and also switched the modifier bits from imodbits_none to imodbits_mace. This will allow our new mace to use all the modifiers specified in the constant imodbits_mace at the beginning of the file.

There are only two further changes we need to make to finish up this item. Notably, we're going to increase its swing damage, and we're going to give it an additional item flag.

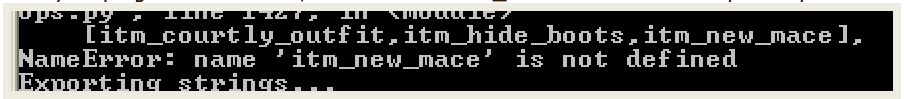
Observe:

```
["new_mace","Geoffreys mace", [("mace_pear",0)] , itp_type_one_handed_wpn|itp_primary|itp_secondary|itp_wooden_parry|itp_unique, itc_scimitar,
 3,weight(1.5)|spd_rtng(103)|weapon_length(90)|swing_damage(26,blunt)|thrust_damage(10,blunt),imodbits_mace],
```

As you can see, we have upped the swing damage from 16 to 26, which will make our new mace a good deal more dangerous in a fight. And, more notably, we've added the flag itp_unique to the Flags field. An item with itp_unique cannot be looted via the normal post-battle loot screen. This will keep the player from getting his hands on it too early, because we have plans for this mace.

For our last adjustment, change the item name from "New Mace" to "Geoffreys_mace". Then, open **module_trooops.py** and replace the itm_club in Geoffrey's inventory with itm_new_mace.

Save your progress in both files, then click on **build_module.bat**. You will probably notice an error that **"itm_new_mace"** is not defined:



```
ops.py", line 1127, in <module>
     [itm_courtly_outfit,itm_hide_boots,itm_new_mace],
NameError: name 'itm_new_mace' is not defined
Exporting strings...
```

This shows us that the order in which the files are compiled may not be the order in which you work on them. It would seem that module_troops.py is compiled before module_items.py. Because of this, our new mace was not yet seen by the code. If we run the **build_module.bat** again, the error will be gone, since module_items.py still compiled even though there was an error earlier on in the compile process. Let this be a lesson to you that you should always compile after working on module file, especially if what you have added is needed in the next module you will be working on.

Congratulations! You've made a brand new item and added it to a troop's inventory. The troop in question is a Hero, so he will always have the new mace in his inventory, and he will always use the best weapon in his inventory that he's able to use.

Regulars, on the other hand, will choose their equipment at random from their inventory list. This is why most regulars have a very varied selection -- otherwise they will all look the same.

Now that we know how to create new items, we can take a look at the various different statistics and examine what they all mean.

5.4 -- Item Stats

In this segment you will find a comprehensive list of stats and a breakdown of their function. As some stats mean different things for different item types, I have organized the list by item types.

Generally:

abundance -- Percentage value.
This stat governs how often the item will appear in merchant inventories and combat loot. 100 is standard; can be more or less than 100 (down to 0), as 100 does not mean that it will show up 100% of the time. Currently I do not know the ceiling number that equals 100%.

weight -- Kilogramme value.
Defines the weight of the item in kilogrammes.

itp_type_horse
body_armor -- Value.
Determines the horse's armour rating and number of hit points. A higher value means more armour and more hit points.

difficulty -- Value.
Determines how high the player's Riding skill needs to be to be able to mount this horse.

horse_speed -- Value.
The horse's speed on the battle map. Higher values make faster horses.

horse_maneuver -- Value.
The horse's manoeuvrability on the battle map.

horse_charge -- Value.

Determines how much damage the horse will do when charging infantry, and how much speed the horse will lose during each collision with an infantryman. Higher values will allow horses to do more damage and wade through more infantry.

itp_type_one_handed_wpn
difficulty -- Value.
The minimum STR score needed to be able to use this weapon. If a troop does not have greater or equal STR, he will not be able to wield it.

spd_rtng -- Value.
The attack speed of the weapon, both swing and thrust.

weapon_length -- Centimetre value.
The length of the weapon in centimetres. This stat determines how far the weapon will be able to reach in-game, regardless of the mesh size.

swing_damage -- Value, damage type.
The base damage and damage type of the weapon when performing a swing attack.

thrust_damage -- Value, damage type.
The base damage and damage type of the weapon when performing a thrust attack.

itp_type_two_handed_wpn
Same as itp_type_one_handed_wpn.

itp_type_polearm
Same as itp_type_one_handed_wpn.

itp_type_arrows
weapon_length -- Centimetre value.
The length of the arrow in centimetres.

thrust_damage -- Value, damage type.
The amount of damage this type of arrow adds to the bow's base damage, and the damage type.

max_ammo -- Value.
The number of arrows in one stack.

itp_type_bolts
Same as itp_type_arrows.

itp_type_shield
hit_points -- Value.
The base number of hit points for this shield.

body_armor -- Value.
The amount of damage subtracted from every hit to the shield,.

spd_rtng -- Value.
The speed with which the shield can be brought up into defensive mode.

weapon_length -- Value.
Shield coverage. Higher values allow the shield to cover more body area, offering shield protection from arrows to larger sections of the body.

itp_type_bow
difficulty -- Value.
The minimum Power Draw score needed to be able to use this bow. If a troop does not have greater or equal Power Draw, he will not be able to wield it.

spd_rtng -- Value.
The bow's reloading speed. IE, how fast a troop will be able to take an arrow from quiver, notch, and draw the bow again. Higher values will mean quicker reload times.

shoot_speed -- Value.
The speed at which ammunition from this bow flies through the air. Higher values will mean faster arrows; note, however, that very fast ammunition may clip through nearby enemies without hitting them.

thrust_damage -- Value, damage type.
The base damage and damage type inflicted by hits from this bow.

accuracy -- Percentage value.
The chance of a shot flying exactly at the troop's aiming point. 100 represents a 100% chance, lower values will greatly decrease the chance of a hit. This is not used on the Native bows and crossbows, but can be added.

itp_type_crossbow
difficulty -- Value.
The minimum STR score needed to be able to use this crossbow. If a troop does not have greater or equal STR, he will not be able to use it.

spd_rtng -- Value.
The crossbow's reloading speed. IE, how fast a troop will be able to take a bolt from quiver, notch, and pull back the string for firing. Higher values will mean quicker reload times.

shoot_speed -- Value.
The speed at which ammunition from this crossbow flies through the air. Higher values will mean faster bolts; note, however, that very fast ammunition may clip through nearby enemies without hitting them.

thrust_damage -- Value, damage type.
The base damage and damage type inflicted by hits from this crossbow.

max_ammo -- Value.
The number of bolts that can be fired from this crossbow before it must be reloaded.

accuracy -- Percentage value.
The chance of a shot flying exactly at the troop's aiming point. 100 represents a 100% chance, lower values will greatly decrease the chance of a hit. This is not used on the Native bows and crossbows, but can be added.

itp_type_thrown
difficulty -- Value.
The minimum Power Throw score needed to be able to use this weapon. If a troop does not have greater or equal Power Throw, he will not be able to use it.

spd_rtng -- Value.
The reloading rate for this weapon. IE, how fast the next piece of ammunition can be readied for throwing.

shoot_speed -- Value.
The speed at which this weapon's ammunition flies through the air.

thrust_damage -- Value, damage type.
The base damage and damage type inflicted by hits from this weapon.

max_ammo -- Value.
The number of weapons (IE, ammunition) contained in one stack.

weapon_length -- Centimetre value.
The weapon's length in centimetres.

itp_type_goods
food_quality
The impact a food item will have on party morale. Values above 50 will improve morale while the item is being consumed, lower values will lower morale.

max_ammo
The number of consumable parts for this item.

itp_type_head_armor
head_armor -- Value.
The amount of damage this piece of armour will prevent to the head of the troop.

body_armor -- Value.
The amount of damage this piece of armour will prevent to the body of the troop.

leg_armor -- Value.
The amount of damage this piece of armour will prevent to the legs of the troop.

difficulty -- Value.
The minimum STR required to wear this piece of armour.

itp_type_body_armor
Same as itp_type_head_armor.

itp_type_foot_armor
Same as itp_type_head_armor.

itp_type_hand_armor
Same as itp_type_head_armor.

itp_type_pistol
difficulty -- Value.
The minimum STR score needed to be able to use this pistol.

spd_rtng -- Value.

The pistol's reloading speed. IE, how fast a troop will be able to reload the pistol and aim it again.

shoot_speed -- Value.
The speed at which ammunition from this pistol flies through the air.

thrust_damage -- Value, damage type.
The base damage and damage type inflicted by hits from this pistol.

max_ammo -- Value.
The number of bullets that can be fired from this pistol before it must be reloaded.

accuracy -- Percentage value.
The chance of a shot flying exactly at the troop's aiming point. 100 represents a 100% chance, lower values will greatly decrease the chance of a hit.

itp_type_musket
Same as itp_type_pistol.

itp_type_bullets
Same as itp_type_arrows.

PART 6

In this part of the documentation, we examine the smallest module files in the system, files with the least overall impact on your module. However, these files can still be very useful in many ways. By the end of this chapter, you will know how to get the most out of each of them.

6.1 -- Module_Strings

module_strings.py is arguably the simplest file in the module system. It contains *strings* -- blocks of text which can be displayed in various ways. Strings are used all throughout the module system, from module_dialogs to module_quests; whenever a block of text needs to be displayed on the screen. Module_strings, however, is a repository for independent strings, which aren't bound to any single file. Therefore they can be called by operations from anywhere in the module system.

You can see these strings used in various places in the game, as white scrolling messages at the lower left of your screen, or in a tutorial box, or even inserted into a game menu or dialogue sequence.

Much like module_factions, this file immediately begins with a Python list: strings = [. Again, the first few tuples are hardwired into the game and should not be edited.

Example of a string:

```
  ("bandits_eliminated_by_another", "The troublesome bandits have been eliminated by another party."),
```
or
```
  ("s5_s_party", "{s5}'s Party"),
```

Tuple breakdown:

1) String id. Used for referencing strings in other files.
2) String text.

One notable feature of strings is the ability to put a register value or even another string inside of it. You can do this by adding, for example, {reg0} in the string. This will display the current value of reg(0). {reg10} would display the current value of reg(10), and so on.

For additional strings, you must use the string register instead of a normal register; this is because strings are stored separately from registers. For example, {s2} would display the contents of string register 2 -- *not* the contents of reg(2). You can freely use string register 2 and reg(2) at the same time for different things, they will not overlap or interfere with each other.

In dialogue, it is also possible to display parts (or all) of a string's contents differently depending on the gender of the person being addressed. For example, inserting {sir/madam} into a string will cause the word "sir" to be displayed when the addressed person is male, and "madam" if the person is female.

All of these options (except gender-based alternation) will work perfectly in any kind of string field. Gender-based alternation only works in dialogue.

For the purposes of the operation "display_message", it is possible to display a string in various colours, by appending a hexadecimal colour code to the operation. For example, (display_message,<string_id>,[hex_colour_code]),

The HEX color code is mapped out as an RGB format such as(for the example of white = 0xFFFFFFFF):

0xFF	**FF**	**FF**	**FF**
Not sure what this is for, but start all colors with it	Red component from 00 (0) to FF(256)	Blue component from 00 (0) to FF(256)	Green component from 00 (0) to FF(256)

So to make RED you would use the code 0xFF**FF**0000 which would set red to 256, blue to 0 and green to 0. Knowing how to blend color will help if you with colors other than the norm.

6.2 -- Module_Factions

module_factions.py contains all the factions used by the module system and by M&B's artificial intelligence. Though a small file, it governs several important settings which we will cover here.

The file immediately begins with a Python list: factions = [. As in most of the module files, the first few tuples are hardwired into the game and should not be edited. If you scroll down just a bit, you will find the "deserters" tuple. Example of a faction:

```
  ("deserters","Deserters", 0, 0.5,[("manhunters",-0.6),("merchants",-0.5),("player_faction",-0.1)], [], 0x888888),
```

This is a short, simple tuple. It governs the "deserters" faction, whose major enemies are the "manhunters", "merchants", and "player_faction" factions. If a faction's relation with "deserters" is not defined anywhere in module_constants, it will automatically be set to 0, hence the two factions will be absolutely neutral to each other.

Breakdown of the tuple fields:

1) Faction id. Used for referencing factions in other files.
2) Faction name.
3) Faction flags.
4) Faction coherence. The relation between different members of this faction.
5) Inter-faction relations. Each relation record is a tuple that contains the following fields:
    5.1) Faction. Which other faction this relation is referring to.
    5.2) Relation. The state of relations between the two factions. Values range between -1 (bad) and 1(good).
6) Ranks
7) Faction color (default is grey)

"deserters" tuple examination:

1) Faction id = " deserters "
2) Faction name = "deserters"
3) Faction flags = 0
4) Faction coherence = 0.5
5) Inter-faction relations:
    5.1) Faction = "manhunters", "merchants", "player_faction"
    5.2) Relation = -0.6, -0.5, -0.1
6) no Rank is listed (**JIK note: Not sure what this is for at this time**)
7) the color of this faction is set the to the hex value of 0x888888 (Just like with strings, only the last 6 digits are the RGB values)

This faction has no flags, a neutral faction coherence, and 3 enemies defined in its tuple. However, if we look elsewhere, we can find that (for example) all "kingdom_#"s have a relation of -0.02 with "deserters", as defined in any of the kingdom tuples (such as "kingdom_1", the "kingdom of Swadia"). This is because a relation works two ways -- it only needs to be set once, and the value will count for both factions.

Now, create a new faction tuple by copying "deserters" and pasting it to the bottom of the list. Change the faction id and name to "geoffrey", and then remove all relations except for "player_faction" from the list. Also set the value to -1. Save the file and click on **build_module.bat**, just in case this is not compiled before the other changes we will be making. We will be using this faction definition in other module files.

Save your progress. Once you've done that, open **module_party_templates.py** and **module_troops.py**, and change the faction of the template "new_template" and the faction of the troop "npc17" to "fac_geoffrey". Save, and close the files. Click on **build_module.bat**. If all went well, both parties will now visibly have "Geoffrey" as their faction, and they will be hostile to the player.

6.3 -- Module_Constants

module_constants.py is a very simple file. It is filled with constants -- you should already know how these work from reading Part 3 of this documentation. The constants in module_constants are exactly the same as constants defined anywhere else, but constants which are used in multiple module files should be defined in module_constants so that the module system always knows where to find them.

Module_constants also serves as an organised, easily-accessible list where you can change the value of a constant whenever you might need to. Any changes will immediately affect all operations using the constant, so that you don't need to do a manual find/replace.

For example, changing hero_escape_after_defeat_chance = 80 to hero_escape_after_defeat_chance = 50 would immediately change any operations using hero_escape_after_defeat_chance, treating the constant as the value **50** instead of the value **80**.

This file also stores the range markers such as kings_begin = "trp_kingdom_1_lord" and kings_end = "trp_knight_1_1".

6.3a – Object Slots (JIK NOTE: Need to go more indepth on what Slots are)

Another powerful tool defined in the constants section is the use of **slots**. **Slots** can be assigned to any tuple object (troops, factions, parties, quest, etc) and can store a specific value that is used by all in that object type. Let's look at where the definition for quest **slots** are listed:

```
########################################################
##  QUEST SLOTS                          #############################
```

```
########################################################

slot_quest_target_center            = 1
slot_quest_target_troop             = 2
slot_quest_target_faction           = 3
slot_quest_object_troop             = 4
##slot_quest_target_troop_is_prisoner = 5
slot_quest_giver_troop              = 6
slot_quest_object_center            = 7
slot_quest_target_party             = 8
slot_quest_target_party_template    = 9
slot_quest_target_amount            = 10
slot_quest_current_state            = 11
slot_quest_giver_center             = 12
slot_quest_target_dna               = 13
slot_quest_target_item              = 14
slot_quest_object_faction           = 15

slot_quest_convince_value           = 19
slot_quest_importance               = 20
slot_quest_xp_reward                = 21
slot_quest_gold_reward              = 22
slot_quest_expiration_days          = 23
slot_quest_dont_give_again_period   = 24
slot_quest_dont_give_again_remaining_days = 25


########################################################
```

On the left side of the '=' are the names given to each slot. The slot number is on the right side of the '='. The names make it easier to know what the slot is used for. To be specific, "slot_quest_xp_reward" is used by native code to store how much XP you get from completing the quest. When referencing this value, you can either reference it as slot_quest_xp_reward, or by the number 21. When looking through the code it's easier to understand the name over the number.

As we get into creating the quest, we will be using some of these slots for our quest. This way, the value can be set in one place, and if it needs to be changed we only need to change it in one place. The other benefit is that if the player has more than 1 quest active, we don't have to worry that we might be over-writing variables we defined. Each quest (like all tuple objects) has its own slots.

6.4 -- Module_Quests

module_quests.py is the last of our small, simple files. It contains quests, including all the text related to those quests. Putting new quests here allows them to be activated via the module system, so that operations can read the quest's current status and use that status as a condition operation. It also defines a quest object, for which the before mentioned slots can be used. Example of a quest:

```
("deliver_message", "Deliver Message to {s13}", qf_random_quest,
  "{s9} asked you to take a message to {s13}. {s13} was at {s4} when you were given this quest."
  ),
```

This quest is one of many generic delivery request a king may give you.

Tuple breakdown:
1) Quest id = "deliver_message"
2) Quest Name = "Deliver Message"
3) Quest flags = qf_random_quest
4) Quest Description = "{s9} asked you to take a message to {s13}. {s13} was at {s4} when you were given this quest."

This quest uses string registers ({s9}, {s13}, and {s4}) to indicate who gave the quest({s9}), the recipient ({s13}), and where the target was last seen ({s4}). Looking at similar quests, it would seem that {s9} is always used as the requester's name, {s13} is the intended target troop/party and {s4} is used as their last location.

If you wish to make a quest without any flags, simply put a 0 in the flags field.

At this point, we make the first step in creating our own little quest. Copy the following quest tuple and paste it to the bottom of the Python list, just above the last entry ("quests_end"):

```
("mod_trouble", "Speak with the troublemakers in Mod Town", qf_random_quest,
  "Constible Haleck has asked you to deal with a bunch of young nobles making trouble around Mod Town.\
  How you want to tackle the problem is up to you; He has promised a small purse if you succeed."
),
```

Note the construction of this tuple. You can see, it takes only a few changes to make a new quest, but incorporating it into the game requires a lot more work. **Note: It seems that all quests listed here have the gf_random_quest flag, which may be needed**. Notice we use a '\' to break the length of the string. This does not add a carrage return in game, it just makes the code easier to read.

We need a place for you to get the quest, the ***"actors"*** that will play a part in the quest and some dialog to make it interesting. Earlier we created the actors (Hareck and Geoffrey). Next we will create the ***"scene"*** where you get your quest and where you confront the ***"target"*** of the quest. After that we will visit **module_dialogs.py** -- where we will be putting our new quest, troops and party templates to use.
Save your progress, close the file and click on **build_module.bat**. If all went well, you will be ready to move on to Part 7 of this documentation.

PART 7

Before we get into making our scene, let's look at getting rid of some minor game testing annoyances. For the most part, this section will focus on adjusting menus, but first there is an easier fix so we don't have to wander Caldara to get to our destination. The game is set to have you start near one of the 5 training grounds. You may know a way of starting at the one you want based on your selections, but let's make it so you start near Mod Town. If you don't want to do the work yourself, you can copy and paste this over the existing training grounds, but as changes may still be made to the code and new versions released, it's better to understand what I have done here and do it yourself:

```
  ("training_ground_1", "Training Field",
icon_training_ground|icon_training_ground|pf_hide_defenders|pf_is_static|pf_always_visible|pf_label_medium, no_menu, pt_none,
fac_neutral,0,ai_bhvr_hold,0,(-3,-3),[], 100),
#  ("training_ground_1", "Training Field",
icon_training_ground|icon_training_ground|pf_hide_defenders|pf_is_static|pf_always_visible|pf_label_medium, no_menu, pt_none,
fac_neutral,0,ai_bhvr_hold,0,(44,61),[], 100),
#  ("training_ground_2", "Training Field",
icon_training_ground|icon_training_ground|pf_hide_defenders|pf_is_static|pf_always_visible|pf_label_medium, no_menu, pt_none,
fac_neutral,0,ai_bhvr_hold,0,(-38, 81.7),[], 100),
#  ("training_ground_3", "Training Field",
icon_training_ground|icon_training_ground|pf_hide_defenders|pf_is_static|pf_always_visible|pf_label_medium, no_menu, pt_none,
fac_neutral,0,ai_bhvr_hold,0,(89.2, 16),[], 100),
#  ("training_ground_4", "Training Field",
icon_training_ground|icon_training_ground|pf_hide_defenders|pf_is_static|pf_always_visible|pf_label_medium, no_menu, pt_none,
fac_neutral,0,ai_bhvr_hold,0,(9.5 , -80.5),[], 100),
#  ("training_ground_5", "Training Field",
icon_training_ground|icon_training_ground|pf_hide_defenders|pf_is_static|pf_always_visible|pf_label_medium, no_menu, pt_none,
fac_neutral,0,ai_bhvr_hold,0,(-34.75, -30.3),[], 100),
```

A quick break down. I copied of the first training ground entry, and then changed its location to (-3,-3). This is close to the location of Mod Town, which is (-1, -1). Then I commented out all the other tuples. When making changes like this, it's a good idea to build the modules and test play the game. Changes like this may seem minor, but if other scripts in the game depend on these entries, the game may 1) not build, 2) may not load or 3) may crash at very specific instances. Always test the area in question if you change/remove things that were part of the native code.

These changes mean that there is only one training ground to spawn near, and it's close to our needed destination. If you don't want the training ground to show up at all you can add the flag pf_disabled to the list of flags. It will still exist to spawn near, but it will not be accessible when playing the game.

7.1 – Fast Character Creation (Designed by MartinF)

One thing that can slow down testing, is the drawn out character creation. It's great for the game, but it wastes a lot of time for testing you mod. Instead, we can preset our character with what stats we need, then start the game. Open up **module_game_menus.py**. Let's do a quick break down of the first tuple, "start_game_1":

1) Game menu ID (The prefix menu_ is automatically added before each game-menu-id) = "start_game_1"
2) Game-menu flags (int). See header_game_menus.py for a list of available flags = menu_text_color(0xFF000000)|mnf_disable_all_keys
3) Game-menu text (string) = "Welcome, adventurer, to Mount&Blade. Before…"
4) mesh-name (string). Not currently used. Must be the string "none"
5) Operations block. This executes when the menu is activated = []
6) List of menu options (list)
  6.1) Menu-option-id (string) used for referencing game-menus in other files. = start_male and start_female
  6.2) Conditions block (list). If the conditions are not met, the menu option will not be shown = [] and []
  6.3) Menu-option text (string). What the player sees as the option to click = Male and Female
  6.4) Consequences block (list). This executes when the option is chosen.

We will be inserting a new menu option between the male and female option. You can copy and paste this right above the female option:

```
############# MF for testing start ###################

    ("start_mod",[(eq,1,1),],"Quick Character (for mod testing)",
       [(troop_set_type,"trp_player",0),
        (assign,"$character_gender",tf_male),
        (troop_raise_attribute, "trp_player",ca_intelligence,-4), #so you dont need to wait time picking extra skills
        #(troop_raise_skill, "trp_player", skl_pathfinding, 5),
        #(troop_raise_proficiency, "trp_player",wpt_crossbow,40),
        #(party_add_members, "p_main_party", "trp_swadian_knight", 10),
        #(troop_add_item, "trp_player","itm_sword_medieval_a", imod_rusty),
        (change_screen_return, 0),
       ]
```

```
   ),
############## MF for testing end ####################

    ("start_female",[],"Female",
      [
          (troop_set_type,"trp_player",1),
```

When you start a new game, you will now have a new option between Male and Female called "Quick Character (for mod testing)". This option will only be visible if the condition is met. This condition (eq,1,1) may seem strange, but it's an easy way to turn the option on or off. Simply edit this file and change it to (eq,1,2). A simple change like this will make the condition fail, and the menu option will then be invisible. You may want to create your own comment at the top of the option stating how to turn it off. The menu option text is pretty straight forward, so we'll skip it and move on to the operations block.

Like in the original Male option, we follow the first 2 lines which gives the player the troop trp_player. Next we assign a gender. These 2 lines need to be set. Afterwards, we can then set up the player as needed. Reducing the players intelligence means less skill points to distribute, which makes the character editing scene go by faster. If you put the 4 attribute points (mandatory) into strength or charisma, you will not have any extra points to spread out in either skills or weapon proficiencies.

The next few lines I have commented out, but they are here to give you some shortcut ideas. If you need to test the player having a specific value is a skill or proficiency, you can set that here. If you want to test out a troop unit, or a special weapon, you can also set that here. The last line, (change_screen_return,0) finishes of the start up and moves you to the character creation screen.

You can also make testing in game functions easier as well. Simple things like adding XP, gold, troops and items, to calling scripts or mission_templates can be easily accessed from a game menu. A common menu to use is the built in camp menu. Search for You set up camp. You can then add in your own option to open up a mod testing menu:

```
("camp",mnf_scale_picture,
   "You set up camp. What do you want to do?",
   "none",
   [(assign, "$g_player_icon_state", pis_normal),
    (set_background_mesh, "mesh_pic_camp"),
   ],
   [
    ("camp_modding",[],"Go to the modding menu.",
       [(jump_to_menu, "mnu_camp_modding"),]
    ),

   ("camp_action_1",[(eq,"$cheat_mode",1)],"Cheat: Walk around.",
```

This alone will just take the player to an new menu option. We will create that new menu now. This we will make at the bottom of the **module_game_menus.py** file:

```
############## MF for testing start ####################
  ("camp_modding",0,"Select an option:","none",[],
     [("camp_mod_1",[],"Increase player's renown.",
         [(str_store_string, s1, "@Player renown is increased by 100. "),
          (call_script, "script_change_troop_renown", "trp_player" ,100),
          (jump_to_menu, "mnu_camp_modding"),
         ]
      ),
### MF - change attributes and skills below, or add weapon proficiencies with (troop_raise_proficiency, "trp_player", wpt_). See header.troops.py for options
      ("camp_mod_2",[]"Raise player's attributes and skills.",
          [(troop_raise_attribute, "trp_player",ca_strength,20),
           (troop_raise_skill, "trp_player",skl_riding,10),
           (jump_to_menu, "mnu_camp_modding"),

### MF - Spawn any party you want near your party. Look in party_templates.py for pt_id
      ("camp_mod_3",[],"Spawn a party nearby",
          [(spawn_around_party, "p_main_party", "pt_looters"),
           (display_message, "@Party spawned nearby."),
          ]
      ),
        ("camp_mod_4",[],"Back to camp menu.",
       [(jump_to_menu, "mnu_camp"),
        ]
       ),
      ]
  ),
############## MF for testing start ####################
```

As you can see, many things can be tested here, but I've given you 3 different ones. The first will boost your renown. This can be done with XP or gold as well. The next section plays with some of your stats. Here we increase strength and the riding skill. If you want to be creative, you can have a menu button for each stat to either increase or decrease it. 3rd, we spawn a party near the player. This can be useful in testing battles or AI responses.

We can make more to give you troops, companions, items, start conversations, and more. This is one of the more useful applications for starter Modders.

The key that you should take out of this is that you can access a menu with the command **(jump_to_menu, "mnu_<*your menu name*>")**. This can be called from just about any operations block. (Thanks again to MartinF for this section)

PART 8

In this part of the tutorial we will be setting up the scene for our quest, which will involve going into the game and using the in-game edit mode. Like I said earlier, we'll also use this mode to give a new face to Constable Hareck. However, before we can do all that we have to finish building Mod Town. So far we've created it and placed it on the map, but there are a few more steps to take before we have a fully functioning town with scenes which we can edit inside the game.

8.1 – Adding a Scene entry

Let's start by defining a Tavern for our town. Scenes are store in **module_scenes.py**. As it is, random town scenes will be subbed in (which you might have noticed if you tried to enter Mod Town). Let's take a look at one of the other tavern_scenes:

```
  ("town_7_tavern",sf_indoors, "interior_town_house_f", "bo_interior_town_house_f", (-100,-100),(100,100),-100,"0",
    ["exit"],[]),
```

By now you should be familiar with the dividers between the items in each tuple (your friend the ','). You should also know to find the tuple definition at the top of each file. We'll list it once again for module_scene.py:

```
 1) Scene id {string}: used for referencing scenes in other files. The prefix scn_ is automatically added before each scene-id.
 2) Scene flags {int}. See header_scenes.py for a list of available flags
 3) Mesh name {string}: This is used for indoor scenes only. Use the keyword "none" for outdoor scenes.
 4) Body name {string}: This is used for indoor scenes only. Use the keyword "none" for outdoor scenes.
 5) Min-pos {(float,float)}: minimum (x,y) coordinate. Player can't move beyond this limit.
 6) Max-pos {(float,float)}: maximum (x,y) coordinate. Player can't move beyond this limit.
 7) Water-level {float}.
 8) Terrain code {string}: You can obtain the terrain code by copying it from the terrain generator screen
 9) List of other scenes accessible from this scene {list of strings}.
    (deprecated. This will probably be removed in future versions of the module system)
    (In the new system passages are used to travel between scenes and
    the passage's variation-no is used to select the game menu item that the passage leads to.)
10) List of chest-troops used in this scene {list of strings}. You can access chests by placing them in edit mode.
    The chest's variation-no is used with this list for selecting which troop's inventory it will access.
```

You will also notice some of the specific definitions for the towns. It's always a good idea to look at these sections before editing a module, as it can point you to useful information, in this case "See header_scenes.py for a list of available flags".

Let's copy "town_7_tavern" to the end of the tavern list, right below "town_18_tavern". Make the following changes:

```
  ("town_mod_tavern",sf_indoors,"interior_town_house_f", "bo_interior_town_house_f", (-100,-100),(100,100),-100,"0",
    ["exit"],[]),
```

Really, all we need to do is make a different reference ID for it. It is important that it be in the same order as the town listings, as they are populated in sequence. This will also signify it as a part of Mod Town. We're using the tavern from Uxkhal (town_7) which is already mostly complete so we'll only have to add a few things to it. Run **build_module.bat**.

If you were curious before and visited the various scenes in Mod Town, you would have noticed that the scene background itself did not match the scene you had chosen from the menu. This is because the scenes for each town (and castle, and village) are set by a loop in **module_script.py**. It will loop through all available scenes in the same order as the towns themselves. This is why it is important to keep everything ordered the same. If you visit the arena in Mod Town, it will look like the dungeon for town_1 as this is the next scene after town_18_arena. Since Mod town would be the 19th town, it gets the scene right after the 18th scene.

8.2 – The In Game Scene Editor

Hopefully we're still getting clean compiles. As we go to start M&B, we need to set edit mode. From the Module selection screen, click on Configure, then click the Advanced tab. Check the checkbox for Enable Edit Mode. Note the warning. We will want to turn this off again once our scene is set, but unless we run into a big battle, things should be fine. Click on the Video tab and make sure that Start Windowed is checked. It's a good idea to change the screen resolution to a setting lower than that which you use for your computer. If you are not sure, use 800x600. Click OK at the bottom then start your mod.

While we are here let's make a new face code for Hareck. Start a new game, and when in the face generator screen, design the face you want to give to Hareck. When you have one you like, press Ctrl-E. The face code will appear in a box above the face. Click on it to copy the code to the clipboard. Paste it somewhere you'll find it later. In case you lose it or don't want to play around with it, here's the code I made:

```
0x0000000e8000558036db6db6dbefb6db00000000001db6db0000000000000000
```

Start the game and head over to Mod Town and check out the tavern. You may get an error that it cannot load scene objects. We don't have any yet in this "scene". Wow, it's dark in here… You will also notice that everyone who should be in the tavern is occupying the same spot. This is far from functional, so we need to EDIT the scene. While Edit Mode is enabled, pressing Ctrl-E will start up the in game edit function. Now you see why we needed to be in windowed mode? The tool bar you see will help you do what you need to do. I won't go into full details on all aspects of Scene Edit mode since I am no expert. I will help you put in what is needed to make your tavern look reasonable, and functional for our quest. First, some basic controls:
Move around the scene using the same movement keys as in M&B (A,W,D,S). Rotate the camera by holding the Left mouse button and moving the mouse. Select objects in the scene by clicking the right mouse button. Note that movement is always in the direction of the camera. So if you are looking at the ground and push W, you will go straight into the ground. Move around a bit to get the feel of it. Now our scene will need some objects to function properly, and others can be added for look.

Things you will need to add:
Entry points
Passage (scene exit)
Lights (more to be able to see the scene better than any other reason)

We will add a few other items just to fill out the scene better, but first Entry Points. Looking at scenes of other taverns, you will need an entry point for our tavern keeper as well as one for the player to enter on. They seem to stick with entry point 9 for the tavern keeper, and 0 for the player entry. We will do the same just in case there are reasons for that. On the Edit Mode tool bar, click on the item selection drop down box next to the Add Object button. Select Entry Point. Make sure Entry Point is highlighted in the window just below it. Now if we click the Add Object button, the mouse will have a yellow arrow following it. Move it to the middle of the floor and click the right mouse button to place it. If you move the cursor, you will find that a new yellow arrow is left there. Click the add button again to toggle off the add mode.

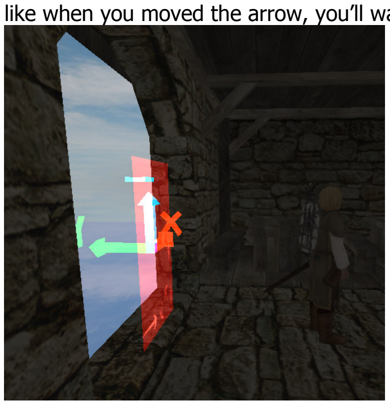
In the window above the add section you will see that you have added Entry Point 1. Just above that is some stats for the entry point. If we right click on the arrow in the game window (after turning the Add Object mode off), it will be selected. We can then have it follow the mouse by holding down the G key. If we move it behind the bar, you will see that it seems to go into the floor. We will need to raise it a bit. Holding down T will allow you to change the height of the selected object when the mouse is moved. Move it up till it can be seen. Lastly we want the tavern keeper to be facing out towards the crowd. Holding down the Z key will let you rotate it. There you have placed your first scene object! These commands can be seen by clicking on the Help button near the top right of the Edit Mode tool bar.

Since we want this to be our tavern keeper's spot, and that is generally entry point 9, we will change the Entry No: to 9. You may not see the change right away, so let's save this first. Click the close button. You will be given the option to save or not. Save it. Leave the tavern and come back. Enter Edit Mode again. You will see (in the scene objects window) that the Entry Point is now #9. Also set up an entry point 0 by the door facing in. Remember that the arrow will point the way the item on it is place. Entry Point 0 is where you enter the room.

At this point you should scatter some more Entry Points around the room. This will be for the visitors to the tavern such as mercenaries, heroes, book sellers, etc. They should be numbered starting at 16 and ending at 24. These entry points, including 0, are hard coded. Entry point 9 is not hard coded, but it is the default used by native, so we will conform to that standard.

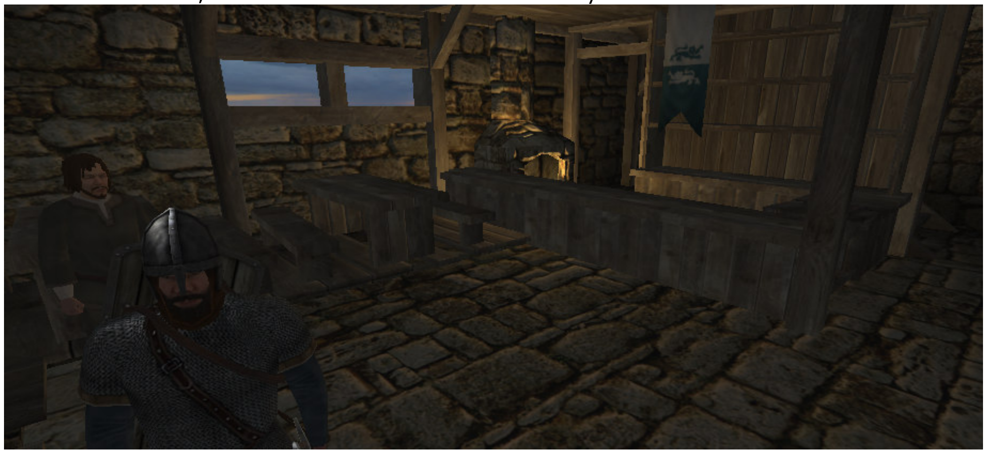
Put 2 more in for our actors, most likely across the room from each other. Number these last ones 5 and 6. I put Entry Point 5 by the door (for the constable) and Entry Point 6, just up the stairs in the back 6 (for Geoffrey). ***In doing this, I will be scripting that Geoffrey is upstairs in my dialogs***. Besides, the downstairs room is crowded enough already!

Looking around you might have noticed that big opening into nothing. There should be a door there, but you might also want to put an exit Passage there as well. Although you don't really need it for the purposes of this tutorial, it will make your scene more real to have a working door and you might also want to use it in other quests. Passages allow you to move between scenes without going back to the map or other menus. In the item selection drop down, we will now select Passage and click the Add button. If you don't see the icon right away, move the cursor over the wall, or into the middle of the floor. It looks kind of odd, but right click to place one and turn off the Add Object toggle. That's better. It now looks like a red rectangle. Just like when you moved the arrow, you'll want to move it over to the door way. But have it on the inside of the door step.


(Passage marker just inside the door)

In the stats window for the Passage, we will change it's Entry No to 4 and its menu item # to 3. This is what I have seen for the other taverns, so let's conform to the norm. Lastly, we will need to put a door into the empty space, making it look more realistic. Check the Item type to Scene Props and select "panel_door_b". Click the Add Object toggle, put one in scene and turn off the Add Object Toggle. I had to raise it up a bit, but then I got it easily into the door frame.

Ok, the last thing to do is add some light. It is also part of the Scene Props and there are various lights. To make it a bit dramatic take the red light and put it in the fireplace behind the bar. Then find and add the fire_big Scene Prop in there. If you put in a light and then delete it, you may find that it seems to still be there. If you leave the scene and come back, the affect will disappear. You will also notice that the people that were in the tavern will now be taking up one of the spots you designated from 16-24. But where is the tavern keeper? Since tavernkeepers are unique NPCs, like merchants or heroes, we'll have to create him in the same way we created our other 'actors'.



If you want, feel free to add what other scene props you want. This is a fun tutorial, so have some fun with it. When you are done save it and exit. Back to the module system. Open up module_troop.py.

8.3 – Main Actors

So we have the scene and the actors, now let's put them together. Open up **module_troops.py** and scroll down to where you have Hareck and Geoffrey. We can now tell them where they need to be. I have also added Hareck's new face code. Here is what it should look like:

```
# Add Extra Quest NPCs below this point
##JIK - new troop entry
  ["new_troop","new_troop","new_troops",tf_guarantee_boots|tf_guarantee_armor|tf_guarantee_helmet,no_scene,reserved,fac_commoners,
   [itm_sword_medieval_a, itm_fighting_axe,itm_leather_jerkin,itm_skullcap,itm_hide_boots],
   str_9|agi_9|level(4),wp(80),knows_common,mercenary_face_1, mercenary_face_2],

  ["npc17","Geoffrey","Geoffrey",tf_hero|tf_unmoveable_in_party_window,scn_town_mod_tavern|entry(6),reserved,fac_geoffrey,
   [itm_courtly_outfit,itm_hide_boots,itm_new_mace],  def_attrib|level(6),wp(60),knows_trade_3|knows_inventory_management_2|knows_riding_2,
   0x000000019d004001570b893712c8d28d00000000001dc8990000000000000000],

 ["hareck","Constable Hareck","Constable Hareck",tf_hero,scn_town_mod_tavern|entry(5),reserved,fac_commoners,
   [itm_leather_jacket,itm_hide_boots],def_attrib|level(5),wp(20),knows_common,
   0x0000000e8000558036db6db6dbefb6db00000000001db6db0000000000000000],
```

Lastly, let's put a tavern keeper behind the bar. I just copied one of the other tavern keepers to the bottom of the tavern keeper list and made a few changes:

```
 ["town_mod_tavernkeeper","Mod_Tavern_Keeper","Mod_Tavern_Keeper",tf_hero|tf_randomize_face|tf_female, scn_town_mod_tavern|entry(9),0,
  fac_commoners,[itm_woolen_dress,        itm_nomad_boots],def_attrib|level(2),wp(20),knows_common, woman_face_1, woman_face_2],
```

If we go back into the game, we will find not only our actors, but the tavern keeper behind the bar. If you talk to any of them, you will get generic responses, mainly from Hareck and Geoffrey seeming to be mercs wanting to join you. This is because we haven't created any dialog for them yet, which is what we'll be doing in the next part.

PART 9

In this Part we will be examining **module_dialogs.py**, by far the largest file in the module system and one of the most important, as it contains all dialogue in Mount&Blade. Any new dialogue that you wish to create will go into this file. Understand that this file is read from top to bottom each time a dialog is started. When the criteria for a tuple is met, then it will be executed.

9.1 -- Breakdown of Module_Dialogs

******NOTE: Need clarification… Do the "anyone" dialogs always run? I thought only 1 tuple is run from the section…******
The module_dialog.py file is a bit more complex than it used to be. Looking at the first few tuples, you will see that the target "anyone" is used with the various start-dialog states (explained later). There are conditions then checked and registries, string registries and $variables are set. What may seem odd is that each one's dialog text states that the dialog is not displayed. These dialogs set up these pieces of information to be used further down in the file. This way, these code blocks only need to be typed in once, not in every conversation.
If the *dialog partner* does not fit the criteria for that tuple, then it is skipped. This makes it easier for you to add new generic NPCs, village elders, etc. like Marnid without having to worry about setting their conversations. Take a quick look at one of them near the top:

```
dialogs = [
  [anyone ,"start", [(store_conversation_troop, "$g_talk_troop"),
                     (store_conversation_agent, "$g_talk_agent"),
                     (store_troop_faction, "$g_talk_troop_faction", "$g_talk_troop"),
#                      (troop_get_slot, "$g_talk_troop_relation", "$g_talk_troop", slot_troop_player_relation),
                     (call_script, "script_troop_get_player_relation", "$g_talk_troop"),
                    (assign, "$g_talk_troop_relation", reg0),

                   (try_begin),
                     (this_or_next|is_between, "$g_talk_troop", village_elders_begin, village_elders_end),
                     (is_between, "$g_talk_troop", mayors_begin, mayors_end),
                     (party_get_slot, "$g_talk_troop_relation", "$current_town", slot_center_player_relation),
                   (try_end),
```

There is more, but this will give you the idea. This will be checked when you "start" a conversation with "anyone". Looking at the **(try_begin)** block, we see that it checks if "$g_talk_troop" is listed between the constants "village_elders_begin" and "village_enders_end". Because the *negation flag* this_or_next is used, this condition and the next condition are read as an OR condition. This would mean if the first condition is not met, but the second condition is (is_between, "$g_talk_troop", mayors_begin, mayors_end), the rest of the code block is still executed.

The *negation flags* and the constants are defined in **module_constants.py**. Thats why, if you want to add a village elder, he/she should be placed between "trp_village_1_elder" and "trp_merchants_end" as defined in that file to take advantage of the settings pre-defined for village elders.

Let's scroll down a bit, and find a more specific dialog, namely the start conversation with Ramun the slave trader:

```
  [trp_ramun_the_slave_trader, "start", [
   (troop_slot_eq, "$g_talk_troop", slot_troop_met_previously, 0),
   ], "Good day to you, {young man/lassie}.", "ramun_introduce_1",[]],
```

Tuple field breakdown:

1) Dialogue partner: This should match the person player is talking to, usually this is a troop-id. You can also use a party-template-id by appending '|party_tpl' to this field. Use the constant 'anyone' if you'd like the line to match anybody. Appending '|plyr' to this field means that the actual line is spoken by the player. Appending '|other(troop_id)' means that this line is spoken by a third person on the scene. (You must make sure that this third person is present on the scene)
2) Starting dialog-state:
   During a dialog there's always an active Dialog-state.
   A dialog-line's starting dialog state must be the same as the active dialog state, for the line to be a possible candidate.
   If the dialog is started by meeting a party on the map, initially, the active dialog state is "start"
   If the dialog is started by speaking to an NPC in a town, initially, the active dialog state is "start"
   If the dialog is started by helping a party defeat another party, initially, the active dialog state is "party_relieved"
   If the dialog is started by liberating a prisoner, initially, the active dialog state is "prisoner_liberated"
   If the dialog is started by defeating a party led by a hero, initially, the active dialog state is "enemy_defeated"
   If the dialog is started by a trigger, initially, the active dialog state is "event_triggered"
3) Conditions block (list): This must be a valid operation block. See header_operations.py for reference. = []
4) Dialog Text (string):
5) Ending dialog-state:
   If a dialog line is picked, the active dialog-state will become the picked line's ending dialog-state.
6) Consequences block (list): This must be a valid operation block. See header_operations.py for reference. = []

Looking at Ramun's opening statement:

1)Dialogue partner : trp_ramun_the_slave_trader
2)Starting Dialog-State : "start"
3)Conditions block : (troop_slot_eq, "$g_talk_troop", slot_troop_met_previously, 0),
4) Dialog text string : "Good day to you, {young man/lassie}."
5) Ending dialog-state : "ramun_introduce_1"
6) Consequences : []

The most important things to note in this tuple are the *dialog-states,* both the starting and ending dialog-states. We will delve into these more deeply now.

The *ending dialog-state* ("ramun_introduce_1") is what leads a conversation from one line to the next. The ending dialog-state can be anything you like -- but there must be another tuple with a matching *starting dialog-state*. For example, if we were to make a tuple with the ending state "blue_oyster", this would lead into any tuple with the starting state "blue_oyster". There must be an exact match; if no match exists, **build_module.bat** will throw an error upon trying to build.

If there are multiple tuples with the starting state "blue_oyster", something special happens. If the tuples are spoken by the player, they result in a menu where the player can choose between the provided tuples. If spoken by an NPC, the module system will use the first tuple in **module_dialogs.py** for which all conditions are met -- even if there are multiple lines that qualify.

To end a conversation, you must use the ending dialog-state "close_window".

To start a conversation, there are several special starting dialog-states from which you can choose. We will refer to these as *initial dialog-states*. Here is the full list of initial dialog-states:

"start" -- Considered when speaking to an NPC in a scene or when a conversation is triggered inside a scene.
"party_encounter" -- Considered when encountering another party on the overland map or in a scene.
"party_relieved" -- Considered when assisting a battling party on the overland map, once the player has won the fight.
"prisoner_liberated" -- Considered when the player defeats an enemy party with one or more Hero prisoners.
"enemy_defeated" -- Considered when the player defeats an enemy party led by a Hero.

"event_triggered" -- Considered when a dialogue is triggered by an operation while not in a scene.

As you can see, each initial dialog-state is designed for a specific situation. It will not be considered anywhere outside its situation.

Since our example tuple is initiated by speaking to Constable Hareck in Mod Town, its initial dialog-state is "start".

9.2 -- Requirements And The Conditions Block

The dialogue interface is very flexible, and can be used in a great number of ways. It handles both events on the overland map and events in scenes. It allows dialogues to be triggered whenever you want them.

In the following segments, we examine how to use the dialogue interface to its fullest, and after that we will learn how to create complex dialogues of our own.

As we have outlined in the last segment, a dialogue line will only be considered if all its requirements are met. First of all, the player must be speaking to the correct troop. A line with trp_ramun_the_slave_trader will not be considered if the player is addressing trp_constable_hareck. The constant anyone may be used if the line is to be spoken by anyone the player is addressing at the time.

Second, the starting dialog-state must be in accordance with the situation -- either the dialogue line is initiated by an initial dialog-state, or the line follows from a matching ending dialog-state in another line. If the starting dialog-state does not meet specifications, it won't be considered for use.

Thirdly, the same logic of the previous two points applies to the *conditions block*. Unless all of a line's conditions are met, the line will not be considered. If either the conditions or the starting dialog-state are erroneous, you will experience problems; either **build_module.bat** will throw an error, or the erroneous dialogue lines will simply freeze in-game, since their ending dialog-states will not be able to find another tuple to activate.

This is the reason why you must be careful with conditions blocks. Make sure you don't break your own dialogue by planting conditions which are not properly set.

However, when everything is working in harmony, conditions blocks can be very powerful. They can contain try blocks. You can call slots from inside a conditions block, and then use the result in a condition operation inside the same block. You can set registers and string registers in order to use them in the actual dialogue. We will do all of these things in this Part of the documentation, one at a time.

You can observe the use of a conditions block in the tuple of module_dialogs use to talk with Ramun the slave trader:

```
  [trp_ramun_the_slave_trader, "start", [(troop_slot_eq, "$g_talk_troop", slot_troop_met_previously, 0),],
```

This block contains only one condition, which requires the variable slot_troop_met_previously of "$g_talk_troop" to be equal to 0. This is because all registers and variables are equal to 0 at the beginning of a new game. And if you look at the next tuple, you will notice the consequences block:

```
  [trp_ramun_the_slave_trader|plyr, "ramun_introduce_1", [], "Forgive me, you look like a trader, but I see none of your merchandise.",
"ramun_introduce_2",[
   (troop_set_slot, "$g_talk_troop", slot_troop_met_previously, 1),]],
```

The **troop_set_slot** operation in this block sets the variable slot_troop_met_previously of "$g_talk_troop" (to whom you are talking) to 1 after the line has been displayed. In other words, after this line has been displayed once, it will never be displayed again -- because afterwards, the variable slot_troop_met_previously will no longer be equal to 0. The dialogue system will then ignore this line and instead go to the next tuple in the file which meets requirements:

```
  [trp_ramun_the_slave_trader,"start", [], "Hello, {playername}.", "ramun_talk",[]],
```

The only requirements on this line are that the player must be speaking to Ramun in a scene. It will always be selected when speaking to Ramun after he has delivered his speech.

It is important to know that various variables are set in the first 3 tuples of this file. They are set when you talk with… **"anyone"**. Each has its specific dialog start state, but only covers conversations between you and someone you are specifically targeting for a conversation. "$g_talk_troop" is just one of the variables cataloged and set in this process. In the first few lines of the first tuple, they also store your relation with "$g_talk_troop" from reg0, which is calculated in the script **troop_get_player_relation** (check it out in **module_scripts.py**) and stored in "$g_talk_troop_relation":

```
  [anyone ,"start", [(store_conversation_troop, "$g_talk_troop"),
                    (store_conversation_agent, "$g_talk_agent"),
                   (store_troop_faction, "$g_talk_troop_faction", "$g_talk_troop"),
#                     (troop_get_slot, "$g_talk_troop_relation", "$g_talk_troop", slot_troop_player_relation),
                   (call_script, "script_troop_get_player_relation", "$g_talk_troop"),
                   (assign, "$g_talk_troop_relation", reg0),
```

Things are done this way so you won't have to do this kind of thing every time you create a new dialog block. If you will be having special dialogs that will take into consideration things like your relation with the target, then it would be good to scan through these (or the specific start-dialog state) and see if what you will be checking is already set.

9.3 -- Adding New Dialogue

Here we finally have a chance to give our new troop Geoffrey some unique dialogue. However, like module_troops, module_dialogs is another file where you cannot simply add more tuples at the bottom of the list. The end of module_dialogs's Python list contains placeholder conversations that will trigger for anyone at all, and because the file is scanned from top to bottom for a match, any dialogue that you add after the placeholder conversations will be completely ignored.

It is recommended that you add new dialogue before the following comment:

```
### COMPANIONS
```

Our first goal is to create an introduction for Geoffrey. This will allow us to gain a little bit of experience before we throw ourselves into the creation of a full-blown quest. Copy the following tuple and paste it just above the ### COMPANIONS comment:

```
  ### JIKs added Dialogs
  [trp_npc17,"start", [], "What? What do you want? Leave me be, peasant, I have no time for beggars.", "geoffrey_talk",[]],
```

First note that I have also included a comment defining this as the start of the comments I will be adding. In places that you will be adding large blocks of code, it's always a good idea to comment them. This is the first tuple in our new conversation. It's activated by the initial dialog-state "start", spoken by Geoffrey, and it leads into the starting dialog-state "geoffrey_talk".

Next we will make a follow-up line, spoken by the player. Copy the following tuple and paste it into module_dialogs, just below the new tuple:

```
  [trp_npc17|plyr,"geoffrey_talk", [], "Nothing, never mind.", "close_window",[]],
```

This line will be displayed right after the first. It is spoken by the player. Due to the ending dialog-state "close_window", this conversation will end after the line has been displayed.

As we have covered before, adding multiple tuples with the same starting dialog-state will result in an option menu from which the player can choose the preferred line. Again, remember that this only works for tuples that are spoken by the player (like [trp_npc17|plyr,...), and not for any other troop (just [trp_npc17,...). Copy the following tuples and paste them into your module_dialogs file:

```
  [trp_npc17|plyr,"geoffrey_talk", [], "And who are you supposed to be?", "geoffrey_talk_2",[]],
  [trp_npc17,"geoffrey_talk_2", [], "Why, I'm Geoffrey Eaglescourt, son of the Baron Eaglescourt! Leader of the Red Riders, bane of bandits,\
   and crusher of pirates!", "geoffrey_talk_3",[]],
  [trp_npc17|plyr,"geoffrey_talk_3", [], "Oh, I see. And how many pirates have you killed?", "geoffrey_talk_4",[]],
  [trp_npc17,"geoffrey_talk_4", [], "See for yourself! I scalp every one of the dogs I kill. They are my battle trophies.", "geoffrey_talk_5",[]],
  [trp_npc17|plyr,"geoffrey_talk_5", [], "That's nice. I'll be going now.", "close_window",[]],
```

This is all part of one possible conversation path -- a small back-and-forth between Geoffrey and the player. Nothing important is happening yet, but we're about to add a little more variety to this exchange. Add the following two tuples to the dialogue:

```
  [trp_npc17|plyr,"geoffrey_talk", [], "Nothing, never mind.", "close_window",[]],
  [trp_npc17|plyr,"geoffrey_talk", [(check_quest_active,"qst_mod_trouble"),(quest_slot_eq,"qst_mod_trouble",slot_quest_current_state,0)],
   "You look familiar.   Haven't I seen your face in a pigsty before?", "geoffrey_hostile",[]],

  [trp_npc17|plyr,"geoffrey_talk", [], "And who are you supposed to be?", "geoffrey_talk_2",[]],
  [trp_npc17,"geoffrey_talk_2", [], "Why, I'm Geoffrey Eaglescourt, son of the Baron Eaglescourt! Leader of the Red Riders, bane of bandits,\
   and crusher of pirates!", "geoffrey_talk_3",[]],
  [trp_npc17|plyr,"geoffrey_talk_3", [], "Oh, I see. And how many pirates have you killed?", "geoffrey_talk_4",[]],
  [trp_npc17,"geoffrey_talk_4", [], "See for yourself! I scalp every one of the dogs I kill. They are my battle trophies.", "geoffrey_talk_5",[]],
  [trp_npc17|plyr,"geoffrey_talk_5", [], "That's nice. I'll be going now.", "close_window",[]],
  [trp_npc17|plyr,"geoffrey_talk_5", [(check_quest_active,"qst_mod_trouble"),(quest_slot_eq,"qst_mod_trouble",slot_quest_current_state,0)], "Really?\
   Those scalps look suspiciously like horse tails to me.", "geoffrey_hostile",[]],
```

These two lines will create extra menu choices for the starting dialog-states "geoffrey_talk" and "geoffrey_talk_5", respectively. It doesn't technically matter where in module_dialogs you place tuples with the same starting dialog-state. The dialogue system will find them all and, if conditions are met, add them to the menu options. However, you should try to keep menu tuples together in order to keep your code straightforward and readable.

The most notable feature of these lines is their condition blocks. Either line will only ever appear as a dialogue option if the quest "qst_mod_trouble" is currently active and if the quest slot "slot_quest_current_state" is equal to 0. If either of these conditions is not met, the line won't be considered for display. Now, copy and paste these last few tuples:

```
  [trp_npc17,"geoffrey_hostile", [], "What?! I'll see you dead for that insult, peasant! Don't you know who I am?", "geoffrey_hostile_2",[]],
  [trp_npc17|plyr,"geoffrey_hostile_2", [], "No . . . Was I supposed to remember?", "geoffrey_hostile_3",[]],
  [trp_npc17,"geoffrey_hostile_3", [], "Why, I'm Geoffrey Eaglescourt, son of the Baron Eaglescourt, and you have delivered the gravest\
   insult to my family's honour! I demand satisfaction! Meet me outside the town walls at noon, or you will be known a coward to every\
   man in this countryside. Good day, {knave/wench}.", "geoffrey_hostile_4",[]],
  [trp_npc17|plyr,"geoffrey_hostile_4", [], "Charming lad. Noon, eh? I shouldn't miss it...", "close_window",
   [(quest_set_slot,"qst_mod_trouble",slot_quest_current_state,1)]],
```

The dialogue system will also display words you can base on gender. In the line "geoffrey_hostile_3", the dialog display the word 'knave' or 'wench' depending on the player's gender. With that done, we now have a conversation with several quest-related conditions and consequences, as well as a ready-made quest tuple; but we have yet to create the conditions that will allow the player to activate the quest in the first place. This we will do in the next segment. If you want, you can run **build_module.bat** and visit Geoffrey, see the basic conversation take place.

9.4 -- Dialogue And Quests

When adding new lines to an existing conversation, we must be very careful that our new tuples play nicely with the existing ones. Remember that the dialogue file is scanned from top to bottom; remember to check your conditions blocks; remember to keep a close eye on your syntax. One misspelling can throw off entire blocks of code. Build your module often so that you can catch any syntax errors early.

First, we're going to add one more tuple for Geoffrey, and we're going to add it before any of his other lines. This means it will be considered before his other lines. If this tuple's conditions are present, it will be selected for use regardless of any others that might have also met conditions.
Currently, Geoffrey's first tuple is this:

```
  [trp_npc17,"start", [], "What? What do you want? Leave me be, pesant, I have no time for beggars.", "geoffrey_talk",[]],
```

Now, copy the following tuple and paste it into your dialogs file above the aforementioned tuple:

```
  [trp_npc17,"start", [(quest_slot_eq,"qst_mod_trouble",slot_quest_current_state,1)],
   "Noon time, {knave/wench}.  I will deal with you then...", "close_window",[]],
```

With this tuple in place, Geoffrey's normal conversation will no longer be displayed once he has challenged you to a duel. This way, you can change conversations depending on their situation, and create infinite varieties of dialogue with conditions blocks to manage when each line appears.

Since the conversations from older versions of M&B are no longer available, we will have to put in a basic dialog that will appear when you talk with Constable Hareck. We'll add his dialog after Geoffrey's. These 2 lines will do nicely:

```
  [trp_hareck,"start", [], "Yes? Can I help you?", "hareck_talk",[]],

  [trp_hareck|plyr,"hareck_talk", [], "Nothing. Good-bye.", "close_window",[]],
```

"hareck_talk" is a menu dialog-state that can have several options, of which this tuple will be the last. In order for a dialogue option to appear in the menu above this one, we have to add it into module_dialogs above this tuple.

Copy the following tuple and paste it into your dialogs file between the two aforementioned tuples. We need to keep the first "start" dialog from Hareck at the top:

```
   [trp_hareck|plyr,"hareck_talk",[(neg|check_quest_active,"qst_mod_trouble"),(neg|check_quest_finished,"qst_mod_trouble")],
    "Is something wrong? You look worried.", "hareck_troublemakers",[]],
```

This tuple will only be displayed if the quest "qst_mod_trouble" is not active and not completed. This is due to the negation-prefix ***neg|***, which causes a condition operation to require the opposite of what it normally requires. For example, the condition operation ***eq*** requires two values to be equal; ***neg|eq*** requires the values to be not equal. Now, just below this new tuple, copy and paste the following tuples. We will be keeping the "Nothing, Good-bye" tuple at the very bottom of all of Hareck's dialog tuples:

```
  [trp_hareck,"hareck_troublemakers", [], "Oh, it's nothing, just . . .", "hareck_troublemakers_2",[]],
  [trp_hareck|plyr,"hareck_troublemakers_2", [], "You can tell me, sir.", "hareck_troublemakers_3",[]],
  [trp_hareck,"hareck_troublemakers_3", [], "No harm in it, I suppose. The trouble is, a few of the town's young nobles . . .\
   Spoiled dandies and fops, the lot of them . . . they've decided that suddenly they're men to be respected, and that they\
   should 'take matters into their own hands', to 'take action where the official government has failed'. They say they're going\
   to kill all the river pirates that have been troubling Zendar of late. Of course, they've not actually gone out to fight any river\
   pirates, but they've been making a great ruckus in town and there's not a thing I can do about it.", "hareck_troublemakers_4",[]],
  [trp_hareck|plyr,"hareck_troublemakers_4", [], "Hmm . . . Would there be a reward for solving this problem?", "hareck_troublemakers_5",[]],
  [trp_hareck,"hareck_troublemakers_5", [], "What? What are you saying?", "hareck_troublemakers_6",[]],
  [trp_hareck|plyr,"hareck_troublemakers_6", [], "Nothing, sir. However, it sounds to me like a neutral third party might be just\
   what you need. I could talk to them.", "hareck_troublemakers_7",[]],
  [trp_hareck,"hareck_troublemakers_7", [], "Heh. Well, you can try, friend. If you manage to do any good, I'll even throw in a\
   few coins for getting the sand out of my breeches. Their leader is a boy named Geoffrey, spends most of his time on\
   watered-down ale and whores. Chances are you'll find him up the stairs in the back.", "hareck_troublemakers_8",[]],
  [trp_hareck|plyr,"hareck_troublemakers_8", [], "Thank you, constable. I shall return.",  "close_window",
   [(setup_quest_text,"qst_mod_trouble"),(start_quest,"qst_mod_trouble"), (quest_set_slot,"qst_mod_trouble",slot_quest_current_state, 0)]],

  [trp_hareck|plyr,"hareck_talk",
   [(check_quest_active,"qst_mod_trouble"),(quest_slot_eq,"qst_mod_trouble",slot_quest_current_state,3)],
   "Constable, I've taken care of the toublemakers for you. They shouldn't be a worry any longer.", "hareck_troublemakers_10",[]],
  [trp_hareck,"hareck_troublemakers_10", [], "Truly? Thank God! A few more days and I would've thrown them all into a cell and thrown\
   away the key. Here, take this. You've earned it.","hareck_troublemakers_11",
   [(troop_add_gold,"trp_player",100),(add_xp_as_reward,750),(succeed_quest,"qst_mod_trouble")]],
  [trp_hareck|plyr,"hareck_troublemakers_11", [], "My pleasure, constable. If you've any other jobs that need doing, please let me know. Farewell.",
  "close_window",[]],

  [trp_hareck|plyr,"hareck_talk", [(check_quest_active,"qst_mod_trouble"),(quest_slot_eq,"qst_mod_trouble",slot_quest_current_state,4)],
   "Constable, I failed. I'm sorry.", "hareck_troublemakers_15",[]],
  [trp_hareck,"hareck_troublemakers_15", [], "Oh . . . Oh well. I suppose you did the best you could. Thanks anyway, friend. Perhaps some other\
   job will suit you better. I shall let you know when I have any. Farewell.", "close_window",[(fail_quest,"qst_mod_trouble")]],
```

The first block sets up the quest, explaining the details and how to start it. The second block finishes the quest with a nice little reward of gold and experience points, once the quest lost "slot_quest_current_state" has been set to 3, which we will do upon defeating Geoffrey. The third block finishes the quest if the player fails to defeat Geoffrey (setting"slot_quest_current_state" to 4), which means he will receive no rewards. Not that we still check that the quest is active. This is a good fall back incase you don't clean up variables. In this case, even after the quest ends it's slot_quest_current_state will still be either 3 or 4, which would mean the end dialogs would continue to be used since the test would be true.

One last thing we will add. We want Geoffrey to act differently when he his confronted for the duel. For this to work, we would need an indicator that he is in the field and ready to duel. We will use when "slot_quest_current_state" is set to 2 to let us know when to use the following dialogs, which we will put at the top of Geoffrey's dialog group. This will be set later on:

```
  [trp_npc17,"start", [(quest_slot_eq,"qst_mod_trouble",slot_quest_current_state,2)],
   "You have come to meet your doom, {knave/wench}...","geoffrey_duelling",[]],
  [trp_npc17|plyr,"geoffrey_duelling", [], "Let's get this over with...",
   "close_window",[(quest_set_slot,"qst_mod_trouble",slot_quest_current_state,4)]],
```

We also assign "slot_quest_current_state" to 4. You may remember that 4 is the fail state for the quest. This is to ensure that once you start the fight with Geoffrey, you either win (and change the state to 3 which is success), or you fail. As you can see, it can take some doing to cover all possible aspects of a quest. Ours is only half-finished. Most the dialogue is now in place, however, so we are now ready to move on to the next part of this documentation, where we will handle the triggers and variable changes.

PART 10

Before we go on, we will make a quick end to our quest. What if we could intimidate Geoffrey into leaving town. This would satisfy Hareck, thus completing the quest. But we can't make it too easy. One thing we could do is make a dialog based on a player's attribute or skill to influence what can be said. Usually big strong men are good for intimidating others, so let's make our quick quest finish based on the player's STR score.

```
  [trp_npc17|plyr,"geoffrey_hostile_4",[(store_attribute_level,":player_str","trp_player",0),(ge,":player_str",14)],
   "[Intimidate with Strength]","geoffrey_flee1",[]],
  [trp_npc17|plyr,"geoffrey_hostile_4", [], "Charming lad. noon, eh? I shouldn't miss it...",
   close_window",[(quest_set_slot,"qst_mod_trouble",slot_quest_current_state,1)]],
```

With this added tuple assigned to the dialog start state of "geoffrey_hostile_4". The first thing we must do is reference the attribute or skill in question. To do so we run the operation **store_attribute_level**, and store the player's 0th attribute (STR) into a local variable ":player_str". We can then compare it to a specific value we set. The value of 14 is a good one to test with, but you can set it as high or as low as you want. You can easily get a starting STR of 14 or 15 by choosing the background choices : Veteran Warrior, Page at Noble's Court, Smith, Personal Revenge. You can also turn on cheat mode and level up.

Lastly to close out this quest, you can add this tuple as well at the very end of Geoffrey's dialogs:

```
  [trp_npc17,"geoffrey_flee2",[],"Bah...  This town is no fun anyway.  Tell your constable that I will leave within the hour.","close_window",
   [(quest_set_slot,"qst_mod_trouble",slot_quest_current_state,3),(remove_troop_from_site,"trp_npc17","scn_town_mod_tavern")]],
```

This will set the value of slot_quest_current_state to 3, which is the success state needed when talking with Hareck. What we have done here is give ourselves a quick way to test the end state of our little quest. We also have Geoffrey removed from the scene, though this will not take place until we leave and come back. As a test of what you have learned, you should be able to add one more "start" dialog for Geoffrey at the top to deal with if the player were to talk to Geoffrey again before leaving the scene. Make sure to test against the current state of the quest. Have Geoffrey say something like "I will be gone within the hour…".

Now that we have gained some significant experience working with the module system, we can turn our hands to one of the more complex components; Triggers and Scripts. Triggers are time-based operations blocks that activate at regular intervals or at specified occasions. Scripts can be run when certain things happen, and can be called from other module files (with the **call_script** operation). When either happens all operations in the operations block will then be executed.

10.1 – Party Encounters > Module_Scripts.py

When ever your party touches another part (as defined in modul_parties.py) the script "game_event_party_encounter" is executed. It is a fairly long tuple, but it is used to handle any encounter as well as setting up variables for use by other tuples that will branch from this one, using the "call_script" or "jump_to_menu" command.

```
  # script_game_event_party_encounter:
  # This script is called from the game engine whenever player party encounters another party or a battle on the world map
  # INPUT:
  # param1: encountered_party
  # param2: second encountered_party (if this was a battle
  ("game_event_party_encounter",
   [
       (store_script_param_1, "$g_encountered_party"),
       (store_script_param_2, "$g_encountered_party_2"),# encountered_party2 is set when we come across a battle or siege, otherwise it's a negative
value
#       (store_encountered_party, "$g_encountered_party"),
#       (store_encountered_party2,"$g_encountered_party_2"), # encountered_party2 is set when we come across a battle or siege, otherwise it's a
minus value
       (store_faction_of_party, "$g_encountered_party_faction","$g_encountered_party"),
       (store_relation, "$g_encountered_party_relation", "$g_encountered_party_faction", "fac_player_faction"),
       (party_get_slot, "$g_encountered_party_type", "$g_encountered_party", slot_party_type),
      (party_get_template_id,"$g_encountered_party_template","$g_encountered_party"),
#       (try_begin),
#         (gt, "$g_encountered_party_2", 0),
#         (store_faction_of_party, "$g_encountered_party_2_faction","$g_encountered_party_2"),
#         (store_relation, "$g_encountered_party_2_relation", "$g_encountered_party_2_faction", "fac_player_faction"),
#         (party_get_template_id,"$g_encountered_party_2_template","$g_encountered_party_2"),
#       (else_try),
#         (assign, "$g_encountered_party_2_faction",-1),
#         (assign, "$g_encountered_party_2_relation", 0),
#         (assign,"$g_encountered_party_2_template", -1),
#       (try_end),

#NPC companion changes begin
      (call_script, "script_party_count_fit_regulars", "p_main_party"),
      (assign, "$playerparty_prebattle_regulars", reg0),

#        (try_begin),
#            (assign, "$player_party__regulars", 0),
#            (call_script, "script_party_count_fit_regulars", "p_main_party"),
#            (gt, reg0, 0),
#            (assign, "$player_party_contains_regulars", 1),
#        (try_end),
#NPC companion changes end


      (assign, "$g_last_rest_center", -1),
      (assign, "$talk_context", 0),
      (assign,"$g_player_surrenders",0),
      (assign,"$g_enemy_surrenders",0),
      (assign, "$g_leave_encounter",0),
      (assign, "$g_engaged_enemy", 0),
#       (assign,"$waiting_for_arena_fight_result", 0),
#       (assign,"$arena_bet_amount",0),
#       (assign,"$g_player_raiding_village",0),
      (try_begin),
        (neg|is_between, "$g_encountered_party", centers_begin, centers_end),
        (rest_for_hours, 0), #stop waiting
      (try_end),
#       (assign, "$g_permitted_to_center",0),
      (assign, "$new_encounter", 1), #check this in the menu.
      (try_begin),
        (lt, "$g_encountered_party_2",0), #Normal encounter. Not battle or siege.
        (try_begin),
          (party_slot_eq, "$g_encountered_party", slot_party_type, spt_town),
          (jump_to_menu, "mnu_castle_outside"),
        (else_try),
          (party_slot_eq, "$g_encountered_party", slot_party_type, spt_castle),
          (jump_to_menu, "mnu_castle_outside"),
        (else_try),
          (party_slot_eq, "$g_encountered_party", slot_party_type, spt_ship),
          (jump_to_menu, "mnu_ship_reembark"),
        (else_try),
          (party_slot_eq, "$g_encountered_party", slot_party_type, spt_village),
          (jump_to_menu, "mnu_village"),
        (else_try),
          (party_slot_eq, "$g_encountered_party", slot_party_type, spt_cattle_herd),
          (jump_to_menu, "mnu_cattle_herd"),
        (else_try),
          (is_between, "$g_encountered_party", training_grounds_begin, training_grounds_end),
          (jump_to_menu, "mnu_training_ground"),
        (else_try),
          (eq, "$g_encountered_party", "p_zendar"),
          (jump_to_menu, "mnu_zendar"),
        (else_try),
          (eq, "$g_encountered_party", "p_salt_mine"),
          (jump_to_menu, "mnu_salt_mine"),
        (else_try),
          (eq, "$g_encountered_party", "p_four_ways_inn"),
          (jump_to_menu, "mnu_four_ways_inn"),
        (else_try),
          (eq, "$g_encountered_party", "p_test_scene"),
          (jump_to_menu, "mnu_test_scene"),
        (else_try),
          (eq, "$g_encountered_party", "p_battlefields"),
          (jump_to_menu, "mnu_battlefields"),
        (else_try),
          (eq, "$g_encountered_party", "p_training_ground"),
          (jump_to_menu, "mnu_tutorial"),
        (else_try),
          (eq, "$g_encountered_party", "p_camp_bandits"),
          (jump_to_menu, "mnu_camp"),
        (else_try),
          (jump_to_menu, "mnu_simple_encounter"),
        (try_end),
      (else_try), #Battle or siege
        (try_begin),
          (this_or_next|party_slot_eq, "$g_encountered_party", slot_party_type, spt_town),
          (party_slot_eq, "$g_encountered_party", slot_party_type, spt_castle),
          (try_begin),
            (eq, "$auto_enter_town", "$g_encountered_party"),
            (jump_to_menu, "mnu_town"),
          (else_try),
            (eq, "$auto_besiege_town", "$g_encountered_party"),
            (jump_to_menu, "mnu_besiegers_camp_with_allies"),
          (else_try),
            (jump_to_menu, "mnu_join_siege_outside"),
          (try_end),
        (else_try),
          (jump_to_menu, "mnu_pre_join"),
        (try_end),
      (try_end),
      (assign,"$auto_enter_town",0),
      (assign,"$auto_besiege_town",0),
     ]),
```

This is the first and only trigger in the list. It is a very simple construction, containing only two separate fields.

Breakdown of the tuple fields:

1) Script id: The prefix "script_" will be inserted when referencing scripts.
2) Operation block: This must be a valid operation block. See header_operations.py for reference.

As we can see by the commented description, this trigger is called by the game engine whenever the player encounters a party on the overland map. We will now individually highlight various sections of the tuple to examine what each does.

Section 1a:

```
  (store_script_param_1, "$g_encountered_party"),
  (store_script_param_2, "$g_encountered_party_2"),# encountered_party2 is set when we come across a battle or siege, otherwise it's a negative
```

"store_script_param_#" are variables passed by the command starting the tuple. As the commenting shows, the first parameter is the first party you encounter, and is stored in "$g_encountered_party". The second parameter is only set if you encounter a battle already in progress, and would be stored in "$g_encountered_party_2". With these set, we can run any operation requiring a party using these variables.

Section 1b:

```
   (store_faction_of_party, "$g_encountered_party_faction","$g_encountered_party"),
   (store_relation, "$g_encountered_party_relation", "$g_encountered_party_faction", "fac_player_faction"),
   (party_get_slot, "$g_encountered_party_type", "$g_encountered_party", slot_party_type),
   (party_get_template_id,"$g_encountered_party_template","$g_encountered_party"),
```

As we can see here, we are putting the newly assigned variable to good use. From it the operations **store_faction_of_party** can be run. From this operation we now haw the encountered party's faction info, "$g_encountered_party_faction", and from that we can then run **store_relation** and compare it with "fac_player_faction", the player's faction. These operations can be found in **header_operations.py**.

Section 2:

```
#NPC companion changes begin
   (call_script, "script_party_count_fit_regulars", "p_main_party"),
   (assign, "$playerparty_prebattle_regulars", reg0),
```

With this section I will take a little side step. Looking at the variables (starting with the "$"), we can determine that the script is attempting to assign NPCs that are suitable for fighting. Jumping a bit ahead, we can see how this is done by examining the script "party_count_fit_regulars":

```
  #script_party_count_fit_regulars:
  # Returns the number of unwounded regular companions in a party
  # INPUT:
  # param1: Party-id

  ("party_count_fit_regulars",
    [
      (store_script_param_1, ":party"), #Party_id
      (party_get_num_companion_stacks, ":num_stacks",":party"),
      (assign, reg0, 0),
      (try_for_range, ":i_stack", 0, ":num_stacks"),
        (party_stack_get_troop_id,      ":stack_troop",":party",":i_stack"),
        (neg|troop_is_hero, ":stack_troop"),
        (party_stack_get_size,          ":stack_size",":party",":i_stack"),
        (party_stack_get_num_wounded, ":num_wounded",":party",":i_stack"),
        (val_sub, ":stack_size", ":num_wounded"),
        (val_add, reg0, ":stack_size"),
      (try_end),
  ]),
```

A short tuple, this does a few things that are good to take note of. First off note that the store_script_param_1 is stored in a variable defined as ":party", not "$party". The reason this is done this way is that ":party" is only needed in this tuple. This is called a local variable. It is created when this tuple starts, and is deleted when this tuple ends. If it were defined as "$party", a global variable, it would be usable anywhere. It would also retain it's value until you assign it a different one.

Next we will talk quick look at the **try_for_range** code block. Those familiar with programming will know this as a for…loop. It and all other lhs_operations can be found in the header_operations.py, but unfortunately their uses are not defined here. So for the sake of those not familiar with a for…loop, I will break it down for you:

try_for_range = the call for the for loop
"i_stack" = the variable that will be incrementing. *If you want a decreasing count, use **try_for_range_backwards***
0 = the starting value of i_stack
":num_stacks" = when the incrementing variable (:i_stack) reaches this number, the loop stops
try_end = the end of the code block for the loop

Between the try_for_range and try_end is the code that will repeatedly be executed until the loop runs out. Let's get back on track.

Section 3:

I will skip pass the assign code, as it is there just to reset the variables, and go to the first **try_begin…try_end**:

```
      (try_begin),
        (neg|is_between, "$g_encountered_party", centers_begin, centers_end),
        (rest_for_hours, 0), #stop waiting
      (try_end),
```

The interesting thing to note about this section is the operation **(try_begin)**. This operation opens a *try operation*, which functions just like a normal operations block, except for one difference. If a try operation contains a condition whose requirements are not met, this doesn't cancel the entire rest of the operations block; it only cancels up to the nearest **(try_end)**. Everything after the (try_end) proceeds as normal. In essence, a try operation functions like an isolated operations block inside an operations block. You may recognize this as an **if…end if** statement.

There is also the additional try operation **(else_try)**, which can be inserted into an active try operation such as the one in **module_simple_triggers**. The contents of an **else_try** block will only be considered for execution if all active try operations before the **(else_try)** (including other **else_try** blocks) have failed to meet their conditions as seen a few lines down in the next section we will look at.

Section 3:

```
      (try_begin),
        (lt, "$g_encountered_party_2",0), #Normal encounter. Not battle or siege.
        (try_begin),
          (party_slot_eq, "$g_encountered_party", slot_party_type, spt_town),
          (jump_to_menu, "mnu_castle_outside"),
        (else_try),
          (party_slot_eq, "$g_encountered_party", slot_party_type, spt_castle),
          (jump_to_menu, "mnu_castle_outside"),
        (else_try),
        .
        .
        .
```

Again, good commenting lets us know that because there is no 2nd party, this is not a battle or a siege in progress that you have come upon. Then we test to see what party type we have encountered by testing **party_slot_eq, "$g_encountered_party", slot_party_type, spt_town**. If "$g_encountered_party" is not equal to spt_town, then we move on to the next **else_try**.

Side Note: If we were to scroll up (search for # *Towns (loop)*), we would find the loop where spt_town is set for each town. There is a lot more that is done here such as setting up the various scenes in the town, but the important thing to note is that it is set (in the first line) for parties between towns_begin and towns_end (both markers are defined in module_constants.py)s. This code block also shows us that the scenes we make for our towns must also be in the same order as the towns themselves!

This continues until one of the conditions is met, then the code between the condition and the next else_try (or try_end if it's the last condition) will be executed. This goes on for quite a bit as you will see there are various non-battle encounters you can have. You can also see that this check can be done 2 other ways:

```
        (is_between, "$g_encountered_party", training_grounds_begin, training_grounds_end),
```

Which tests a range defined in module_constants.py and:

```
        (eq, "$g_encountered_party", "p_four_ways_inn"),
```

Which is a direct reference to a party defined in **module_parties.py**. The latter is a more specific definition and should not be used except in special cases. If you want to use the default menus for towns, castles, villiages, and other party types that are abundant, you should have them set within the ranges defined in **module_constants.py**, and let them be handled like the rest.

Section 4:

```
      .
      .
      .
      (else_try),
        (jump_to_menu, "mnu_simple_encounter"),
      (try_end),
```

Coming to the end of the non-battle/non-siege encounter, we have the final "did not find a match with any conditions" outcome. Jump_to_menu, "mnu_simple_encounter". With a complex and long list of try blocks, it's a good idea to end it with a defaulting value. That way if non of the conditions are met, you can be sure that something will happen, even if it's to broadcast a test "error" message. We end it off with the try_end. But this isn't the overall end. Remember, this started with a check for a 2nd party. We have now exhausted the "no 2nd party" outcome. The code must now deal with the other side of that outcome, meaning you came across a battle or a siege in progress.

Section 5:

```
    (else_try), #Battle or siege
      (try_begin),
        (this_or_next|party_slot_eq, "$g_encountered_party", slot_party_type, spt_town),
        (party_slot_eq, "$g_encountered_party", slot_party_type, spt_castle),
```

I stopped here because we have another form of condition that may confuse some. After the try_begin declaration, the first conditions starts with this_or_next|. Though header_operations.py doesn't document this well, I would interpret it as: "if this condition is true, or the next condition is true". The first test being party_slot_eq, "$g_encountered_party", slot_party_type, spt_town, and the second test being party_slot_eq, "$g_encountered_party", slot_party_type, spt_castle. If either is true, the code block below it will run. For it to fail, both conditions must fail. In other languages this would be seen as:

If (<condition_1> or <condition_2>) then…
If you are familiar with this and are asking where is the "this_and_next condition?", those are more easily done, as you just need to put one condition after the other. As the conditions are worked out, if one fails, the try_begin block stops. We have seen this before in the dialogs, though I didn't stop to point it out then. Look here:

```
  [trp_hareck|plyr,"hareck_talk", [(check_quest_active,"qst_mod_trouble"),(eq,"$geoffrey_duel",3)], "Constable, I've taken care of the toublemakers for you. They shouldn't be a worry any longer.", "hareck_troublemakers_10",[]],
```

The highlighted conditions must both be true (check_quest_active and $geoffrey_duel is equal to 3) making this an IF (<condition1> AND <condition2>).

We will skip the next few condition checks and go to the last 2 lines:

```
    (assign,"$auto_enter_town",0),
    (assign,"$auto_besiege_town",0),
```

It is important to reset variables that might be changed during the course of your tuples. You also want to clean up variables that may be used in other code blocks. If there is a chance that the initial value can change, there is also the chance that the next time you run this tuple, the value won't. It may then be holding on to a value set 1 or more cycles ago. $auto_enter_town is used just a few lines up, and the next time it is used, we want to make sure it has a fresh value. These are things to keep in mind when using $variables that are used in the native code.

10.2 -- Editing the Party Encounters Trigger – SKIP TO 9.3 FOR NOW. NEED CLARIFICATION

In order to make a town work, we have to make sure that any encounter with this town directs the player to the proper game menu. This leaves us with two choices; either we can direct our new party to use one of the Native menus, such as **mnu_town**, or we can create a new one.

For our new_town, we will create a custom menu, though if you have travelled there, you will know that it's much easier to use the defaults.

For what we want to do, we will focus on Section 3, where the check for spt_town is located:

```
    (try_begin),
      (lt, "$g_encountered_party_2",0), #Normal encounter. Not battle or siege.
      (try_begin),
        (party_slot_eq, "$g_encountered_party", slot_party_type, spt_town),
        (jump_to_menu, "mnu_castle_outside"),
      (else_try),
        (party_slot_eq, "$g_encountered_party", slot_party_type, spt_castle),
        (jump_to_menu, "mnu_castle_outside"),
      (else_try),
      .
      .
      .
```

The highlighted part is where the check is made. If we wish to add a new menu for our town, there is 2 ways to do it. If it's a 1 shot deal, then we can just add another try that tests for the specific ID of your town. To make things interesting, let's assume that this is a new structure, such as a dungeon, and it will be one of many. Though we may want to keep the other aspects of the town (making/defining scenes), we want a specific menu to pop up for this type of party. We can go into **module_constants.py** and (near where the towns, castles, and villages are set, we can set our own like this:

```
towns_begin = "p_town_1"
castles_begin = "p_castle_1"
villages_begin = "p_village_1"
modtown_begin = "p_mod_town"

towns_end = castles_begin
castles_end = villages_begin
villages_end   = "p_salt_mine"
modtown_end = castles_begin
```

Now we have a range that is still within the town range, but we can be more specific and target p_mod_town, and any others between it and p_castle_1. Just be careful when defining constants like this. You want to be sure that the names you pick are not already used. Going back to module_scripts.py, and make these changes:

```
    (try_begin),
      (lt, "$g_encountered_party_2",0), #Normal encounter. Not battle or siege.
      (try_begin),
        (party_slot_eq, "$g_encountered_party", slot_party_type, spt_town),
        (try_begin),
          (is_between, "$g_encountered_party", modtown_begin, modtown_end),
          (jump_to_menu,"mnu_modtown_menu"),
        (else_try),
           (jump_to_menu, "mnu_castle_outside"),
        (try_end),
        (party_slot_eq, "$g_encountered_party", slot_party_type, spt_castle),
        (jump_to_menu, "mnu_castle_outside"),
      (else_try),
      .
      .
      .
```

This highlighted section will, after checking that the "$g_encountered_party" is of **slot_party_type** spt_town, then check if "$g_encountered_party" is a part of the grouping midtown_begin to midtown_end. If it is, then it will **jump_to_menu** "mnu_modtown_menu". If not, it will use default menu of "mnu_castle_outside". If we were to try and run build_module.bat, it would fail, as we have not yet created the menu "mnu_modtown_menu".

Now the only thing left to do to make our town operational is creating a menu for it. Before we move on to module_game_menus, however, there is a lot more functionality waiting to be explored in triggers. Open **module_triggers.py** now and move on the next segment.

10.3 -- Breakdown, Module_Triggers

There are two types of triggers in the M&B module system: simple triggers and expanded triggers. Simple triggers are stored in **module_simple_triggers.py**, and expanded triggers are contained in **module_triggers.py**. They work via the same theory, but expanded triggers have several more options Lets check out **module_triggers.py**.

```
# Tutorial:
  (0.1, 0, ti_once, [(map_free,0)], [(dialog_box,"str_tutorial_map1")]),
```

The first trigger in this file is nice and simple. It is the trigger that pops up the map tutorial when the player first spawns on the overland map. Breakdown of the tuple fields:

1) Check interval: How frequently this trigger will be checked

2) Delay interval: Time to wait before applying the consequences of the trigger, after its conditions have been evaluated as true.
3) Re-arm interval. How much time must pass after applying the consequences of the trigger for the trigger to become active again.
   You can put the constant ti_once here to make sure that the trigger never becomes active again after it fires once.
4) Conditions block (list). This must be a valid operation block. See header_operations.py for reference.
   Every time the trigger is checked, the conditions block will be executed.
   If the conditions block returns true, the consequences block will be executed.
   If the conditions block is empty, it is assumed that it always evaluates to true.
5) Consequences block (list). This must be a valid operation block. See header_operations.py for reference.

Tutorial tuple examination:

1) Check interval = 0.1
2) Delay interval = 0
3) Re-arm interval = ti_once
4) Conditions block = (map_free,0)
5) Consequences block = (dialog_box,"str_tutorial_map1")

As we can see from the examination, this trigger is checked every 0.1 hours of game time; it has no delay interval; it never rearms due to *ti_once*. This means that, if the conditions in the trigger's conditions block are all met, the trigger fires once and never again. In this case, the trigger will fire if the player is placed anywhere on the map.

We can now begin to make a trigger of our own. Copy the tutorial trigger and paste it to the bottom of the Python list.

What we are going to do with this new trigger is spawn another party on the map; namely, Geoffrey's party. Replace (map_free,0) in the trigger's conditions block with the following operations:

```
(check_quest_active,"qst_mod_trouble"),
(quest_slot_eq,"qst_mod_trouble",slot_quest_current_state,1),
(store_time_of_day,reg(100)),
(gt,reg(100),12)
```

If you remember what we did in Part 8 of this doscumentation, we assigned the variable "slot_quest_current_state" to 1 after you talk Geoffrey into challenging you to a duel. Therefore, (quest_slot_eq,"qst_mod_trouble",slot_quest_current_state,1) checks whether or not Geoffrey has challenged you. If he hasn't, the conditions block will fail.

The following two operations store the time of day to reg(100), and then check if reg(100) is greater than 12; essentially, whether the time of day is later than 12:00. If it is, the conditions block will succeed and allow the trigger to fire.

Now, replace (dialog_box,"str_tutorial_map1") in the consequences block with the following operations:

```
(set_spawn_radius,0),
(spawn_around_party,"p_mod_town","pt_new_template"),
(assign,"$geoffrey_party_id",reg(0)),
(party_set_flags,"$geoffrey_party_id",pf_default_behavior,0),
(party_set_ai_behavior, "$geoffrey_party_id", ai_bhvr_attack_party),
(party_set_ai_object,"$geoffrey_party_id","p_main_party"),
```

The first operation sets the spawn radius; you must do this every time you want to spawn a party, or the party will be spawned at the most recently set radius, which can be very unpredictable.

The second operation spawns a party template ("pt_new_template") around a party ("p_mod_town") - the town we created, and the town in which you meet Geoffrey. Because of the way the operation is defined in **header_operations.py**, the identifier of the spawned party is automatically assigned to reg(0) when the party is spawned. We then assign this identifier to a variable so that it will not be lost when reg(0) is next overwritten.

The last 2 lines set the party's AI behavior and object. These operations are fairly straightforward. We are telling "$geoffrey_party_id" to attack "p_main_party". When you set a party to attack another party, it will relentlessly hunt down the other party and then attack it, regardless of faction and other considerations.

Let's add 2 more lines. When done, your Geoffrey duel tuple should look like this:

```
##JIKs Geoffrey duel quest trigger
        (0.1, 0, ti_once,
                [
                 (check_quest_active,"qst_mod_trouble"),
                 (quest_slot_eq,"qst_mod_trouble",slot_quest_current_state,1),
                 (store_time_of_day,reg(100)),
                 (gt,reg(100),12)
                ],
                [
                 (set_spawn_radius,0),
                 (spawn_around_party,"p_mod_town","pt_new_template"),
                 (assign,"$geoffrey_party_id",reg(0)),
                 (party_set_flags,"$geoffrey_party_id",pf_default_behavior,0),
                 (party_set_ai_behavior, "$geoffrey_party_id", ai_bhvr_attack_party),
                 (party_set_ai_object,"$geoffrey_party_id","p_main_party"),
                 (remove_troop_from_site,"trp_npc17","scn_town_mod_tavern"),    #Since he is in the field, take him from the tavern
                 (quest_set_slot,"qst_mod_trouble",slot_quest_current_state,2)  #Allow of a new conversation to start the duel
                ]
        ),
```

The last 2 lines do a bit of clean up leading into the duel. The comment makes what I am doing a bit self explanatory, as any of your comments you create should. Now come noon (time of day being 12) when the party "new_template" spawns (the party Geoffrey is in), he will no longer be found in the tavern in Mod Town. We have also advanced the variable "slot_quest_current_state" to 2.

Save your progress and run **build_module.bat**. So now we have the quest ready to start. If you play through it now, you will find that no matter the outcome of the battle with Geoffrey and his goons, you will fail the quest. This is because we do not yet have code here that will change "slot_quest_current_state" to 3, the success state.

******ALL WORKS UP TO THIS POINT******
NEED TO WORK OUT THE DIALOG FOR WHEN YOU CONFRONT GEOFFREY AND HIS THUGS. NEED TO LOOK AT WHEN YOU DEFEAT ANOTHER LORD.

PART 11

The next module file we will pay attention to is **module_mission_templates.py**. This file contains all the combat code. This can be a standard battle on the field, to laying siege to a castle, to dueling for a lady's honor. There are also a few quest specific battles such as killing the merchant. If you need to test outcomes of combat, this is the file you will need to work in. Thanks to Phosphoer and MartinF, we will first look at creating a duel (1 on 1 in the arena). After that we will then look at the standard battle in the field to change values based on its outcome.

11.1 -- Breakdown of Module_Mission_Templates.py
As stated above, anytime you have a confrontation battle, these code blocks are checked. You can also set up what macro keys (such as TAB) will do during the battle. Before the meat of the mission_templates, a bunch of local constants are set. The first two are designed as item overrides. We will see more on this later, but if you are picking up the workings of the module system, you should understand what they do. Next you will see some other longer tuple constants, such as common_battle_mission_start. You should recognize the format of these constants. They look just like tuples from **module_triggers.py**. This is because they are triggers. As you can see in the header of module_mission_templates.py, the 6th part of these tuples are triggers. By defining some common triggers at the top as constants, you will not have to type these triggers for each tuple that will need them.

At about line 650, you will see the start of the mission_templates(mission_templates = [ ). Let's look at the breakdown of the first mission template tuple "town_default" as defined in this file's header:

1. Mission template ID (prefixed with mt_) = "town_default"
2. Flags (as defined in header_mission_template.py) = 0
3. Mission Type interger. Should be 'charge' or 'charge_with_ally' otherwise must be -1 = -1
4. Mission description text = "Default town visit"
5. List of spawn records (list). Breakdown of the list (entry-no spawn point, spawn flags, alter/equipment override flags, AI flags, # of troops to spawn, list of equipment to add to troops spawned here-max 8) =

```
        [(0,mtef_scene_source|mtef_team_0,af_override_horse,0,1,pilgrim_disguise),
        (1,mtef_scene_source|mtef_team_0,af_override_horse,0,1,[]),(2,mtef_scene_source|mtef_team_0,af_override_horse,0,1,[]),
        (3,mtef_scene_source|mtef_team_0,af_override_horse,0,1,[]),

       (4,mtef_scene_source|mtef_team_0,af_override_horse,0,1,[]),(5,mtef_scene_source|mtef_team_0,af_override_horse,0,1,[]),(6,mtef_scene_s
       ource|mtef_team_0,af_override_horse,0,1,[]),(7,mtef_scene_source|mtef_team_0,af_override_horse,0,1,[]),

       (8,mtef_scene_source,af_override_horse,0,1,[]),(9,mtef_scene_source,af_override_horse,0,1,[]),(10,mtef_scene_source,af_override_horse,0,
       1,[]),(11,mtef_scene_source,af_override_horse,0,1,[]),
       (12,mtef_scene_source,af_override_horse,0,1,[]),(13,mtef_scene_source,0,0,1,[]),(14,mtef_scene_source,0,0,1,[]),(15,mtef_scene_source,0,
       0,1,[]),

       (16,mtef_visitor_source,af_override_horse,0,1,[]),(17,mtef_visitor_source,af_override_horse,0,1,[]),(18,mtef_visitor_source,af_override_horse
       ,0,1,[]),(19,mtef_visitor_source,af_override_horse,0,1,[]),(20,mtef_visitor_source,af_override_horse,0,1,[]),(21,mtef_visitor_source,af_overrid
       e_horse,0,1,[]),(22,mtef_visitor_source,af_override_horse,0,1,[]),(23,mtef_visitor_source,af_override_horse,0,1,[]),(24,mtef_visitor_source,af
       _override_horse,0,1,[]),

       (25,mtef_visitor_source,af_override_horse,0,1,[]),(26,mtef_visitor_source,af_override_horse,0,1,[]),(27,mtef_visitor_source,af_override_horse
       ,0,1,[]),(28,mtef_visitor_source,af_override_horse,0,1,[]),(29,mtef_visitor_source,af_override_horse,0,1,[]),(30,mtef_visitor_source,af_overrid
       e_horse,0,1,[]),(31,mtef_visitor_source,af_override_horse,0,1,[]),],
```

6. List of triggers (see module_triggers.py for format on triggers) =

```
 [
   (1, 0, ti_once, [], [
      (store_current_scene, ":cur_scene"),
      (scene_set_slot, ":cur_scene", slot_scene_visited, 1),
      (try_begin),
        (eq, "$sneaked_into_town", 1),
        (call_script, "script_music_set_situation_with_culture", mtf_sit_town_infiltrate),
      (else_try),
            (eq, "$talk_context", tc_tavern_talk),
            (call_script, "script_music_set_situation_with_culture", mtf_sit_tavern),
          (else_try),
            (call_script, "script_music_set_situation_with_culture", mtf_sit_travel),
          (try_end),
        ]),
      (ti_before_mission_start, 0, 0, [], [(call_script, "script_change_banners_and_chest")]),
      (ti_inventory_key_pressed, 0, 0, [(set_trigger_result,1)], []),
      (ti_tab_pressed, 0, 0, [(set_trigger_result,1)], []),
    ],
  ),
```

We will be creating a new mission_template which will be similar to the standard duel for the lady's honor. We will add our new template to the end of the file before the last closing bracket ']'.

By now you should have the hang of checking the top of the file for information on the tuple format, so I will forgo that, and get into making our Duel mission. Let's move down to the end of the file. As mission_templates are called, the order here shouldn't matter. Move to the end of the file and make some space before the last closing bracket ']'. By looking at some of the other Duel tuples here, we can work out the best way to make our custom duel with Geoffrey.

As with all tuples, we start with the opening parentheses '('. As a way to track where it ends, I like to also place the closing parentheses with the comma separator at the end '),". We will give it is ID. It should look like this:

```
  ("arena_duel_geoffrey",

  ),
```

Looking at the description of the tuples here, we know that next we need the mission flag and the mission type. This will be an arena fight, so we'll give it the mtf_arena_fight flag, and the mission type **-1**. From the top we know that if it's not charge or charge_with_ally, it should be **-1**. Next will be the mission text description. "Dueling Geoffrey" should do. Pretty basic so far. It should look like this:

```
  ("arena_duel_geoffrey",mft_arena_fight,-1,
   "Dueling Geoffrey",

  ),
```

Make sure to remember the commas that divide each part of the tuple. We are now getting into the more complex parts of the tuple. We are now setting up how the various parties in the mission will be placed. The spawn records are kind of like a tuple in the mission_template tuple. The first part is the entry point the team starts on. It is important to know your entry points on specific scenes. We will be using a standard arena for our scene. If you search for "arena_melee_fight", you can see examples of the spawn records. The next tuple "arena_challenge_fight", is more close to what we are looking to do. 2 spawn records, for our 2 duelists. They use entry points 56 and 58. If you go to the arena and edit the scene, you will find that these 2 points are near the center. We will do the same as this is what we are looking to do. We'll come back to these 2 tuples for examples often for our duel mission_template.

Since our duel will be very much like the arena_challenge_fight, we will copy these 2 spawn records to our tuple. Let's take a look at the rest of the list in the spawn records:

```
      (56, mtef_visitor_source|mtef_team_0, 0, aif_start_alarmed, 1, []),
      (58, mtef_visitor_source|mtef_team_2, 0, aif_start_alarmed, 1, []),
```

After the spawn entry points, we need to set the spawn flags. The first flag sets the confrontation style. Here we use mtef_visitor_source. This can also be set to mtef_attackers or mtef_defenders, though these flags are for field encounters. Attackers would be the party initiating the encounter (usually the player?), the defender would be the other party.

The second spawn flag (after the '|') mtef_team_# sets which team the spawned troop would be part of. This ranges from mtef_team_0 to mtef_team_3, meaning a maximum of 4 teams. We now look at the alter flags. These are set to override certain items that may not be allowed in this mission. For our mission, it would be more sporting to fight on foot. We will override horses. The next flag is the AI flag. I have always seen it set to aif_start_alarmed, so let's keep it that way. This makes the AI ready to fight. After this we set the number of troops to spawn. Since we need no more than the heroes, we are spawning 1 troops.

The last part of the list (inside the '[]') is for equipment to add to the troops spawned here, to a maximum of 8 items. If you override all other equipment, the troop will pick from what is listed here. If you want the troop spawning here to have a choice between a wooden staff or a wooden sword, override all equipment (alter flag af_override_all) and add the equipment to be randomly chosen.

At this point your mission_template should look like this:

```
  ("arena_duel_geoffrey",mft_arena_fight,-1,
   "Dueling Geoffrey",
   (56, mtef_visitor_source|mtef_team_0, af_override_horse, aif_start_alarmed, 1, []),
   (58, mtef_visitor_source|mtef_team_2, af_override_horse, aif_start_alarmed, 1, []),

  ),
```

We have the two troops spawning at different points, and they are not allowed to use horses. We're almost done here. Next we need to put in some triggers. The reason we need triggers, is to check, on timed intervals, if things have happened. What things? Things like being knocked out, Timed battles, or possible scene changes like knocking down a door. Right now, we will just play with who gets knocked out. Since there are 2 different parties here, we will have make a trigger for each.

We also want to prevent players from accessing their inventory. There is a pre-made trigger for this called common_inventory_not_available. This will be or first constant trigger.

Let's start with if you are knocked out. We need to test this quite often, but not too often. There can be a bit of delay as you slump to the floor. Remembering the tuple format for triggers, the first thing is when to run the trigger. I would assume that this will be in seconds. We'll start it on the 1st unit of time. The second part of the tuple is the delay until we check again. 3 seconds should be good. We will also make sure that the trigger runs once, after that it will not check again. Luckily, they have a check built in to see if you've been beaten, (main_hero_fallen). With this condition on the trigger, we would have to also set the code block to execute if this condition is met. This would mean that we should set the fail state for the quest, then exit the mission. You mission_template should look like this now:

```
  ("arena_duel_geoffrey",mft_arena_fight,-1,
   "Dueling Geoffrey",
   (56, mtef_visitor_source|mtef_team_0, af_override_horse, aif_start_alarmed, 1, []),
   (58, mtef_visitor_source|mtef_team_2, af_override_horse, aif_start_alarmed, 1, []),
     [common_inventory_not_available,
        (ti_tab_pressed, 0, 0, [(display_message, "@Geoffrey: Trying to flee, Peasant?")], [])
        (1, 3, ti_once,[(main_hero_fallen),],
          [(quest_set_slot,"qst_mod_trouble",slot_quest_current_state,4),
           (finish_mission),
          ]
        ),

    ],
  ),
```

We ended off with the pre-defined code (finish_mission). This will end the mission_template. If not, you would be stuck in the mission until you exited the game. Well, maybe not. The player can press TAB and get the generic tab menu. We don't want that, it might ruin the outcome of the duel. Right after common_inventory_not_available, we will add another trigger based on pressing the TAB key. Add this line:

```
(ti_tab_pressed, 0, 0, [(display_message, "@Geoffrey: Trying to flee, Peasant?")], [])
```

ti_tab_pressed is a defined trigger action so we don't need to set a delay or a re-arm. This should happen every time the player presses the TAB key. In this case, they will get a message from Geoffrey. We want to make sure that the player stays in till the end.

Lastly we will test if the player wins. Similar to when the player is knocked out, we can use a condition state that checks if all enemies are defeated, all_enemies_defeated. If the enemies are defeated, this value will be **1**. We also should check to see if both opponents are defeated at the same time (such as by an arrow or bolt). This way we don't get mixed outcomes. If the player is defeated at all them mission will fail. So we check the neg of the previous check. To end it off, don't forget to change slot_quest_current_state and end the mission template. You finished product should be similar to this:

```
  ("arena_duel_geoffrey",mft_arena_fight,-1,
   "Dueling Geoffrey",
   (56, mtef_visitor_source|mtef_team_0, af_override_horse, aif_start_alarmed, 1, []),
   (58, mtef_visitor_source|mtef_team_2, af_override_horse, aif_start_alarmed, 1, []),
     [common_inventory_not_available,
        (ti_tab_pressed, 0, 0, [(display_message, "@Geoffrey: Trying to flee, Peasant?")], [])
        (1, 3, ti_once,[(main_hero_fallen),],
          [(quest_set_slot,"qst_mod_trouble",slot_quest_current_state,4),
           (finish_mission),
          ]
        ),

        (2, 3, ti_once,[( all_enemies_defeated, 1),(neg|main_hero_fallen, 0),],
          [(quest_set_slot,"qst_mod_trouble",slot_quest_current_state,3),
           (remove_troop_from_site,"trp_npc17","scn_town_mod_tavern"),  ##Remove Geoffrey from the tavern scene
           (finish_mission),
          ]
        ),
    ],
  ),
```

I added one more part that should clean things up. If I didn't remove Geoffrey from the scene, he would still give dialogs based on the current state of the quest. You may notice that I didn't remove him from if he wins. I did this purposely. You can (based on the value of slot_quest_current_state) set up a dialog that Geoffrey will say to the player to taunt him in his loss. You can try it yourself, but I will clean this stuff up at the end of the next part.

PART 12

Heraldic armor basically displays the family crest/coat of arms/banner/"heraldry" on it. But this section is not limited to that. I will also show you have to have your soldiers carry your team color as well. If you have organized your banners (for me there are 8 colors of 8 banners in order), you can also organize your backgrounds for the banners to match the banner color. You can do so with this command:

```
    (troop_set_slot, "trp_banner_background_color_array", 0, 0xFF8f4531),
```

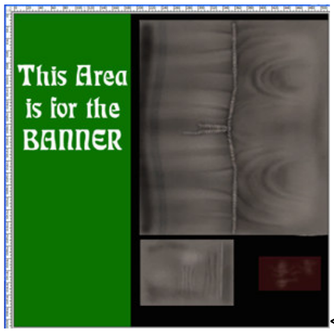
In the first script in module_scripts.py you can see where this is defined for each of the banners.

12.1 Textures and Alphas

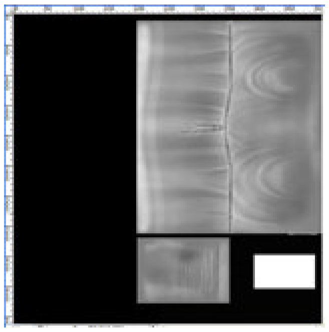
I am starting with the basic shirt. I will be making it match the banner background. Please note that I have my banners are color coded, and I have personally set the background colors for the banners to match the banner colors. This is adjusted at the beginning of the starting script.
First things first, let's take it's material and make a new one with an alpha. Open the costume1.dds file. I'm going to make my texture 512x512, so I will start a new file and copy the shirt texture (middle top) the sleeve of the same color, and the brown pants (any of them really, doesn't matter too much) to your new file.

I turned the shirt texture sideways to allow for more space for the banner image, which with my setting will be on the left side. In testing I have found that with the setting I use for the tableau_material, the banner will take up (counting 0, 0 as upper left) from 0, 0 to 195,512. What I did next was to mark this area of (in a separate layer) then scale the rest of the material (shirt, sleeves, pants) to leave this area free. If you were to leave the default settings that the other tableau_materials have, the banner will be in the center of the image.

Since I have played with this texture and the UV maps for it, I know how it is laid out. You may want to play around with the original mappings yourself first. Using SantasHelper's tips, We will make the alpha based on the original texture to keep the under lying texture of the cloth which will be under the background color. I'm not going to play with the pants, that will stay the same color as it is now. Select the areas of the shirt texture and make a copy and paste it into the alpha channel. I inverted the colors and I played a bit with the brightness and contrast. I put a black background (generic like), and ended up with this:



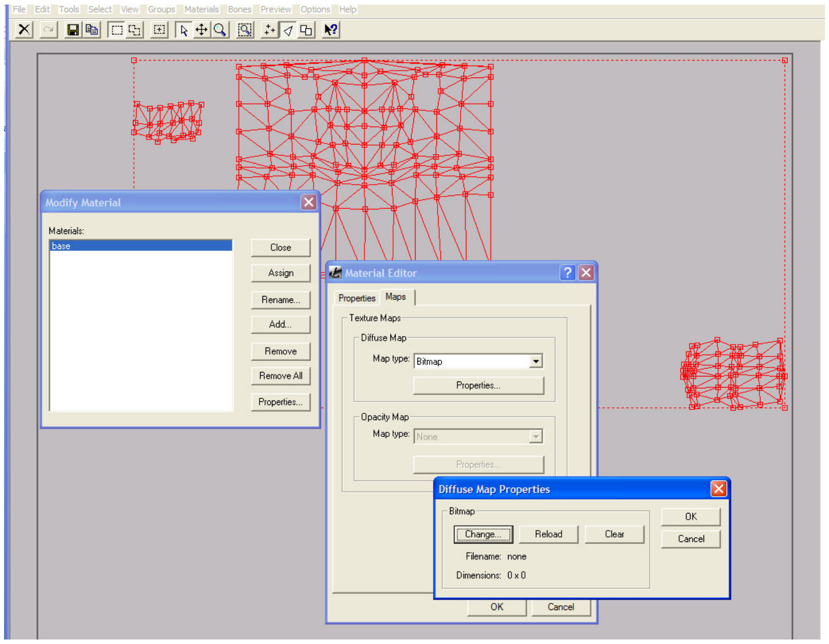
<Texture


<Alpha

I will now save this texture as j_cloth01.dds. Make sure to save it in the DXT5 format. With the texture done, we will now need the mesh to map it to. I will be using the **BRFeditor 9.8.5**, and a free UV mapping software called **LithUnwrap**. I will also be using an unofficial M&B tool called **obj2smd.exe**. You can grab obj2smd.exe from this link: http://forums.taleworlds.net/index.php/topic,55698.0.html (thanks to **Manitas**).

12.2 UV Mapping

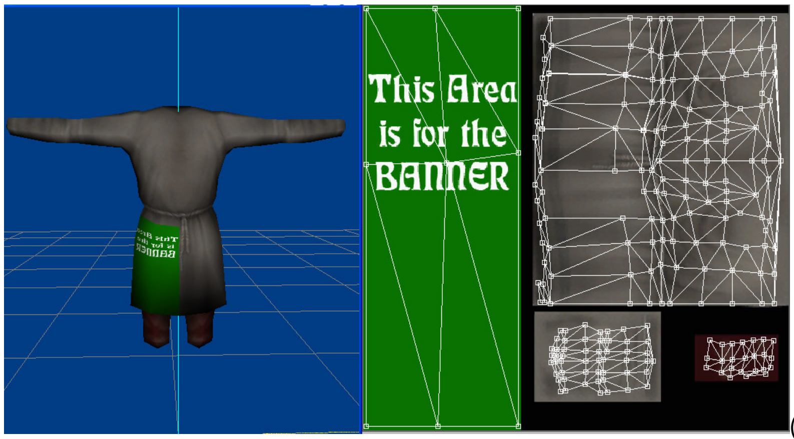
With the BRF editor, I will export the shirt in 2 formats, OBJ and SMD. With the SMD make sure to select the human skeleton. Using LithUnwrap, I will load in the OBJ file (File>Model>Open). You will see a wireframe of the UV map appear in the window. We now need to add our material. Click Material>Modify… If you see any materials there, highlight and remove them. Click Add, and give any description you want. I called mine 'base'. Highlight this material and click on Properties. In the Properties window, click the Maps tab. For the Diffuse map, change the type to bitmap, and click on Properites. Click Change and select your texture that you created from the first step (mine is j_cloth01.dds).



Once selected, click OK two times. You should be back with the Modify Material window. Make sure that your material is still selected, and click Assign. This will only work if you had selected the faces of the mesh, if not the assign button will not be functional. This will allow us to see the material on the mesh with the preview turned on. Click Close. Click on View>Material… Select your material and click apply. You should see it as the background. We need to make sure that the uvmap lines up with the texture.

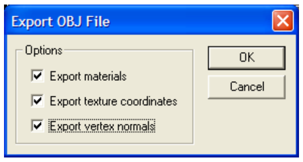
I will first move the sleeve and pants to the right spots, I will then rotate the main body of the shirt to match the orientation of the texture (rotate 90 clockwise). The scale is off a bit too. Since the original size was 1024x1024 and we scaled it down to 512x512, a 2x scale will be perfect. These options are found in the Edit menu. I won't be going into detailed usage of this software package, you will have to play around with that yourself. I will tell you that you should click Preview>Show Model, so you can see how the texture is looking. At this point I have it all lined up. A nice thing with this texture is that the tie of the rope belt is dead center front. So we can see which area is the front, which is the back. This will be helpful when deciding where the heraldry will be. Chest center, right breast, back, it's up to you. I will be using a strip of cloth just to the side of the rope belt on the leg.

One of the drawbacks of this software is it's face selection. To select a face, you need to marquee around a vertex it's a part of. The bad part of this, is that you also get the other faces attached to this vertex. Unlike most other software packages, deselecting is done by holding Ctrl (not Alt), so you can deselect the unwanted faces. For what I needed, I shifted the vertices to move the robe belt a bit to the left, then I took the faces from the right side, and moved them over to the blank space. I scaled it up a bit as well (by about 2.5x) Right now it looks like this:



(Preview on left, UV map on right.)

As you can see the black area in the preview is the part of the UV map that I put out to the lower left. When I define the tableau_material, this is where the banner will appear, at about that size. We are done here, so click File>Model>Save, to save it. Make sure to specify the right folder to save it in (I often get caught saving it in the texture folder since that was the last one used by the program), and make sure it's OBJ format.
A pop up window will ask you to save what settings, check all boxes:


Export OBJ File

Options
- [✓] Export materials
- [✓] Export texture coordinates
- [✓] Export vertex normals

Next we are going to merge the OBJ file with the SMD file to copy the rigging back. Unfortunately, the export of the OBJ does not store the animation information, but the SMD file does. Manitas' little EXE file will combine the 2. If you have UVmapping software that works with SMD files, then you can bypass this and do the above steps with the SMD file.

There is a readme with the **obj2smd.exe** file, so I will not explain it's usage. When you are done, rename the in.obj and the out.smd to your new object name, in my case this will be j_cloth01.obj and j_cloth01.smd. It's a good idea to keep them both as you may want to go back and make changes. The in.smd file is obsolete. You can now import your new SMD file into the BRF editor. Again, read the readme file with **obj2smd.exe** as it has info on this process too.

12.3 BRFing around

I will not be training you here on the general usage of the BRF editor. I will assume you know the basics, and have imported models into a Mod before. If you haven't, this is not the place to start learning.

Open the BRF editor. We need to do some set up before bringing in our new armor model. First we need the BRF editor to know the texture. Click the Tex tab. Click add at the bottom. A default texture DDS file name will appear there. Highlight it and change it to the name of your DDS file, in my case, j_cloth01.dds. Next click on the Mat tab. I will do this the long way because I have had difficulties with the check boxes. Click on Import and find **heraldic_armors.brf** in the CommonRes folder (the main data area for M&B, not in your module). From here we will import 2 Materials, heraldic_armor_a and sample_heraldic_armor_a.

Once they are in your materials area, we will rename them using their naming structure. In my case they will be called j_cloth01 and sample_j_colth01. IF you are a careful user, you may instead want to clone them so you have the base samples until yours are right.
Since I am not using a specular map (it's just cloth) I will remove the specular info from sample_j_cloth.

We now have the textures and material setup. I have found that if you leave the heraldic material stuff we imported that you sometimes get load errors, so remove it before going on. Since we have set our own materials right, they can be the new templates for later materials.

Click on the Mesh tab. Click Import, and import your recently merged SMD file. Remember to remove the checkbox for Y/Z axis swap! We will now assign our texture to the imported model. Type in the material name, in my case this is j_cloth01 (not the sample!). Save the BRF. Make sure that if this is not one of the BRFs you have as part of your mod, that you add it accordingly. We will need to import once more. Import tableau_mesh_heraldic_armor_a from the **heraldic_armors.brf** file. Rename it to match our naming structure (tableau_mesh_j_cloth01 for me), and change its material to match you main object's material (again for me it would be j_cloth01). If you cloned it, once it is set, remove the heraldic one. Save your BRF file.

This ends the graphic part of things, now it's time for the code side.

12.4 Meshes and Tableau materials

Open up **module_meshes.py** and search for ***tableau_mesh_heraldic_armor_a***. This will bring you down to where similar items are defined. Like in most cases, we will copy this entry and change it to fit our design. Following my example it should look like this:

```
###JIK Heraldic (banner) armor tableau meshes
  ("tableau_mesh_j_cloth01", 0, "tableau_mesh_j_cloth01",  0, 0, 0, 0, 0, 0, 1, 1, 1),
```

I will say it again, comment the areas you are playing with. Also, I will not be going over the header info for this or other modules. If you are this far in the tutorial, you can look it over yourself. This is all we need here, so let's move on to **module_tableau_materials.py.** Search for ***heraldic_armor_d***. Copy this tuple just below itself. We are going to make some changes here:

```
        ("j_cloth01", 0, "sample_j_cloth01", 512, 512, 0, 0, 0, 0,
              [
               (store_script_param, ":banner_mesh", 1),

               (set_fixed_point_multiplier, 100),
               (store_sub, ":background_slot", ":banner_mesh", arms_meshes_begin), #banner_meshes_begin),
               (troop_get_slot, ":background_color", "trp_banner_background_color_array", ":background_slot"),
               (cur_tableau_set_background_color, ":background_color"),

               (init_position, pos1),
#              (cur_tableau_add_mesh_with_vertex_color, "mesh_heraldic_armor_bg", pos1, 200, 100, ":background_color"),
               (init_position, pos1),
               (position_set_x, pos1, -65),
               (position_set_y, pos1, 105),
               (cur_tableau_add_mesh, ":banner_mesh", pos1, 0, 0),
#              (cur_tableau_add_mesh, "mesh_banner_a01", pos1, 116, 0),
               (init_position, pos1),
               (position_set_z, pos1, 100),
               (cur_tableau_add_mesh, "mesh_tableau_mesh_j_cloth01", pos1, 0, 0),
               (cur_tableau_set_camera_parameters, 0, 200,200, 0, 100000),
              ]
        ),
```

Going over the changes in order, first we need a unique ID. Next is the sample material that we set up in the BRF. I don't understand why this is done this way, but I will go with the flow till told otherwise. Since my texture was 512x512, I left this alone. I next commented out the line defining mesh_heraldic_armor_bg. This line makes a small flag background for the banner. May need this or something similar later, but let's leave it out. If you wanted to, you can semi-alpha behind the designated banner spot to get the same effect. 2 lines down, I offset pos1 by -65x and 105y. This moves the banner image to the lower left of the texture area. Not sure why 0,0 is not in one corner, but this was done through testing. Next I changed the cur_tableau_add_mesh to match my tableau entry.

If you are just looking to keep the same background color, but don't need the banner image (possibly saving in processing) comment out the line **(cur_tableau_add_mesh, ":banner_mesh", pos1, 0, 0),**.

With that done, all we have to do is define our object and give it to someone.

12.5 Items definition

These type of items use triggers, so to get an idea of the triggers we should again look at an existing item. Do a search for heraldic_mail_with and you will get this:

```
["heraldic_mail_with_surcoat", "Heraldic Mail with Surcoat", [("heraldic_armor_a",0)], itp_merchandise| itp_type_body_armor  |itp_covers_legs ,0,
 3454 , weight(22)|abundance(100)|head_armor(0)|body_armor(49)|leg_armor(17)|difficulty(7) ,imodbits_armor,
 [(ti_on_init_item, [(store_trigger_param_1, ":agent_no"),(store_trigger_param_2, ":troop_no"),(call_script, "script_shield_item_set_banner",
"tableau_heraldic_armor_a", ":agent_no", ":troop_no")])]],
```

The important part is just after the ",imodbits_armor". This is the trigger. Not sure why a second trigger param is stored, it doesn't seem to be used, but we will leave it as is. We only need this part and can append it to a copy of the shirt tuple:

```
["j_cloth01", "Heavy Shirt", [("j_cloth01",0)], itp_merchandise| itp_type_body_armor |itp_civilian |itp_covers_legs ,0,
 47 , weight(2)|abundance(100)|head_armor(0)|body_armor(10)|leg_armor(2)|difficulty(0) ,imodbits_cloth, [(ti_on_init_item, [(store_trigger_param_1,
":agent_no"),(store_trigger_param_2, ":troop_no"),(call_script, "script_shield_item_set_banner", "tableau_j_cloth01", ":agent_no", ":troop_no")])]],
```

The only change was to change the tableau name to match our tableau tuple ID. If you didn't run **build_module.bat** between editing the module files, you will notice errors the first time your run it. Run it a second time, and the errors should be gone.

Add this item to your troop and try it out.

12.6 LOD is your Friend

As you looked through the various BRFs you would notice that some models have LOD models associated with them. With shirt, there is shirt.lod2. The older models have only once instance, but most of the new ones (especially the heraldic type armors) have lod1 to lod3. LOD stands for Level Of Detail, and as you look at these other models, you can see that they lose detail (polygons) as they move towards lod3. This is done to save on processing the graphics for models that are far away. If not, even though things would be too small to see, the player's computer would still do all the high detail calculations.

Making LOD models are easy in some ways and hard in others. If you are basing your texture off a model with only 1 LOD alternate (like with the shirt having only LOD2), you will have to find armor with similar features for the LOD1. LOD2 and LOD3 have barely any details, so you can pick any LOD2/3 model as your base. You will have to UV map them, but if you were ok with the above, I think you can handle it.

For the M&B to recognize the LODs you need to follow the same naming structure. In my case it would be j_cloth01, j_cloth01.lod1, j_cloth01.lod2, j_cloth01.lod3.

Chapter 13(?)

~~The next module file we will pay attention to is **module_game_menus.py**. This file contains all game menus in M&B. Game menus serve as an intermediary between encountering a party and jumping to a scene, though they are flexible enough to have many other uses. The in-game Passage system also uses game menus for part of its functionality.~~

13.1 -- Breakdown of ????

JIK NOTE : Will be attempting to do this myself. Wish me luck…

We we encounter Geoffrey's party on the map, a conversation should ensue, then later a confrontation. All encounters on the map will use the standard combat mission_template unless you intervene with specific code. We can see this in the dialog with bandits (module_dialogs.py):

```
  [anyone,"bandit_barter_3a", [], "Heh, that wasn't so hard, was it? All right, we'll let you go now. Be off.", "close_window",[
    (store_current_hours,":protected_until"),
    (val_add, ":protected_until", 72),
    (party_set_slot,"$g_encountered_party",slot_party_ignore_player_until,":protected_until"),
    (party_ignore_player, "$g_encountered_party", 72),
    (assign, "$g_leave_encounter",1)
    ]],
```

There are a few things to note here. First the use of the slot slot_party_ignore_player_until. This value (by the lines preceding this one) is set to **72** hours after the current time. Not sure why (redundancy), but they next use the operation party_ignore_player, and set the encountered party to ignore the player, again for 72 hours. Lastly, they assign "$g_leave_encounter" to **1**. This is what stops the standard confrontation from happening. If we wanted to, we could set up a separate dialog here to either duel Geoffrey, 1 on 1, or fight the team of them, but we already have out private duel, so let's stick to the standard battle.

We will add 2 more dialogs here (makes things look nice and complete), with another back and forth before the

Appendix (i)

This appendix will attempt to store some of the less common but still useful functions and values.

$variable - variables declared with a $ are global variables. They are available to be referenced in any part of the code once they are defined. Like all variables, the default value is 0.
:variables – variables declared with a : are local variables. They only exist in the current code block, though the values can be passed as parameters for scripts.
slots – All tuple objects have slots (about 240) associated with them. They are a list of integer(numbers only) values which by default are 0. NATIVE assignments can be seen in module_constants.py. If a slot number is not used, then you may assign it in the module_constants.py. It is a bad idea to use slots that are currently assigned for different purposes.

Registry uses:

Reg(0)

- When a party is spawned, it is first stored in reg(0)