Modding Standards & Utilities (MSU)

Documentation for v0.6.6

# Skills

## Automated Resetting

MSU provides several functions to reset the m table of a skill back to its base values:

* softReset()
* hardReset()
* resetField( \_field )

MSU stores the base values of a skill immediately before its first onUpdate is called. This ensures that any changes made to the skill before or during adding to an actor are considered as the base state of the skill.

### softReset()

This function resets the following fields of the skill’s m table:

* ActionPointCost
* FatigueCost
* FatigueCostMult
* MinRange
* MaxRange

softReset() is **automatically** called during the update() function of the skill\_container before any skills’ onUpdate or onAfterUpdate are called. It can also be manually called using <skill>.softReset().

### hardReset()

This function is never automatically called, but can be manually called using <skill>.hardReset() to reset every value in the m table back to its original value.

### resetField( \_field )

This function can be used to reset a particular field back to its base value e.g. <skill>.resetField(“Description”).

### Use cases

This system allows changing a value in a skill’s m table in increments rather than assigning it particular values, which opens up possibilities for flexible and compatible modding. For example, now you can do:

function onAfterUpdate ( \_properties )

{

this.m.ActionPointCost -= 1

}

In the original game, doing this will cause the action point cost of the skill to continue to reduce indefinitely on every call to this function. However, with the MSU resetting system, this will ensure that the skill’s action point cost is only reduced by 1 compared to its base cost. Another mod can then hook the same function and add or subtract from the cost in further increments.

## Scheduled Changes

Skills in Battle Brothers are updated in the order of their SkillOrder. Imagine we have two skills:

* Skill A with this.m.SkillOrder = this.Const.SkillOrder.First
* Skill B with this.m.SkillOrder = this.Const.SkillOrder.Any

Whenever the skill\_container runs its update() or buildPropertiesForUse (which calls the onAnySkillUsed function in skills) functions, the respective functions are called on the skills in the order of their SkillOrder. Hence, skill A will update before skill B in the above example.

If you want skill A to modify something in skill B after skill B’s update, you would have to change the order of skill A to be something later than that of skill B e.g. set skill A’s order to this.Const.SkillOrder.Last. Usually this is quite doable.

However, there may be cases where you absolutely want skill A to be updated before skill B but still want skill A to be able to change something in skill B when skill B is updated. MSU allows you to do this via the scheduleChange function.

Multiple changes to the same skill can be scheduled by using the function multiple times. Scheduled changes are executed in the onAfterUpdate function of the target skill after its base onAfterUpdate function (i.e. the one defined in the skill’s own file) has run.

**Usage:**

<skill>.scheduleChange ( \_field, \_change, \_set )

\_field is the key in the m table for which you want to schedule a change

\_change is the new value

\_set is a boolean which defaults to false. If true, it sets the value of \_field to \_change. If false, and if \_field points to an integer or string value, then it adds \_change to the value of \_field.

**Example:**

The following code is written in skill A and will reduce the action point cost of skill B by 1 even if skill A updates before skill B.

function onUpdate ( \_properties )

{

skillB.scheduleChange(“ActionPointCost”, -1, false);

}

## Damage Type

MSU adds a robust and flexible DamageType system for skills. The purpose of this system is to allow skills to deal a variety of damage types and to streamline the injuries system. This system also eliminates the need to use this.m.InjuriesOnBody and this.m.InjuriesOnHead in skills. Only the DamageType needs to be defined.

Each skill now has a parameter called this.m.DamageType which can be set during the skill’s create() function. This parameter is an array which contains tables as its entries. Each table contains two keys: Type and Weight. For example:

this.m.DamageType = [

{ Type = this.Const.Damage.DamageType.Cutting, Weight = 75 },

{ Type = this.Const.Damage.DamageType.Piercing, Weight = 25 }

]

The above example will give this skill 75% Cutting damage and 25% Piercing damage.

When attacking a target, the skill pulls a weighted random DamageType from its this.m.DamageType array. The rolled damage type is then passed to the \_hitInfo in onBeforeTargetHit. The type of injuries the skill can inflict is also determined at this point based on what type of damage it rolled, and which part of the body is going to be hit.

The skill’s rolled damage type’s “Weight” parameter is passed to \_hitInfo. This allows the target to access this information and receive different effects depending on how much weight of that DamageType the skill has.

MSU also adds the damage types of a skill to the skill’s tooltip automatically including their relative probabilities.

### Adding a new DamageType

this.Const.Damage.addNewDamageType ( \_damageType, \_injuriesOnHead, \_injuriesOnBody )

\_damageType is a string which will become the name of a slot in the this.Const.Damage.DamageType table

\_injuriesOnHead and \_injuriesOnBody are arrays of strings where each entry is an ID of an injury skill

### Getting a list of injuries a damage type can inflict

this.Const.Damage.getInjuriesForDamageType ( \_damageType )

\_damageType is a slot in the this.Const.Damage.DamageType table.

Returns a table { Head = [], Body = [] } where Head and Body are arrays containing IDs of the injury skills this damage type can inflict on the respective body part.

### Getting a list of injuries applicable to a situation

this.Const.Damage.getApplicableInjuries ( \_damageType, \_bodyPart, \_targetEntity )

\_damageType is a slot in the this.Const.Damage.DamageType table.

\_bodyPart is the body part hit

\_targetEntity defaults to null. If not null, it removes the ExcludedInjuries of the \_targetEntity from the returned array.

Returns an array which contains IDs of the injury skills that this damage type can apply in the given situation.

### Checking if a skill has a DamageType

<skill>.hasDamageType ( \_damageType, \_only )

\_damageType is a slot in the this.Const.Damage.DamageType table.

\_only is a Boolean that defaults to false. If true, then this function only returns true if the skill has no other damage type.

Returns a true if the skill has the damage type and false if it doesn’t.

### Adding a damage type to a skill

<skill>.addDamageType ( \_damageType, \_weight)

\_damageType is a slot in the this.Const.Damage.DamageType table.

\_weight is an integer

Adds the given damage type to the skill’s this.m.DamageType array with the provided weight.

### Removing a damage type from a skill

<skill>.removeDamageType ( \_damageType )

\_damageType is a slot in the this.Const.Damage.DamageType table.

Removes the given damage type from the skill if the skill has it.

### Getting the damage type of a skill

<skill>.getDamageType()

Returns the this.m.DamageType array of the skill.

### Getting the weight of a skill’s particular damage type

<skill>.getDamageTypeWeight ( \_damageType )

\_damageType is a slot in the this.Const.Damage.DamageType table.

Returns an integer which is the weight of the given damage type in the skill. Returns null if the skill does not have the given damage type.

### Setting the weight of a skill’s particular damage type

<skill>.setDamageTypeWeight ( \_damageType, \_weight )

\_damageType is a slot in the this.Const.Damage.DamageType table.

\_weight is an integer

Finds the given damage type in the skill’s damage types and sets its weight to the given value. Does nothing if the skill does not have the given damage type.

### Rolling a damage type from a skill

<skill>.getWeightedRandomDamageType()

Selects a damage type from the skill based on weighted random distribution and returns it. The returned value is a slot in the this.Const.Damage.DamageType table. For example, this function is used by MSU in the msu\_injuries\_handler\_effect.nut to roll a damage type from the skill when attacking a target.

## Item Actions

Item Action refers to swapping/equipping/removing items on a character during combat. MSU completely overhauls how the action point costs for this are handled. Now you can add skills that modify how many Action Points it costs to swap items, depending on what kind of items they are. This is accomplished via three things:

1. ItemActionOrder
2. getItemActionCost ( \_items )
3. onPayForItemAction ( \_skill, \_items )

**ItemActionOrder** defines the priority in which a skill is consumed for changing the AP cost of an item action. This will become clearer later. The values are defined in a table this.Const.ItemActionOrder and can be expanded.

**getItemActionCost ( \_items )**

This is a function defined in a skill. By default it returns null for all skills, but you can choose to return a different value depending on what you want this skill to do. For example, Quick Hands now uses this system and always returns 0 as long as the this.m.IsSpent variable is set to false in the perk. \_items is an array which contains the items currently involved in the action e.g. when swapping a sword for a mace the two items will be in the array. Hence, the skill has access to the items and can decide to return null or a different value based on what items are being swapped. So, if you want a skill that sets the action cost of swapping tools to 0 but not reduce the action point cost of weapons, you can do that now.

When determining how many action points it costs to do an item action, the skill\_container goes through all the skills on the actor and chooses the one with the lowest AP cost. If multiple skills provide the lowest AP cost then it chooses the one with the earliest ItemActionOrder value. This allows you to prioritize which skills get used e.g. a more specific skill gets used instead of a more flexible one.

If you want this skill to not change the AP cost of an item action make sure to return null in this function.

**onPayForItemAction ( \_skill, \_items )**

This function is called on all skills once the item action has been performed and its action point cost has been paid. \_skill is the skill which was chosen by the skill\_container to reduce the AP cost, if such a skill was present, otherwise it is null. \_items contains the array of the items involved in the item action.

### Example

For example, in the new system Quick Hands is implemented as given below. For those of you who have delved into how item action costs are handled in vanilla Battle Brothers, you will notice how much more robust and flexible this system is. The vanilla system is extremely tightly coupled.

this.m.IsSpent = false

this.m.ItemActionOrder = this.Const.ItemActionOrder.Any

function getItemActionCost ( \_items )

{

return this.m.IsSpent ? null : 0;

}

function onPayForItemAction ( \_skill, \_items )

{

if ( \_skill == this)

{

this.m.IsSpent = true;

}

}

function onTurnStart()

{

this.m.IsSpent = false;

}

## Injuries

MSU adds a system to exclude certain sets of injuries from certain entities easily. MSU comes with the following sets of injuries built-in (only include vanilla injuries):

* Hand
* Arm
* Foot
* Leg
* Face
* Head

### Creating a new set of excluded injuries

this.Const.Injury.ExcludedInjuries.add( \_name, \_injuries, \_include )

\_name is a string

\_injuries is an array containing skill IDs of injuries

\_include is an array which defaults to an empty array. If passed, this must be an array of slots from the this.Const.Injury.ExcludedInjuries table.

Creates an entry in the this.Const.Injury.ExcludedInjuries table with \_name as key and \_injuries as value. Extends the \_injuries array with the injuries in \_include. The slots passed in the \_include array must already exist.

Example:

this.Const.Injury.ExcludedInjuries.add(

“Hand”,

[“injury.fractured\_hand”, “injury.crushed\_finger”]

);

this.Const.Injury.ExcludedInjuries.add(

“Arm”,

[“injury.fractured\_elbow”],

[this.Const.Injury.ExcludedInjuries.Hand]

);

### Getting a set of excluded injuries

this.Const.Injury.ExcludedInjuries.get ( \_injuries )

\_injuries is a slot in the this.Const.Injury.ExcludedInjuries table

Returns an array including the skill IDs of the injuries.

Example:

this.Const.Injury.ExcludedInjuries.get(this.Const.Injury.ExcludedInjuries.Arm)

### Adding excluded injuries to actors

<actor>.addExcludedInjuries ( \_injuries )

\_injuries is a slot in the this.Const.Injury.ExcludedInjuries table

Adds the injuries defined in that slot to the actor.

Example:

Let’s say you want to prevent Serpents from gaining Leg injuries. So you hook the onInit function of the serpent actor and add:

this.addExludedInjuries(this.Const.Injury.ExcludedInjuries.Leg)

## Other Utility Functions

### Getting a list of skills from skill\_container

MSU adds an efficient way of getting particular skills from skill\_container:

<skill\_container>.getSkillsByFunction ( \_self, \_function )

\_self is the object from which the function is being called, usually a skill. Hence it is passed as **this**.

\_function is a function which defines how the skills are selected

This allows you to pass a function by which the skills will be selected and returned. This removes the need for multiple iterations over the skills array when finding a particular skill. For example:

**Old system:**

this.m.Skills = [ “skillA”, “skillB” ]

local actives = skillContainer.getAllSkillsOfType(this.Const.SkillType.Active)

foreach (active in actives)

{

if ( this.m.Skills.find(active.getID()) != null)

{

// Do something

}

}

**New system:**

this.m.Skills = [ “skillA”, “skillB” ]

local skills = skillContainer.getSkillsByFunction(this, @(\_skill) this.m.Skills.find(\_skill.getID()) != null)

if (skills.len() > 0)

{

// Do something

}

### Using a skill

MSU adds the following function(s) which are run on all the skills of an actor immediately before and after using any skill. These functions can help to simplify the coding of many perks/effects.

1. onBeforeAnySkillExecuted ( \_skill, \_targetTile )
2. onAnySkillExecuted ( \_skill, targetTile )

\_skill is the skill being used and \_targetTile is the tile on which the skill is being used.

The first function is called before \_skill’s onUse happens, whereas the second function is called after onUse is complete.

### Actor movement

MSU adds the following function(s) related to actor movement.

1. onMovementStarted ( \_tile, \_numTiles )
2. onMovementFinished ( \_tile )

\_tile is the tile on which the movement was started/finished and \_numTiles is how many tiles the actor is trying to move (?).

During the calls to these functions the this.m.IsMoving Boolean of the actor is true.

### Damage

MSU adds the following function(s) related to actor damage:

1. onAfterDamageReceived()

This function is called after the actor receives damage.

### Time of Day

MSU adds the following function(s) related to time of day:

1. onNewMorning()

This is called when the time of day reaches Morning. This is different from the onNewDay() function which runs at noon.

# Weapons

## WeaponType and Categories

In the vanilla game, each item contains a this.m.Categories parameter which is a string and determines what is shown in the weapon’s tooltip e.g. “Sword, Two-Handed”. However, the actual type of the item is defined separately in this.m.ItemType. So it is entirely possible for someone to make a mistake and write “Two-Handed” in categories but assign this.m.ItemType to this.Const.Items.ItemType.Onehanded.

Similarly a weapon may be a Sword but someone can write “Hammer, One-Handed” in the categories and it won’t cause any errors. But this can lead to issues in terms of player confusion and especially if any mod adds skills/perks which require a certain type of weapon e.g. if the skill should only work with Swords.

MSU eliminates the need for manually typing this.m.Categories and builds this parameter automatically using assigned WeaponType and ItemType values.

### Weapon types

Weapon types are defined in a table this.Const.Items.WeaponType.

### Add new weapon types

New weapon types can be added to the system using:

this.Const.Items.addNewWeaponType ( \_weaponType, \_weaponTypeName )

\_weaponType is a string which will become a key in the this.Const.Items.WeaponType table.

\_weaponTypeName is an optional string parameter in case you want this weapon type’s name to show up differently in tooltips. If not given then the same string as \_weaponType is used as the name.

Example:

this.Const.Items.addNewWeaponType(“Musical”, “Musical Instrument”) will add a weapon type that can then be accessed and checked against using this.Const.Items.WeaponType.Musical and will show up as “Musical Instrument” in tooltips.

### Get the name of a weapon type

this.Const.Items.getWeaponTypeName( \_weaponType )

Returns a string which is the the associated name of \_weaponType. For instance, in the above example from 2.1.2 it will return “Musical Instrument” if this.Const.Items.WeaponType.Musical is passed as a parameter.

If \_weaponType does not exist as a weapon type, it returns an empty string.

### Adding a weapon type to a weapon

<weapon>.addWeaponType ( \_weaponType, \_setupCategories )

\_weaponType is a slot in the this.Const.Items.WeaponType table

\_setupCategories is a Boolean that defaults to true

Adds the given weapon type to the weapon. A weapon can have multiple weapon types.

If \_setupCategories is true, then MSU will recreate the this.m.Categories of the weapon.

### Removing a weapon type from a weapon

<weapon>.removeWeaponType ( \_weaponType, \_setupCategories )

\_weaponType is a slot in the this.Const.Items.WeaponType table

\_setupCategories is a Boolean that defaults to true

Removes the given weapon type from the weapon. Does nothing if the weapon doesn’t have the given weapon type.

If \_setupCategories is true, then MSU will recreate the this.m.Categories of the weapon.

### Setting a weapon’s weapon type

<weapon>.setWeaponType ( \_weaponType, \_setupCategories )

\_weaponType is a slot in the this.Const.Items.WeaponType table

\_setupCategories is a Boolean that defaults to true

Sets the weapon’s this.m.WeaponType to \_weaponType. You can pass multiple weapon types in \_weaponType by using the bitwise |.

If \_setupCategories is true, then MSU will recreate the this.m.Categories of the weapon.

### Checking if a weapon has a certain weapon type

<weapon>.isWeaponType ( \_weaponType, \_only )

\_weaponType is a slot in the this.Const.Items.WeaponType table

\_only is a Boolean that defaults to true

Returns true if the weapon has the given weapon type. If \_only is true then it will only return true if the weapon has the given weapon type and no other weapon type.

### Setting a weapon’s categories

<weapon>.setCategories ( \_categories, \_setupWeaponType )

\_categories is a string

\_setupWeaponType is a Boolean that defaults to true

Sets the weaopn’s this.m.Categories to \_categories.

If \_ setupWeaponType is true, then MSU will recreate the this.m.WeaponType of the weapon based on the new value of this.m.Categories.

# Logging

MSU adds additional logging capabilities for quicker debugging while modding.

**this.MSU.Log.printStackTrace ( \_maxDepth, \_maxLen, \_advanced )**

\_maxDepth is an integer which defaults to 0

\_maxLen is an integer which defaults to 10

\_advanced is a Boolean which defaults to false

Prints the entire stack trace at the point where it is called, including a list of all variables. If \_advanced is true, it also prints the elements of any array or table variables up to \_maxDepth and \_maxLen.

# Utilities

### Tile

this.MSU.Tile.canResurrectOnTile ( \_tile, \_force )

\_tile is a Battle Brothers tile instance

\_force is a Boolean that defaults to false

Returns false if there is no corpse on the tile. Returns true if there is a corpse on the tile which can resurrect or if \_force is true. This function can be hooked by mods to add additional functionality.

### String

this.MSU.String.getCapital ( \_string )

Returns the passed string with its first letter having been capitalized.

this.MSU.String.replace ( \_string, \_find, \_replace )

Finds the string \_find in the string \_string and replaces it with the string \_replace. Then returns the result.