

Item Boxes and Frenzies

Full Internals Guide

By Sam 78it

*CREDITS TO : **aturtledude** for his continuous invaluable help*

ver. 2.0

Foreword

This guide is not intended for normal players, but as a reference of information for dataminers. It's not simple, it's complete.

For the sake of uniformity, in this document we will use some of the code terminology, which differs from the normal english one. The terms we will use are:

- Machine – the kart (Pipe Frame, Koopa Dasher, etc)
- Wing – the glider
- Kart – the combination of a Driver, a Machine and a Wing. Could be a Player or a CPU
- CPU or bot – a Kart used by the game, not by a real player
- Rank – the position in a race (1st, 2nd, etc)
- P2P or PVP – Multiplayer
- Fever – the Frenzy
- Rolling – the action of getting an item, no matter the source. It could be passing an item box or using a ticket
- Throw – it represent the obtaining of a random number. The term figuratively represent the thrown of a dice in order to roll it and get a random number. Each throw could have a different base (0-1, 0-100, etc). The majority of the random numbers thrown by the game are floating point numbers.

What you could expect to find? The document includes everything that was discovered while reversing the slots selection process, fevers included. It reports function and objects used in the process.

What you could expect to don't find? Actually many functions are heavily parameterized, and this parameters could be changed virtually everywhere in the code. It's impossible to discover all the various points in the code where this changes happens without having the real vanilla code. However, everything that was found has been included.

Base concepts

The game determine items and frenzies using "Probability Tables".

The tables are loaded at every race start from the assets and customized for each Kart. The advantage of loading the tables from the assets is to allow to adjust the parameters without the need of releasing a new version of the game (theoretically). The customized tables are then further filtered every time an item box is passed. As explained after in the document, even results outside the tables are possible under certain circumstances.

There are 4 different base collections of probability tables¹:

- MASTER – used for normal races
- BOSS – used in the boss challenge, the one where it should be defeated the gigantic bot
- P2P – Used in multiplayer with normal slots
- P2PFIXED – Used in multiplayer with 2 fixed slots

Each collection holds a set of “scriptable tables” with the probabilities for both the PLAYER and the CPU, including the fever probabilities. The collection matching the race characteristics it’s loaded into a `ScriptableRaceItemSlotTable`² object, and the structure is the following:

- `mProbTable` – this table holds the probabilities to determine which kind of item you will get when a fever is NOT triggered.
- `mFeverTableMaster` – this table holds the probabilities for the PLAYER to determine if a fever could be triggered and, once triggered, which kind of fever would be got.
- `mFeverTableCpu` – same as above, but for the CPU
- `mRankMapPvP` – this table is used in PVP. It has multiple purposes (explained in the rest of the document) and it overrides the normal behaviour.
- `mBakedData` – this table is used as a cache to check if a certain item has not probabilities at all to be got.

Inside the game, the items obtainable from the item boxes are listed (enumerated) as “`EItemSlotResult`”. Those represent what you can get as result of an item box, rather than the specific item. That’s because some results give multiple items, as it happens with Lucky7 or birdo’s egg. At the same time, internally when you have a “plus” skill (cause the driver reached a higher level), the results are considered differently than the normal ones. So from now on, what is obtainable from the item boxes will be called “result” rather than “item”, which instead represent what the results produce in the game.

Table Preparation

Before the race starts, the game creates in his memory one `ItemOwner` object for each Karts in the race, with customized settings in a `ItemOwnerSetting` object depending on the DKG combination and their characteristics. Then, from this customized settings, a function `prepareProbTable` it’s called for each Kart to load the proper `ScriptableRaceItemSlotTable`, process it and create a customized versions of the subtables.

The `ItemOwnerSetting` variables interested are:

- `mProbTableAsset` – a copy of the table loaded from the assets
- `mProbabilityTable` – the customized table for the normal results
- `mFeverProbabilityTable` – the customized table for the fevers
- `mSlotResultSetting`
- `mFlags`
- `mSpecialProbUpper`

Normal Results

The first tables to be prepared are the ones for normal results (when no fever is triggered). This is done for each one of the “slots” available, for each one of the possible ranks.

1 see appendixes B1-B4 for the dumps

2 see appendix A1 for the structure

Both the Player and the CPU could get up to 3 results, depending on the shelf or on the race settings. The results are held in placeholders called “slots”. Every Player’ slot has its own probability sub-table, all the CPU slots have the same sub-table. The tables are numbered based on their “strength”. 0 is the weakest in terms of kind of results it could receive, 2 is the strongest.

When the player has only 1 slot, it’s strength 2. When the player has two slots, they are of strength 2 and 1. This is done so even a bottom shelf driver could get its special skill item (cause they are not present in the table for slot 0). The position of the slots is decided randomly every time a item box is passed (so a strong item could appear every time in a different position). This means that holding an item could potentially block from getting some kind of special items, if the slot held is the strongest.

The probability sub-table for every slot is further divided in a different sub-table for every Rank, so the kind of results you could get will change if you cross the item box at position 1, 2, 3, etc.

So, summarizing, the structure is Slot/Rank/EltemSlotResult.

After every single section is retrieved, it’s filtered to exclude all the results which the Kart could never get, no matter what, and to exclude the “dead” entries (the tables contains more than what is really used in the game).

The customized table structure for the single Kart/Slot/Rank it’s the following³:

- `cNotZeroProbTable` - This is the filtered list with the results probabilities
- `cNotZeroProbTableAddtional` - This is the filtered list with the results additional probabilities
- `mEffectiveProbTable` - This is a list that is populated using the above ones every time an item box is passed. That’s needed cause even if all the results in the above tables are acceptable for the Kart, they could be not all acceptable in a specified moment cause of other restrictions applied to the items (for instance, a banana could be forbidden cause there are already too many copies of it on the race)
- `mProbabilitySum` - This is the sum of probabilities in the `mEffectiveProbTable`. This is needed to determine the random result cause the sum of the probabilities in a certain moment very likely will not amount to 100.

The criteria used to filter the results probabilities are the following:

- the result must be marked as “vaild” in a global catalog. This is used to exclude test or empty results
- the result must have the property `mIsEnableProbTable` in its global parameters. This is used to exclude support entries
- if the owner has a list of restricted approved `EltemSlotResult`, it must be one of those. This is used to force the result kind in some challenges (only red shells, only bombs, etc)
- if the global result parameters have the flag `isFeverOnly` enabled, the driver must be top shelf
- if it’s a special result (belogning to a driver skill as the Coin-Box or Lucky7), the Driver must have that skill
- The probability to get the result should be > 0

After the normal probabilities, the same is done for the additional probabilities.

The criteria used to filter them are all of the above, plus:

- `EltemSlotResult` should match the Wing skill. This is managed by the flag `mIsProbAdded` (must be true) of the object `SlotResultInfo` obtained through the Kart skill settings.

3 see appendix A2 for the structure

In this case, before the probability is added to the list, it's further adjusted with a factor retrieved from the Kart skill settings. This is recovered from the parameter `AdjustProb` of the object `SlotResultInfo` obtained through the Kart skill settings. It's a floating point number and it's always:

- 0.8 for Normal Wings
- 1 for Super Wings
- 1.2 for HE Wings

This means, for instance, that the chance to get a Coin using a HE Coing Glider is grater than using a Normal one.

The next step is to process the `mRankMapPvP` member of the scriptable table. This member is basically copied in the `mRankMap` property of the customized table as it is.

After that, the flag `mIsEnabledFirstSlotRotate` of the `mRankMapPvP` is checked. If it's true, then what was done for the above it's repeated using the property `mFirstProbTable` of the `mRankMapPvP` object and adding it to the property `mProbabilityOfFirstRotateInThisRace` of the customized table.

This table is used in substitution of the standard probabilities only for the first item box passed, and it's enabled ONLY in multiplayer.

Fevers

After the normals, it's the fevers time.

Depending on the kind of Kart, `mFeverTableMaster` or `mFeverTableCpu` from the scriptable table it's used to create a customized version of the table for the Kart.

The fever tables are subdivided in Ranks/Lap/EltemSlotResult. That's similar to the normal results, with the exception of Lap. That's because the fevers have different (increased) probabilities for the last lap, no matter how many laps a race holds. This means for instance that on a race with 5 laps, the ones from 1 to 4 will have the same base fever odds, the 5th will have different odds.

The structure of the customized fever kart table for each subsection is the following⁴:

- `mFeverProbability` - a base % probablity to trigger a fever
- `cEffectiveTable` - a dictionary of probabilities for determining the kind of fever after it's triggered, with `EltemSlotResult` as key and the probability as the value
- `cEffectiveTable` - same as above, but for the additional probabilities
- `mProbSum` - the sum of all of the above. This is generated when an item box is passed
- `mRankMap` - a copy of the `mRankMapPvP` property of the scriptable table

The criteria used to filter the fever types are all the ones for the normal results, plus:

- the probability to trigger the fever for the item (from `GameItemFeverProbabilityOfLapData::mProbOfEachItem`) must be > 0

Before the probability is added to the dictionary, it's further adjusted with a factor retrieved from the Kart skill settings. This is recovered from the parameter `AdjustProbFever` of the object `SlotResultInfo` obtained through the Kart skill settings. It's a floating point number, and when `EltemSlotResult` matches the Wing skill, the value it's:

- 0.8 for Normal Wings
- 1 for Super Wings
- 1.2 for HE Wings

4 see appendix A3 for the structure

If the glider skill doesn't match, it's always 1.

This means that with a Normal coin Wing it's less likely to get a coin fever, using a Super one will leave it untouched, using an HE will raise it. Using any other kind of Wing will leave the probability untouched in itself, but will affect it indirectly cause it will lower or rise parallel probabilities (for instance, a normal green shell glider since will affect negatively the shells, will rise the coin probability; an HE banana glider, since will rise the banana probabilities will lower consequently the coin ones).

If the corresponding `EltemSlotResult` row value in the `mAdditionalProbOfEachItem` table is > 0 , then the same is done for `cAdditionalEffectiveTable`, but this time the factor used is `AdjustProb` of the object `SlotResultInfo`, as it happens for the normal results.

At the moment, all the additional probabilities for the fevers are zeroed.

Passing through an Item Box

There are two different possible patterns for passing through an item box:

- With auto item activated and some item in the slots
- All the others

In the first case, the slots are first emptied by the game. This is done by the function

`ItemOwner::calcDelayedEvents`.

Basically this function populate and process a list of actions which are not executed in the exact moment they are triggered. This is needed for many reasons

- One is to avoid the object overlapping, as it could happen if two shells are thrown in the same exact instant. In the best scenario they will just trigger the destroy event for both being "physically" in contact.
- Another one it's that in a certain moment the slots could be busy because an event is using them. That's the case of a Kart which crosses an item box while in fever. The slots cannot be filled until the fever is finished, cause some frenzies alter the course parameters which in turn affects the probabilities. That's the case for instance of a banana fever: at the end of the fever there could be too many bananas on the course, forcing the banana to be inaccessible for the next slot, so the slot needs to be calculated at the fever end.

Of course there are many other cases, but for the sake of this document, those are the significant ones.

As soon as the slots are emptied or the timed actions (as fevers) are finished, the flow proceeds as normal calling the main function responsible for filling the slots, `startRotatingItemSlotImpl` from the `ItemOwner` object.

This function gets 3 significative parameters:

- `rotateType` - what led to this function. It could be:
 - `ItemBox` - passing through the item
 - `UIButton` - using a ticket
 - `Teresa` - Probably the ghost skill of stealing an item, which is not already implemented
- `itemGetNum` - how many slots needs to be refreshed. This is set only when coming from `calcDelayedEvents` (see below).
- `isSpecialFixed` - force the fever to be of the driver special item kind

The first thing this function does, is to increment the `mSlotRotateCount` property on the `ItemOwner` object, in order to keep track of how many times an item box has been crossed.

Then a series of checks to see if a fever is allowed are done.

Fevers

First the the function `ItemOwner::CheckAllSlots` it's called. This function check if a fever could be triggered.

In order to do that, it resets and refill a temporary list called "PossessList", which holds informations about every item already in the slots.

Then for every empty slot, it verify if there is a pre-determined result. This is done checking:

- if slot have a state "Decided", which basically forces the item selection for that specific slot
- If the `mWillStockItemSlot` property of the `ItemSlot` object is set

If so, the forced item is added to the PossessList.

After every item is added, the property `mNoFeverOnHaving` of the result parameters is checked. If It's true, then no fevers will be possible.

After checking if a fever is possible or not because of the slots and the item in the slots, `startRotatingItemSlotImpl` continues checking if a fever is forbidden. This is done checking:

- if special limitations are on, calling the function `ItemLimitationNotP2P::Check` which basically check the flag `hasOnlyOne` of the object `ItemLimitationNotP2P`. Despite the name, it seems this variable it's used as a flag for disabling the fevers (it could be a reuse of a parameter).
- If the flag `IsAllItemRun` of the object `KartInfoProxy` is settings
- if the flag `mIsNextSlotNoFever` of the `ItemOwner` object is set. This is used to disable subsequent frenzies.
- if `mFeverCount` of the `ItemOwner` object is greater or equal than the result of limit of fevers in one race

If a fever is allowed, then the function `ItemOwnerSetting::CalcFeverItem` is called, which determines if and which fever will be triggered.

In case a fever is triggered and the flag `isSpecialFixed` is set, the fever will be forced to be of the Driver special item kind regardless of this function result.

The function `CalcFeverItem` calls the function `CalcFever` from the `mFeverProbabilityTable` object of the `ItemOwnerSettings` object.

The `CalcFever` function first get the "normalized" rank index. This is needed with races with less than 8 karts total, usually in multiplayer.

First, if the `mAddedRank_LowerRankSlotTable` property of the `SkillSetting` object for the kart is > 0 , then its value is added to the normal rank index. If the resulting value it's \leq of the total karts in the game (minus 1), then it's taken, otherwise the last rank index depending on the number of karts it's taken.

This "normalized" rank index it's then used to get the final rank index matching it with the `mRankMap` property of the `ItemFeverProbabilityTable` object. This is done only when the `mIsEnabled` property it's true, and this happens only in multiplayer. If the resulting `MappedRankIndex` is negative, then the fever is forbidden.

If this phase is passed, then the `mFeverTable` property is checked. If is set, then the corresponding `ItemFeverProbabilityOfRank` object is taken, the lap index is retrieved and the `ItemFeverProbabilityOfRank::CalcFeverItem` is called with the proper lap index. The function getting

the lap index will return 0 for every lap except the last, where it returns 1. So the only lap with different (increased) odds, it's the last. In this phase, the `rotateType` parameter is always set to 0, so for the frenzy purpose the source which triggered the rolling (if ticket, item box, etc) is meaningless.

The function `ItemFeverProbabilityOfRank::CalcFeverItem` is finally the one responsible for determining if the fever is triggered. This function first determine if the goal has been passed. If so, the fever is not permitted.

Then the proper `ItemFeverProbabilityOfLap` object is taken from the `mProbOfLap` property of the `ItemFeverProbabilityOfRank` object.

The game then get a random float between 0 and 100, and uses this number to compare with the `ItemFeverProbabilityOfLap::GetFeverProbPercent` function result. If the random number is \geq , then the fever it's not triggered, otherwise it's triggered.

`GetFeverProbPercent` returns `ItemFeverProbabilityOfLap::mFeverProbability` plus `ItemOwnerSkillSetting::mFeverProbUpPercent` in case of a normal race, `ItemRankMapData::mFeverProbOfLap` plus `ItemOwnerSkillSetting::mFeverProbUpPercent` in case of a P2P race. This value is "clamped" to a global limit for the fever probability taken from global settings for the items, by default with the value of 50 (`GameScriptableItemParamGeneral::mFever::mMaxFeverProbPercent`).

The property `ItemOwnerSkillSetting::mFeverProbUpPercent` represent the driver frenzy bonus.

If the frenzy is triggered, the function `ItemFeverProbabilityOfLap::CalcFeverItem` it's then called. This function is responsible to determine which kind of frenzy will be triggered.

This function will first determine the sum of all the single item probabilities. In order to do that, all the available items will be cycled and filtered by the function `ItemInnerUtil::CanGetFever`.

This function checks:

- if there are too many items of a kind in the race
- if enough time it's elapsed since the beginning of the race
- if enough time it's elapsed since the last time an item was got
- if the kind of frenzy is allowed in the race type

The same is done for the additional probabilities, if present.

After that, the game determine a random number between 0 and the sum. This number it's compared to the results probabilities in the order they are present in the probability table first, in the additional probability table after. During this procedures all the checks specified above are repeated many times.

In the end, if auto-item is disabled and there are items in the slots, the game checks if the result it's compatible with what it's already on the slots. So for instance, if a banana is present in the slots, only banana frenzy are allowed. If a different frenzy was triggered, it will be ignored.

At this point the control is gived back to the main function, `startRotatingItemSlotImpl`, and as said before if the parameter `isSpecialFixed` it's set, the frenzy will be replaced with the special item owned by the driver.

If a frenzy is not triggered, then the game proceed in determining the single slots.

Single Results

According to the probability tables, there are three different lists possible for the slots, each one with its own “strength”: 2 is the maximum strength, 0 is the minimum. Very often, special items are allowed only on strength 2. If there is only one slot available, will be strength 2. If there are two slots available, they will be strength 2 and 1. If there are three slots available, they will be strength 2, 1 and 0. The strength order is randomized at every roll, so one time it could be 2-1-0, another time it could be 0-2-1, etc.

If the parameter `isSpecialFixed` is set to true, then only for the slot of strength 2 the special item is automatically taken.

In all the other cases, the function `ItemOwnerSetting::CalcRandomItem` it's called in order to determine the result for the slot.

The function `ItemOwnerSetting::CalcRandomItem` first checks the property `mSlotRotateCount` of the `ItemOwner` object. If it's `!= 1` (so if it's not the first itembox passed) the function checks the `ItemOwnerSetting::ItemSpecialProbUpperMaster` object.

This object, when enabled, grants an extra chance to get the special item belonging to the driver (coinbox, boomerang, etc). Every time a result of strength 2 is rolled and the special item is not obtained, this object is enabled and an extra probability is stored and upped progressively at each missed opportunity, of 1 unit each time. Each unit grant a 1% chance of getting the special item.

If the object it's enabled and the kart rank is `>= 1` (0 is the 1st position, so it must be less than first) and it's `>` than the total number of karts in the race (so not in last position), a random number on base 100 it's thrown. If the result it's `<` than the amount of the value stored in the `ItemSpecialProbUpperMaster` object, then the special item is selected. Even if it's selected, the game anyway checks if the item could be obtained, verifying:

- if there aren't too many items of its kind already in the game
- if the right amount of time it's passed since the race start
- if the right amount of time it's passed since the last time the item was got
- if the event allows that kind of item
- if the item has the flag `mCanHoldOnlyOne` enabled and the kart entity doesn't have a duplicate

If `ItemOwnerSetting::CalcRandomItem` is not enabled or something fail in the above checks, then the game tries to get an item through the probability tables. In order to do that, the function `ItemProbabilityTable::CalcRandomItem` it's called.

This function purpose it's to recover the right `itemProbabilityTableOfRank` object. In order to do that, this function first checks if the roll is the first in the race and if the `mProbabilityOfFirstRotateInThisRace` object is enabled (as a reminder, this object is enabled only in PVP). If so, the corresponding strength table is got.

Otherwise, the game maps the rank index in order to obtain the right one for querying the probability tables.

As for the fevers, first the `skillSetting::mAddedRank_LowerRankSlotTable` is checked. If it's enabled, the rank it's increased by that setting.

After that, the `ItemRankMapData::mRankMap` object is checked and used to get the mapped rank index. As a reminder, this object is enabled only in PVP, cause the race could have less than 8 karts.

Once the normalized rank index is obtained, the proper `ItemProbabilityTableOfRank` object is retrieved and the function `ItemProbabilityTableOfRank::CalcRandomItem` it's called.

The first thing this function does is to recalc the probability tables in order to generate the `mEffectiveProbTable` object using the `cNotZeroProbTable` and `cNotZeroProbTableAddtional` objects. This is necessary because even if every result in the `cNotZeroProbTable*` objects it's valid for the kart, not every result it's valid in a specific moment of the race, because of the objects limitations.

Each element of the `cNotZeroProbTable*` objects it's processed and added to the `mEffectiveProbTable` object if comply to the limit reported above for the special item.

If the item is allowed, then the probability is adjusted by a rate and added to the table.

The rate is taken from the `ItemOwnerSetting::mSlotResultSetting::ProbRatio` property. In my test this property was always 1, but there is no guarantee that it cannot be changed by piece of code I didn't find yet.

When processing Thunder and Blue shell, only in case the kart it's a CPU, the race it's a normal one and the `ScriptableItemParamGeneral::mAdjustmentDifficulty::mIsReduceRateProbEmenyStrongItemByFeverNum` property is set, the `ItemParam::mReduceRateProbEmenyStrongItemOnFever*` properties are taken depending on the numbers of fever already got by the player, and further multiplied by the `ProbRatio`. This is done in order to reduce the probabilities for the bots to get Thunders and Blue shells the more frenzy the player have got. This parameters are actually set to:

- 0.9 with 1 frenzy
- 0.7 with 2 frenzy
- 0.1 with 3 frenzy

The final `ProbAdjustRate` is then multiplied by the `baseProbability` from the `ItemProbabilityTableOfRank` object.

If the overall probability is ≥ 1 , the result it's added to the `mEffectiveProbTable` object. And the `mProbabilitySum` property it's increased. This property will be used as a base for the throw.

The same is done for the `cNotZeroProbTableAddtional` table, so if there are additional probabilities available, are added at the end of the list.

After the `mEffectiveProbTable` object is built, the game throw a random number with the `mProbabilitySum` value as base in order to determine the result. The random value must be \leq of the progressive `mProbability` sum.

For instance, just imagine the probabilities in `mEffectiveProbTable` are:

- | | |
|---------------------|----|
| • Banana | 30 |
| • Green Shell | 15 |
| • Coin | 55 |
| • Coin (Additional) | 18 |

The `mProbabilitySum` value will be: $30+15+55+18 = 118$

The game throws a random floating point number between 0 and 118.

Let's say I get 50. In order to understand which item I will get, I have to progressively sum up the probabilities. So it will be:

- | | |
|---------------------|------------------------|
| • Banana | from 0 to 30 |
| • Green Shell | from 30.000001 to 45 |
| • Coin | from 45.000001 to 100 |
| • Coin (additional) | from 100.000001 to 118 |

Since the random was 50, I got a coin.

After the result is got, the function `ItemInnerUtil::ReplaceRandom` it's called.

The porpouse of this function is to change the result obtained with 3 different patterns, but only if the `ItemSlotResultParam::mRandomReplaceType` it's enabled.

The possible `mRandomReplaceType` values and patterns are:

- `RandomFromHigherRankKarts` - This pattern is valid only if the kart rank is > 0 (not first), and tries to change the item hold by other karts in more advanced positions if they got their special item and the special item is the same for the current kart and the `ItemParam::mRandomReplaceType` is set and is different from `RandomFromHigherRankKarts`.
So for instance, if the kart rolling the items have a driver with the boomerang skill, and one of the kart in front of him have the boomerang, the game replace that item with and item choosen from the `ItemGlobalFields::mTmpSlotResults_ReplaceRandom` list. I suppose this is done in order to free the item count for the kart.
- `RandomFromAllSpecial` - if the list `ItemGlobalFields::mActiveSlotResultsSpecial` have some entry, it's cycled and checked. If for every entry the `ItemParam::mRandomReplaceType` is set and is different from `RandomFromHigherRankKarts`, and the item can be got (with all the usual checks), the result is replaced with the one from the list.
- `RandomFromList` - if the `ItemParam::mReplacedSlotResult` list have some entry it's cycled, and if the item can be got (with all the usual checks), the result is replaced with the one from the list.

In all this cases, the value it's not changed right after, but added to a temporary list (`Game.ItemGlobalFields::mTmpSlotResults_ReplaceRandom`).

If no possible replacements are found with the pattern above and the temporary list is empty, the original result is returned.

Otherwise, the temporary list is cycled and checked against the

`Game.ItemSlotResultParam::mReplacedSlotResultProbability` array in order to obtain the sum of all the probabilities for the replacement (cause for every replacement there is a fixed probability). The this sum is used to throw a random number and get one item from the temporary list.

If for some reason the sum is equal to 0, but the list have some items inside it, a random number with the temporary list length as base is thrown, and an item it's returned from the list.

If for some reason noone of the above patterns works, the result is set to 0.

If for this reason or for some other reason a result cannot still be obtained (for instance the probability list it's empty because of the item limitations), the function `ItemOwner::CalcPinchItem` it's called.

This function retrieve the proper list of results from the `Game.ItemDefine` object: `cPinchItemsP2P` for the pvp, `cPinchItems` for normal races. Those are "safe" results that could be obtained no matter what.

At the moment, there are just 2 items in the list: the coin and the mushroom, in this order for the normal races, in reverse order for the PVP races.

If the slots are less than 3, or there is only one item in the pinch list, the first item in the pinch list is used.

If the slots are 3 and there is more than one item in the pinch list, then the slots are cycled and tested again the corrspondent item in the pinch list. If it's different, the item from the pinch list is returned.

When finally the result is determined, the control goes back to `startRotatingItemSlotImpl`.

The result is added to the kart `ItemPossessList`, and if it's a special item `mSpecialProbUpper` it's resetted, otherwise it's incremented (only for the slot of strength 2).

APPENDIXES

APPENDIX A – TABLE STRUCTURES

A1 – SCRIPTABLE SLOT AND FEVER

```
ScriptableRaceItemSlotTable
    ItemProbabillyOfKartDataSet mProbTable // EitemTableType: 0 = Master (player) 1 = Cpu
    [
        ItemProbabillyOfKartData mTable // The item slot, 0-2; 0 = the weakest, 2 = the strongest
        [
            ItemProbabillyOfRankData mTableOfRank // race position, 0-7
            [
                INT32 mProbability[] // base probability of getting a specific item
                INT32 mAdditionalProbabilly[] // applied only when possessing a matching glider skill *
            ]
        ]
    ]

GameItemFeverProbabilityOfRankData mFeverTableMaster // race position, 0-7
[
    GameItemFeverProbabilityOfLapData mProbTable // lap, 0 = not-last (regardless of the laps amount), 1 = last
    [
        INT32 mFeverProb // base probability to trigger a fever (percent)
        INT32 mProbOfEachItem[] // probability to get a specific kind of fever if a fever is triggered
        INT32 mAdditionalProbOfEachItem[] // modifier for the above setting.
    ]
]

ItemFeverProbabilityOfRankData mFeverTableCpu[]

ItemRankMapData mRankMapPvP
    BOOL mIsEnabled
    RankMapDataOfKartNum mRankMap
    [
        INT32 mRankMap[]
    ]
    BOOL mIsEnabledFeverProb
    FeverProbDataOfLap mFeverProbOfLap
    [
        FeverProbDataOfKartNum mFeverProb
        [
            INT32 mFeverProbPercent[]
        ]
    ]
    BOOL mIsEnabledFirstSlotRotate
    ItemProbabillyOfRankData mFirstProbTable[
        INT32 mProbability[]
        INT32 mAdditionalProbabilly[]
    ]

ItemProbabilityBakedData mBakedData
    Boolean mIsProbAllZeroMaster[] // the kind of item as index
```

* no special items have extra probabilities, except the Triple Mushroom

A2 – CUSTOMIZED SLOT TABLE

```
ItemProbabilityTable [ // SLOT

    INT32 cKartIndex // a numeric index of the Kart owning the table (0-7)

    ItemProbabilityTableOfRank mProbability // RANK as index
    [
        INT32 cKartIndex // same as above
        ItemProbabililyOfRankData cDataRef // a copy of the original data used to generate the table
        System.Collections.Generic.List<ItemProbResultSet> cNotZeroProbTable // filtered list of valid results
            EItemSlotResult mResult // the result type
            INT32 mProbability // the probability to get that specific result

        System.Collections.Generic.List<ItemProbResultSet> cNotZeroProbTableAdditional /* filtered list of additional
                                                                                          probabilities */

        System.Collections.Generic.List<ItemProbResultSet> mEffectiveProbTable /* further filtered list filled at
                                                                                      every item box */

        INT32 mProbabilitySum // the sum of all the probabilities in mEffectiveProbTable
    ]

    ItemRankMapData mRankMap // a copy of the mRankMapPvP property of the scriptable table

    ItemProbabilityTableOfRank mProbabilityOfFirstRotateInThisRace[] /* RANK as index - as above, but just for the first
                                                                                          box passed in PVP */
]
```

A3 – CUSTOMIZED FEVER TABLE

```
ItemFeverProbabilityTable
    ItemFeverProbabilityOfRank mFeverTable // RANK (0-7)
    [
        ItemFeverProbabilityOfLap mProbOfLap // LAP: 0 all but last, 1 last
        [
            INT32 cKartIndex
            INT32 mFeverProbability // the base probability to get a fever
            ItemSlotResultKeyDictionary<INT32> cEffectiveTable // INT32 it's the value type, the key it's EItemSlotResult
            ItemSlotResultKeyDictionary<INT32> cAdditionalEffectiveTable // same as above
            INT32 mProbSum
            ItemRankMapData mRankMapData
        ]
    ]
    ItemRankMapData mRankMap
```

APPENDIX B – TABLE DUMPS

Please refer to the attached spreadsheet for the table dumps.
Each sheet is numbered accordingly to the B index references:

B1 – ScriptableRaceItemSlotTable MASTER (normal races)

B2 – ScriptableRaceItemSlotTable BOSS

B3 – ScriptableRaceItemSlotTable MULTIPLAYER NORMAL SLOTS

B4 – ScriptableRaceItemSlotTable MULTIPLAYER 2 SLOTS

B5 – Relationship between Skills and EltemSlotResult

B6 – EltemSlotResult list and relationship with the Items