Data structures - Hearts of Iron 4 Wiki

This is a community maintained wiki. If you spot a mistake then you are welcome to fix it.

Within Hearts of Iron IV, there are several types of data structures, used to store data to be used for later. This can be used in-game as a way to check if some effect block was executed or not, as a counter of sorts, or something else.

Constants[编辑 | 编辑源代码]

Constants serve as a way to use the same numeric value with a reference. The constants are marked with the `@` symbol when defining or using them. An example definition is the following:

@CONSTANT = 10

This can later be used in the file as, for example, `cost = @CONSTANT`. These definitions can be in any point of the file.

Constants are available in the vast majority of game files, including most, if not all, *.txt and *.gui files. Constants can be used as a way to link several values to be the same if they can easily be changed at any point in the development.
For example, if there is a large decision system where every decision is intended to have the same cost, then these constants may be used if the political power cost gets changed mid-development for better balance.
Another example would be a scripted GUI container: if a large multitude of elements are to be at the same X or Y position, it might be worth it to link them to be a constant in case their position could be changed for better appearance or to fit in another GUI element.

Defines[编辑 | 编辑源代码]

> *Main article: Defines*

Defines are a particular type of constant that are used within internal calculations only, defined within /Hearts of Iron IV/common/defines/*.lua files. These cannot be referenced in any other files and are only used within the internal, unchangeable code.

Some of the things included in the defines are that events by default stay for 13 days before auto-selecting the first option[1] or that the ai_will_do value is 50% higher for focuses which were unlocked by the just-completed focus[2]. Most defines are numerical in nature, but some can be strings or blocks, such as the end date at which the game over screen appears[3] or the list of experience need for a division to progress to the next level[4].
The list of defines is vast and touches on every mechanic in the game, so if something can't be changed within the mechanic's respective files, it may be worth to check defines. Defines don't include the Modifiers#Static modifiers, which can serve a similar purpose but with modifier blocks.

As defines are Lua code rather than a PDXscript-interpreted file, any Lua code can go in there, however modules allowing to go out of this directory are not available. As a consequence of this, **there is no need to copy the entire file to edit it**. Defines, including graphical defines, are merely a Lua-contained array, and it is possible to modify a single member of the array in Lua using, for example, `NDefines.NGame.START_DATE = "1936.1.2.12"`. Each of these lines is contained on a separate line and there are no commas separating them, such as the following:

NDefines.NGame.START_DATE = "1936.1.2.12"
NDefines.NGraphics.COUNTRY_FLAG_TEX_MAX_SIZE = 2048

This goes into a separate file set to be evaluated after the base game defines, which is done by making the filename be later in the alphabetical order (decided by Unicode character IDs). As most Unicode characters (including every uppercase or lowercase letter) have larger IDs than the code used for the number 0, the filename is almost irrelevant.
In the base game files, the graphical defines are set to be merged into NDefines in the last line of the base game file. However, since Lua does not create copies of tables by default, the base game's NDefines contains pointers to the actual elements, which are contained within NDefines_Graphics regardless. For this reason, modifying either NDefines or NDefines_Graphics works for changing graphical defines.
A mod should **never** contain the 00_defines.lua and 00_graphics.lua files within of itself: these files are commonly changed even in minor game updates, and having a define missing from a file results in the mod being unstable, potentially having a crash on startup.

Flags[编辑 | 编辑源代码]

In this case, flags are used in the sense of a computer variable that is used to set data to and later read it from, commonly but not necessarily of a boolean value (i.e. either 0 or 1). For editing the flags used for representing countries in-game, see Country creation for the starting flags and cosmetic tags for changing them in the middle of the game.

Flags within PDXScript are a 32-bit signed integer, ranging from -2147483648 to 2147483647. A flag can only be set to a specific value (Potentially for a certain amount of days), modified in an additive way by a certain value and read, no other operations can be done. Variables serve as a more operational alternative, with far more capacity for changing (Such as multiplication or non-integer values, reading game values, being able to set to other database entries rather than numbers, etc). Flags are still recommended to use as a boolean value, as these are more optimised than the variables, and so using variables will lead to an unnecessary increase in memory usage and will lead to the game running slower compared to flags.

Flags are divided into 4 scopes:

Global. These can be read from anywhere in the game, without the need for scoping.

Country. These are assigned to individual countries, being impossible to assign or read elsewhere. A country flag can have different values depending on which country it's checked in.

State. These are assigned to individual states, being impossible to assign or read elsewhere. A state flag can have different values depending on which state it's checked in.

Character. These are assigned to individual characters, being impossible to assign or read elsewhere. A character flag can have different values depending on which character it's checked in.

In general, the same principles apply to them with little changes. While unit leader flags exist, these are marked as deprecated and are likely to be removed within the next updates.

A flag can be set in *any* effect block. There is no file where flags need to be defined, they will exist if set and stop existing when cleared.
The simplest formatting for setting a flag is `set_country_flag = my_flag_name`. This will create the country flag (If it did not exist otherwise) and set it to be a value of 1. *This has no tooltip* and so it will be invisible to the player. custom_effect_tooltip may be used to tell the player that something will be done, if wanting to keep them in the know.
Afterwards, that flag can be checked with `has_country_flag = my_flag_name` as a trigger. This is set to be true if the flag was set, regardless of which value it has. This will create a tooltip using the flag's name as the localisation key. As such, `my_flag_name: "Has my flag"` will make that appear when the country flag is checked.
If a flag is no longer useful, it can be cleared by using `clr_country_flag = my_flag_name`, which'll mark it as no longer being set. This also has no tooltip and the player will not be aware unless specifically told with the custom effect tooltip that the flag was cleared.

The exact same principles apply with global, state, and character flags. The only differences are in the used effects/triggers (In which case the word `country` is swapped with the type's name like `set_global_flag`, `has_state_flag`, or `clr_character_flag`) scopes in which they can be set and where they can be checked. For global flags, it can be done in any scope rather than the 'global' scope: it can be set in either country, state, character, or rare other scopes.

Flags can also be targeted towards a certain country. This is usually done by appending the `@TAG` to the name of the flag, though it can be placed anywhere within the tag name except for the first character. For example, `set_country_flag = my_targeted_flag_@PREV` will set it to be targeted towards the country represented by PREV, which we'll take as TAG. *This just appends the PREV's tag to the flag's name*. After this, `has_country_flag = my_targeted_flag_TAG` will come up as true for the same country it was set in. This can be used to create dynamic flags without relying on meta effects. You cannot have more than one target specifier in a single tag, You can only use @ROOT, @PREV, @FROM, and @THIS, even @PREV.PREV does not work. Both of these restrictions do not apply to targeted variables, so those may be a more flexible alternative if you need.

Example[编辑 | 编辑源代码]

The most common use of flags is to simply to mark a certain event (in the generic sense of a word, not necessarily the in-game event mechanic) as having occurred, such as an election, an invasion, a diplomatic deal, et cetera. For example, here's an event that starts off an election campaign in the country:

```
country_event = {
  id = algeria_election.0      # Namespace is assumed to exist.
  title = algeria_election.0.t
  desc = algeria_election.0.desc

  is_triggered_only = yes

  immediate = {
    set_country_flag = ALG_1936_elect_campaign  # Since set_country_flag has no tooltip, no need to hide it.
  }

  option = {
    < ... >                # Irrelevant what's in the option
  }
}
```

Flags provide a way to mark that this event has been received, in this case set within the immediate effect block of the event.

In order to utilise this, the `has_country_flag` trigger will have to be used similarly to this example decision:

```
ALG_1936_campaign_category = {      # Decision category is assumed to exist and contain 'allowed = { tag = ALG }'
  ALG_1936_select_candidate_A = {
    available = {
      has_country_flag = ALG_1936_elect_campaign
    }

    icon = ALG_elect
    cost = 50
    fire_only_once = yes

    complete_effect = {
      country_event = algeria_election.1
    }
  }
}
```

This will work fine, however, as is, the player will see either a ✔ check mark or a ✖ cross next to 'ALG_1936_elect_campaign', with no indication what that truly means. In order to correct that, a localisation entry can be added to any working localisation file (i.e. is an *.yml file within the /Hearts of Iron IV/localisation/ folder or any of its subfolders, has the internal name of the language in the end of filename before the extension, and is encoded in UTF-8 with the byte order mark present):

```
 ALG_1936_elect_campaign: "The 1936 election campaign is currently ongoing"
```

In order to clear the flag when needed, the `clr_country_flag` effect would be used as such:

```
country_event = {
  id = algeria_election.100      # Namespace is assumed to exist.
  title = algeria_election.100.t
  desc = algeria_election.100.desc

  is_triggered_only = yes
  option = {
    name = algeria_election.100.a
    trigger = {
      has_country_flag = ALG_candidate_A_won
    }
    clr_country_flag = ALG_1936_elect_campaign
    < ... >                # Irrelevant what else is in the option
  }
  option = {
    name = algeria_election.100.b
    trigger = {
      has_country_flag = ALG_candidate_B_won
    }
    clr_country_flag = ALG_1936_elect_campaign
    < ... >                # Irrelevant what else is in the option
  }
  < ... >                  # Other potential options
}
```

Note that neither of the effects that modify country flags have tooltips, so a custom effect tooltip can be used to notify the player in some cases.
Everything is the exact same for flags of other scope types.

Complexities[编辑 | 编辑源代码]

This will cover the non-boolean usage of flags. Variables usually do this job better, but this still has some usages.

The `set_<type>_flag` effect can also be expanded with the optional `days` and `value` arguments. This will appear like the following:

```
set_state_flag = {
  flag = my_state_flag
  days = 50
  value = 3
}
```

This in particular will set the flag my_state_flag to the state to be equal to 3 for 50 days. After these 50 days end, the flag will be cleared completely, reset to 0. If days are left out, the flag will be set to

that value forever, while if the value is left out it'll be set to 1.

And the trigger if a flag is set can also be expanded as such:

```
has_global_flag = {
  flag = my_global_flag
  value > 2
  days > 30
  date < "1938.1.1"
}
```

While the flag and the value arguments are self-explanatory, `days` checks how many days it was *since the flag was set* (Not how many are remaining), while the `date` checks the date at which the flag is set. Every argument aside from the `flag` one is optional, just assumed to be always true if not set. This trigger will still always fail if the flag was never set or if it was cleared.

Additionally, it is possible to *modify* a flag as such:

```
modify_character_flag = {
  flag = my_character_flag
  value = 4
}
```

This *adds* the amount to the flag, with negatives being possible. However, there's one catch: **this does nothing if the flag is not set**, having a value of 0.

Each of the preceding examples can be used for any flag type.

Event targets[编辑 | 编辑源代码]

Event targets serve as a primitive way to set a pointer to a scope (country, state, character) to be referred to later. Any single event target can only point towards a single scope at a time, however a scope may have several event targets at the same time. Similarly to flags, there is no set folder for event targets, they can be assigned in any event block.
There are two types of event targets: regular and global. The difference between them is purely when they get cleared.

A scope can be checked if it has an event target assigned using `has_event_target = name_of_event_target`.

Global event targets[编辑 | 编辑源代码]

Global event targets can be declared using `save_global_event_target_as = my_event_target` as an effect. This will create the global event target with the name of `my_event_target` and it will last forever. Afterwards, it can be referred to as `event_target:my_event_target`, both as a scope and as a target.

If `save_global_event_target_as = my_event_target` is used when the event target is already set, the event target will change scopes to the one where the effect is used.
Additionally, `clear_global_event_target = my_event_target` (to be used in any scope, not necessarily the event target) will clear the specified global event target entirely, while `clear_global_event_targets = ROOT` will clear every global event target from the specified scope.

For example, by using start_civil_war as an effect block, the following will assign the TAG_democratic_side event target to the revolter:

```
start_civil_war = {
  size = 0.5
  ideology = democratic
  save_global_event_target_as = TAG_democratic_side
}
```

After this, TAG_democratic_side event target will refer to the democratic side. For example, this code later in another effect block can be used to transfer a state and a part of the armies to the event target:

```
event_target:TAG_democratic_side = {
  transfer_state = 123
}
transfer_units_fraction = {
  target = event_target:TAG_democratic_side
  size = 0.2
}
```

It would make sense to clear the event target after it no longer exist, such as by using the on_civil_war_end on action.

When using the event target as a scope for a namespace in localisation, the `event_target:` prefix is not needed, instead it'll look like `[TAG_democratic_side.GetName]`.

Regular event targets[编辑 | 编辑源代码]

Regular event targets can be declared using `save_event_target_as = my_event_target` as an effect. The difference between regular and global event targets is in when they get cleared. While global event targets are used forever and can only be cleared manually, regular event targets only last as long as the effect block does (carrying over into events fired within as well), clearing automatically afterwards, and there is no way to clear them manually.

For example, this in an effect block will save a random owned and controlled state as the event target `target_state` and then fire the event mod_event.0 in 10 days:

```
random_owned_controlled_state = {
  save_event_target_as = target_state
}
country_event = { id = mod_event.0 days = 10 }
```

Assuming no other events, this event target will only exist within this effect block and the event mod_event.0 (Where it can be used in its corresponding trigger or effect blocks). If the event mod_event.0 fires another event within a corresponding effect block, this event target will exist in that event as well, meaning that this carries over to the entire mod_event.0-starting event chain. For example, the generic event generic.23 ([From.GetNameDefCap] Demands Control of [demanded_prov_target.GetName]) uses this with demanded_prov_target as the event target. In order to make ROOT demand the state 123 from TAG using this, this effect block will be used:

```
123 = {
  save_event_target_as = demanded_prov_target
}
TAG = {
  country_event = generic.23
}
```

Event targets can be used in *most* effect blocks, they are called event targets only because they carry over to events. However, there are a few effect blocks, such as scripted diplomatic actions, where event targets are impossible to set.

Country tag aliases[编辑 | 编辑源代码]

Country tag aliases are somewhat similar to event targets: they are used as pointers towards a specific scope, a single alias can point towards a single scope, and a single scope can have several aliases. However, there are differences. Primarily, country tag aliases can only point towards countries (as per the name) and they are set and cleared entirely automatically without any manual involvement. A country tag alias can point towards a non-existing country.
Usually, these are used to ease referencing to a specific side of a civil war by using country flags set when creating the revolter within either the start_civil_war or the create_dynamic_country effects. Both serve as effect blocks running the effects on the newly-executed country.

Country tag aliases are stored within /Hearts of Iron IV/common/country_tag_aliases/*.txt files. Each country tag alias is stored as a separate block within the file, with the name of the block being the name of the alias. Additionally, the country tag aliases serve as a trigger block, so any trigger can be used within. These are the arguments that can be used within:

`variable = global.var_name` - This will make sure that the country tag alias is equivalent to the country that the specified variable points to, in this case `var_name` assigned to the global scope.

`event_target = my_event_target` - This will make sure that the country tag alias is equivalent to the country that the specified **regular** event target points to, in this case `my_event_target`.

`global_event_target = my_event_target` - This will make sure that the country tag alias is equivalent to the country that the specified **global** event target points to, in this case `my_event_target`.

`original_tag = TAG` - This puts every country that originates from TAG (created with start_civil_war or create_dynamic_country) as a possible target for the alias. In this case, further specification is needed with triggers. This is equivalent to the original_tag trigger, but this is treated as an argument here as it does make certain errors go away.

`targets = { ABC DEF GHI }` - This is a whitespace-separated list of country tags that the country tag alias can select. In this case, further specification is needed with triggers.

`target_array = global.my_array` - This puts every country that's within the specified array (`my_array` in the global scope in this case) as a possible target for the country tag alias. In this case, further specification is needed with triggers.

`country_score = { ... }` - This is a MTTH block evaluated for each country. The country which has the highest total score will be picked for the country tag alias, provided the conditions imposed by every other argument are met.

`fallback = TAG` - This is what the country tag alias will point to if no country fulfills the conditions specified within triggers or other arguments.

Here are some examples of country tag aliases:

```
MOD = {
  targets = { ABC DEF GHI }
  has_country_flag = my_flag
  country_score = {
    base = 1
    modifier = {
      add = 2
      tag = ABC
    }
    modifier = {
      add = 1
      tag = DEF
    }
  }
  fallback = ABC  # This will point to ABC, DEF, or GHI if they have the specified country flags. ABC is prioritised over DEF, which is prioritised over GHI. If neither have the country flag, defaults to ABC.
}
ARB = {
  target_array = global.arab_countries
  has_cosmetic_tag = ARABIA_UNITED    # This will point to any country in the global.arab_countries array if they have the ARABIA_UNITED cosmetic tag, provided they exist.
  exists = yes
}
```

Variables[编辑 | 编辑源代码]

A variable within Hearts of Iron IV is a signed 32-bit fixed point number with scaling of {\displaystyle {\frac {1}{1000}}}. As such, it ranges from -2 147 483.648 to 2 147 483.647. Adding further amounts above or below the variable will result in an overflow, causing the value go from the maximum to the minimum or vice-versa.

Variables are not stored within any particular file and can be created dynamically. Each database object (such as countries, states, ideas, or even equipment types) has a unique internal numeric ID, which can be assigned to a variable. For example, setting a variable's value to a country tag will make it take on the ID of the tag, and then the variable will be taken as a country when used as a target or as a scope. For tokenizable database types, a `token:` prefix is necessary when assigning a literal, for example: `var_name = token:some_idea_name`.

When using a variable as a scope, the `var:` prefix is needed, however in some cases the prefix is optional: most notably when using it as the target of an effect or trigger. For example, in `transfer_state = var:my_state_variable`, the `var:` can be omitted with no difference (For some cases, there will be logged errors, yet it will work in-game), but in `var:my_country_variable = { transfer_state = 123 }` the `var:` is *mandatory*. Despite functioning similarly to a scope, the `var:` prefix is optional when a variable is used as a weight inside of random_list. Same as with event targets, using variables in localisation for namespaces does not require the `var:` prefix, such as `[var_name.GetName]`. A good rule of thumb is that when passing a variable in a place where a raw token (rather than, say, a quoted string, a number, or a numeric state ID) is acceptable, it must usually be prefixed with `var:` so that the parser can distinguish whether the token being passed is meant to reference a database object or a variable. If `var:` is absent in a location where it should be present, the code may work, but the game will often complain in the error log. In general, it is best to err on the side of caution in this regard, as what may result in a merely-complaining error in one context, might result in an effect-nullifying, or worse, trigger-falsifying, error in another context. However, `var:` is always optional in variable and array manipulation operators (including `check_variable`), and so it can be elided in those operators for readability's sake.

Within their names, variables can have any word character, i.e. the 26 English letters (Either lowercase or uppercase), underscores, and the decimal digits from 0 to 9. However, a variable name cannot *begin* with a decimal digit, so a name like `123_my_variable` is invalid. This requirement common in many programming languages, and is implemented here for the same reason that the `var:` prefix exists: so that the parser can determine what syntactic type a token will be from the first (few) characters, and therefore doesn't have to worry about backtracking as much. In addition, it is recommended to avoid any names by variables that are already used for something else. For example, naming a variable `capital` can prevent any subsequent usage of the game variable with the same name within the internal code. A common naming convention to avoid overlap is to add a prefix signifying where it's used, such as the country's tag (`POL_capital`) or the name of the mechanic (Usually shortened like `PC_capital`).

Variable types[编辑 | 编辑源代码]

There are two types of variables: regular and temporary. There are the following differences between the two:

Different effects are used to set and modify regular and temporary variables. Typically, the difference is just in changing `variable` to `temp_variable`.

Temporary variables only exist within the effect or trigger block where they're defined, similarly to regular event targets (although without carrying over to events). This does not mean they don't survive outside of the scope where they're defined, but the '*entire* effect block, such as a focus reward or an event option. Meanwhile, regular variables exist forever. Note that this, however, does not always function properly in regards to scripted effects or triggers: temporary variables defined within of them may not still exist outside of them within the same effect/trigger block. However, a temporary variable defined before the scripted effect or trigger's usage will always carry on into it and get modified properly within.

Due to the previous point, **temporary variables can be defined and modified in trigger blocks** as well as in effect blocks. This enables them to be computed as-needed in scripted triggers

called from the `trigger` block of scripted localizations. Meanwhile, regular variables are restricted to only being defined in effect blocks.

Temporary variables are unscoped: they will remain the same regardless of in which scope they're defined and where they're checked. Meanwhile, regular variables are restricted to scopes, so that the variable has different values depending on which scope it's checked in, such as how different countries may have different stability values.

Scoping[编辑 | 编辑源代码**]**

Scoping only exists for regular variables. They can be assigned to countries, states, characters, and to the global scope.
The scopes are assigned by prepending any dual scope that can be used as a target before the variable's name, separated with a dot, such as `PREV.PREV.var_name` being the value of `var_name` as checked for `PREV.PREV` or `event_target:my_event_target.var_name`. This also includes `global` in order to assign it globally. Other variables, if pointing to a scope, can also be used, but the `var:` is unneeded. Variables can also be chained using the `:` separator (rather than the usual dot), such as `ABC.capital:var_name` being the variable `var_name` as it's assigned to the capital state of the country ABC, where `capital` is a built-in variable pointing to the capital state of the country. Note that the scope specifier preceding `capital` is separated from it by a `.` since it is a static (that is, non-variable) scope. The way to intuit this syntax is that the whole `ABC.capital` clause forms a variable scope specifier, in which `var_name` is accessed.

If there is no scope specified, then the current scope (or `THIS`) is implied. For example `set_variable = { var_name = 10 }` will create a variable with the name of `var_name` within the scope it's executed in (country, state, character, etc). However, in some cases where the implicit scope differs from `THIS`, such as state events and certain types of scripted GUI, using the implicit scope can cause in-game confusion on whether it refers to the default scope or ROOT. In these cases, it's better to specify `THIS.var_name` for the current scope or `ROOT.var_name` for ROOT. **It is necessary to put in 'global' to scope into global**, it is never assumed by default.

Here are some examples of scoped variable references:

```
GER.var_name                    # Scopes to GER
123.var_name                    # Scopes to state 123
global.var_name                 # Scopes to global
province_controllers^1234:capital:var_name  # Scopes to the capital state of the country that controls the province 1234.
```

Trigger usage[编辑 | 编辑源代码**]**

Only temporary variables can be modified within a trigger block. Regular variables can only be used in effects, while temporary can be used in both triggers and effects.

However, there's one aspect of triggers that's important to know: *when* each scope stops evaluating. If a scope stops evaluating, then the variable will not be modified anymore. In general, trigger scopes that can select multiple targets are divided into two groups: `all_<scope type>` and `any_<scope type>`. The first group, requiring all targets to meet triggers, evaluates each target in order and stops when it encounters one where the triggers aren't met. The second group, requiring any target to meet triggers, evaluates each target and stops when it encounters a true one.
Empty trigger blocks are considered true by default, so one would instantly stop the validation of the second group (`any_` triggers). For this reason, since scope limits are not supported in trigger scopes, then the proper usage for modifying temp variables would be an if statement that modifies the variable if the limit is met with no other triggers that can come up as true or false coming up.

Localisation[编辑 | 编辑源代码**]**

> *Parts of this section are transcluded from Localisation#Formatting variables*

In order to display the value of a variable in localisation, `[?var_name]` is used, with the question mark signifying that it's a variable. **The question mark is mandatory** in order to distinguish the variable from the event targets. For example, such a localisation value will work: `var_value_tt: "There are [?num_of_dogs] dog(s) in the country"`
By default, the scope where this is used gets assumed. Scoping works the same way as it does with variables regularly: `GER_var_value_tt: "There are [?GER.num_of_dogs] dog(s) in [GER.GetNameWithFlag]"`

If the variable points to a certain address with more localised text, such as a country, namespaces can be used in the same manner as with a regular tag or event target: `most_dogs_tt: "[?most_dogs.GetNameDefCap] has the most dogs out of any country."` Question marks are still mandatory to include in this case.

Variables have a special way to more easily apply the colouring characters, as well as special formatting characters. These are applied after a pipe placed at the end of the variable's name, such as `[?my_variable|R]` that will turn the colour of the variable `my_variable` red. The exact list of formatting characters that are restricted to variables only are the following:

Code	Effect
%	Converts the variable to percentage, multiplying by 100 and appending a %.
%%	Appends a percentage to the end of the variable without multiplying by 100.
*	Converts the variable to SI units—appends "K" or "M" and divides the variable appropriately, such as 65 536 becoming 65.53K and 1 500 000 becoming 1.50M. Displays 2 decimals after the dot by default.
^	Same as *.
=	Prefixes the variable with + if the value is positive or - if it is negative.
0..3	Controls the number of decimals to display. Due to the nature of the game's variables, there are no more than 3 decimals that can be used. Optionally possible to prepend a dot to the beginning.
+	Colours the variable green if positive, yellow if zero, red if negative.
-	Colours the variable red if positive, yellow if zero, green if negative.

Here are some examples of formatting characters in usage:

```
loc_key: "Democratic party popularity: [?party_popularity@democracy|%G0]"
loc_key_2: "Modifier token's value: [?modifier@my_modifier|.1%%+]"
```

Within these examples, the first string depicts the current scope's democratic popularity as a percentage multiplied by 100 (%), in green (G), rounded to a whole number with 0 decimals (0). The second string displayes the `my_modifier` modifier token's value as a 'good' number (+ making it green if positive, red if negative), with a percentage sign appended in the end (%%) and rounded to a number with one decimal (.1).

There's also a more precise way to display *arguments* that modify variables. Some such arguments have an optional `tooltip = localisation_key` argument within. This tooltip is notable since the value of that localisation key allows using two strictly-internal variables marked with dollar signs: `$LEFT$` and `$RIGHT$`.
`$LEFT$` represents the variable's value before the operation, while `$RIGHT$` represents the variable's value after the operation. This can be used to let the player know the current and/or the potential value of the variable in the same localisation key without taking up more space with `custom_effect_tooltip`.

Here's an example of it being done:

```
add_to_variable = {
   var = num_dogs
   value = ENG.num_cats
   tooltip = add_cats_to_dogs_tt
}
```

This is the effect itself. Within localisation, this would be done:

```
 add_cat_to_dogs_tt: "Adds [ENG.GetAdjective] cats to our $LEFT$ dog(s) to get $RIGHT$ in total."
```

Additional usage[编辑 | 编辑源代码**]**

By default, an unset variable is equal to zero when checked. However, the game also provides a null coalescing operator for variables, used with a question mark. For example, `cost = var_name?100` within a decision will result in the cost of the decision being the same as var_name for the country. However, if that variable was never set or was cleared beforehand, then the decision will cost 100 Political Power instead of 0.

Similarly to flags, a variable can be set as targeted by appending the target with `@TAG`. However, there are differences: in variables this does not append the target's ID to the name but stores it as a separate variable, as well as it being possible to target far more than countries, but also, for example, states or ideology groups. A variable reference `var_name_@GER` would resolve to a variable named `var_name_GER`. If called in a scope where `PREV= { tag = ENG }`, a variable reference `var_name_@PREV` would resolve to a variable named `var_name_ENG`. This can even be applied to temporary variables. You can also target variables using the syntax `var_name_@var:target_var_name`, regardless of the type of their value: be it country, state, token, even a raw integer. Variables targeting tokenizable types such as ideology groups and ideas provides a powerful alternative to scoped variables for associating a set of variable fields together, enabling one to get around the fact that ideas, ideologies, etc. do not have their own scopes. Generally, when you target a token, it is pasted in as its underlying integer ID (a number in the ten-thousands, usually), but this generally works fine, because it's consistent; the only problem is it's not human-readable. **Note that it is not possible to use targeting in the name of an array, this will crash the game to desktop without a helpful error or crash message.** If you must have a dynamically-named array, an alternative is to use a meta-effect or meta-trigger to paste the appropriate token into the array name. In general, variable targeting is substantially more optimized than meta-effect/trigger token pasting, not to mention more readable and understandable, so should be preferred wherever feasible.

It is technically possible, under some circumstances, to use multiple target specifiers in one variable name, usually strung together back-to-back at the end of the name, like `var_name_@THIS_@PREV`. However, this is extremely inconsistent, and the exact rules that govern how it behaves are currently an open question.

Operators[编辑 | 编辑源代码]

The following is a list of variable-related effects and triggers. Variable-modifying effects have an equivalent for temporary variables, with `temp_variable` being used instead of `variable`, and these temporary variable operators are also valid triggers, as described above. Every operator can be used with variables that do not exist, assuming a value of 0 unless a null-coalescing operator is used.

Variable-related arguments: 折叠

Name	Parameters	Examples	Description	Notes

set_variable	`var = <variable>`
	The variable to modify or create.
	`value = <decimal>/<variable>`
	The value to set the variable to.
	`tooltip = localisation_key`Localisation used by the operation. Optional.
		set_variable = {
		var = my_variable
		value = 100
		tooltip = set_var_to_100_tt
		}
		set_temp_variable = { temp_var = ROOT.overlord }
			Sets a variable's value to the specified amount, creating it if not defined.	Shortened version exists with `set_variable = { <variable> = <value> }`.

set_variable_to_random	`var = <variable>`
	The variable to modify or create.
	`min = <decimal>`
	The minimum possible value, defaults to 0.
	`max = <decimal>`
	The maximum possible value, defaults to 1.
	`integer = <bool>`
	Sets if the variable *must* be an integer or if it can be decimal. Defaults to false.
		set_variable_to_random = {
		var = random_num
		max = 11
		integer = yes
		}
		set_temp_variable_to_random = my_var
			Sets a variable's value to the specified amount, creating it if not defined. The result will be greater than or equal than the minimum and strictly less than the maximum.	Shortened version exists with `set_variable_to_random = <variable>`, setting it to a decimal between 0 and 1. Can be used in triggers.

clear_variable	`<variable>` Variable to clear.	clear_variable = my_variable	Clears the value from the memory entirely.	Can only be used on regular variables.

check_variable	`var = <variable>`
	Variable to compare.
	`value = <decimal>/<variable>`
	The value to compare the variable to.
	`compare = <compare type>/<variable>`
	The comparison type. Variants are `equals`, `not_equals`, `less_than`, `less_than_or_equals`, `greater_than`, and `greater_than_or_equals`. Defaults to `greater_than_or_equals`.
		check_variable = {
		var = my_var
		value = 10
		compare = greater_than_or_equals
		}
		check_variable = { var_name > 10 }
			Compares the value of a variable. Used as a trigger.	Temporary and regular formatting use the exact same formatting. Shortened variant exists as `check_variable = { <variable> < <value> }`, where the operator in the middle is an equality sign (=), greater than sign (>), or a lesser than sign (<). The comparison type is always strict in this case: `check_variable = { var_name > 10 }` will come up as false if `var_name` is exactly 10.

has_variable	`<variable>` Variable to check.	has_variable = my_variable	Checks if the value was set, regardless of its amount. Used as a trigger.	A variable is considered set even if its value is 0, as long as it wasn't cleared.

add_to_variable	`var = <variable>`
	The variable to add to.
	`value = <decimal>/<variable>`
	The value to add to the variable.
	`tooltip = localisation_key`Localisation used by the operation. Optional.
		add_to_variable = {
		var = my_variable
		value = 100
		tooltip = add_100_to_var_tt
		}
		add_to_temp_variable = { temp_var = num_owned_states }
			Increases a variable's value by the specified amount, creating it if not defined.	Shortened version exists with `add_to_variable = { <variable> = <value> }`.

subtract_from_variable	`var = <variable>`
	The variable to subtract from.
	`value = <decimal>/<variable>`
	The value to subtract from the variable.
	`tooltip = localisation_key`Localisation used by the operation. Optional.
		subtract_from_variable = {
		var = my_variable
		value = -100
		tooltip = add_100_to_var_tt
		}
		subtract_from_temp_variable = { temp_var = num_owned_states }
			Increases a variable's value by the specified amount, creating it if not defined.	Shortened version exists with `subtract_from_variable = { <variable> = <value> }`. Equivalent to adding a negative amount.

multiply_variable	`var = <variable>`
	The variable to multiply.
	`value = <decimal>/<variable>`
	The value to multiply the variable by.
	`tooltip = localisation_key`Localisation used by the operation. Optional.
		multiply_variable = {
		var = my_variable
		value = 100
		tooltip = multiply_var_by_100_tt
		}
		multiply_temp_variable = { temp_var = num_owned_states }
			Multiplies a variable's value by the specified amount.	Shortened version exists with `multiply_variable = { <variable> = <value> }`.

divide_variable	`var = <variable>`
	The variable to divide.
	`value = <decimal>/<variable>`
	The value to divide the variable by.
	`tooltip = localisation_key`Localisation used by the operation. Optional.
		divide_variable = {
		var = my_variable
		value = 100
		tooltip = divide_var_by_100_tt
		}
		divide_temp_variable = { temp_var = num_owned_states }
			Divides a variable's value by the specified amount.	Shortened version exists with `divide_variable = { <variable> = <value> }`.

modulo_variable	`var = <variable>`
	The variable to modulo.
	`value = <decimal>/<variable>`
	The value to modulo the variable by.
	`tooltip = localisation_key`Localisation used by the operation. Optional.
		modulo_variable = {
		var = my_variable
		value = 50
		tooltip = get_modulo_of_var_by_50_tt
		}
		modulo_temp_variable = {
			Makes the variable become the remainder of Euclidean division of the variable by the specified value.	Shortened version exists with `modulo_variable = { <variable> = <value> }`.

Name	Parameters	Examples	Description	Notes
		temp_var = num_controlled_states }		
round_variable	<variable> The variable to round.	round_variable = my_variable round_temp_variable = temp	Rounds the variable towards the closest integer value.	If exactly between two integers (Such as 1.5), the larger option gets chosen.

clamp_variable	var = <variable>
	The variable to clamp.
	min = <decimal>/<variable>
	The minimum value of the variable after the clamp.
	max = <decimal>/<variable>
	The maximum value of the variable after the clamp.
		clamp_temp_variable = {
		var = my_var
		min = 0
		}
			Clamps the variable to ensure its value is between the two specified numbers, raising to the minimum if smaller or lowering to the maximum if larger.	Either min or max can be omitted, in which case it'll not be checked. Does nothing if the variable is already in the range between min and max. **This only changes the current value of the variable**, it can still go beyond the minimum or the maximum after the clamp.

Usage examples[编辑 | 编辑源代码**]**

A common usage for variables is to grant a more 'complex' effect or a trigger. This is best done with temporary variables, as they can be set in either triggers or effects and they're not bound to scopes making operations on them more simple.
For example, the following can take 0.1% of every controlled state's population and add it directly to the manpower pool if the country has more democratic party support than fascist and communist party support combined:

```
if = {
  limit = {
    set_temp_variable = { fascom_pop = party_popularity@fascism }
    add_to_temp_variable = { fascom_pop = party_popularity@communism }
    check_variable = { party_popularity@democratic > fascom_pop }  # Completely hidden from the player.
  }                                                # 'limit = {}' is always hidden so it doesn't matter here, but use custom_trigger_tooltip when wanting the player to be aware.
  hidden_effect = {  # Every controlled state can look weird to the player otherwise.
    set_temp_variable = { temp_manpower = 0 }  # Optional, can be omitted as variables are assumed to be 0 if unset.
    every_controlled_state = {
      add_to_temp_variable = { temp_manpower = state_population_k } # Executed in the state's scope, so state_population_k is of the state.
    }                                              # However, temp variables are unscoped so it doesn't matter where they're edited.
  }
  add_manpower = var:temp_manpower # 'add_manpower = temp_manpower' also works
}
```

Additionally, it is common for a variable to be a counter of something within a mechanic. For example, let CAN_beavers be a counter of beavers within the country.
When using variables in this sense, **it is better to use** scripted effects **to modify them**, in order to make changes to the variable easier. For example, a small gain in beavers could be the following:

```
CAN_beavers_small_add = {
  custom_effect_tooltip = CAN_beavers_small_add_tt  # Tooltip for the player to be aware.
  add_to_variable = { CAN_beavers = 100 } # Adds 100 beavers.
}
```

However, if 100 seems like too little, the scripted effect can be changed and it'll immediately apply the changes every single time that scripted effect was used without needing to find them manually. These effects may also share a single 'recalculating' scripted effect in common. This can be used to clamp the variable between a minimum and a maximum, to have an instant effect as soon as some variable value is reached, or to have another variable 'tied' to the value to be kept at an exact or a dynamic ratio. For example, these scripted effects can be used with the same CAN_beavers example:

```
CAN_beavers_calibrate = {
  clamp_variable = {  # The variable will be kept between these levels by calling this each time the variable is changed.
    var = CAN_beavers
    min = 0
    max = 2000
  }

  set_variable = { CAN_beavers_modifier = CAN_beavers }
  divide_variable = { CAN_beavers_modifier = 1000 }  # Another variable tied to the base value.

  if = {
    limit = {
      check_variable = { CAN_beavers = 0 }
    }
    country_event = CAN_beavers_extinction.0  # The event gets fired as soon as CAN_beavers hits 0.
  }
}

CAN_beavers_small_add = {  # Same as before, but with calling the previous effect
  custom_effect_tooltip = CAN_beavers_small_add_tt
  add_to_variable = { CAN_beavers = 100 }
  CAN_beavers_calibrate = yes
}

CAN_beavers_small_remove = { # Another example of a variable-modifying effect
  custom_effect_tooltip = CAN_beavers_small_remove_tt
  add_to_variable = { CAN_beavers = -100 }  # 'subtract_from_variable = { CAN_beavers = 100 }' also works.
  CAN_beavers_calibrate = yes
}
```

After creating these scripted effects, these can be used within any effect block to modify the variable at will. Showing its value in localisation to the player would also be helpful, such as within a decision category or in scripted GUI. **Variables cannot be used within idea modifiers directly,** dynamic modifiers can be used instead.

Remember that variables are not necessarily used as numbers only. For example, the simplest way to make the current ideology group to change to the one that's the most popular (aside from the ruling party) would be a `set_politics = { ruling_party = highest_party_ideology@exclude_ruling_party }`. Similarly, some event target-type variables can be used to cut down on code, such as `transfer_state = GER.capital` immediately transferring the capital of GER to the current scope.

MTTH variables[编辑 | 编辑源代码**]**

MTTH variables allow to dynamically create a MTTH block to be used as a variable. These are stored within /Hearts of Iron IV/common/mtth/*.txt files, where each MTTH variable is a separate block, the name of which gets taken as the MTTH variable.
As per the name, the MTTH variables serve as a MTTH block, so everything that applies to MTTH blocks applies here: The starting base is defined with `factor` or `base` and then each `modifier = { ... }` begin a trigger block needed to be fulfilled for the `add` (for adding) or/and `factor` (for multiplying) within to be applied. For example, the following is an example of a /Hearts of Iron IV/common/mtth/*.txt file:

```
my_mtth_variable_1 = {
  base = 10
  modifier = {
    factor = 10
    is_major = yes
  }
  modifier = {
    add = -10
    tag = FRA
  }
}
my_mtth_variable_2 = {
  base = other_variable
  factor = 0.5
  add = 10
}
```

This can be called the same way as a game variable, but somewhat differently: `set_variable = { my_var = mtth:my_mtth_variable_1 }`. This will check the MTTH block for the scope where it's checked and return the total result as a variable.

Debugging variables[编辑 | 编辑源代码]

There are the following arguments that can be used in trigger or effect blocks:

Debugging-related arguments: 折叠

Name	Parameters	Examples	Description	Notes
log	`<string>` What to write in the log.	log = "Current value of myVar: [?myVar]"	Creates an entry in the /Hearts of Iron IV/logs /game.log file in the user directory and, if console is open and debug mode is turned on, in console as well.	While it nominally has nothing to do with variables, its most common use is for variables.

print_variables	`file = <string>`
	The name of the file, not counting the file extension. Defaults to "variable_dump".
	`text = <string>`
	What will be printed in the beginning of the file. Defaults to "No Header".
	`append = yes`
	If set, will append the current variables (and the text) to the end of the file instead of overwriting.
	`print_global = yes`
	If true, will print the variables from the global scope as well.
	`var_list = { ... }`
	A whitespace-separated list of variables the values of which should be printed. Optional: every variable in the scope (and global if set) if left out.
		print_variables = {
		file = my_file
		text = "These variables are printed at [GetDateText]"
		append = yes
		print_global = yes
		}
		print_variables = {
		file = my_file_2
		var_list = { temp1 temp2 temp3 }
		}
			Dumps the specified variables from the current scope and optionally the global scope into a log file with the specified name.	The log will be within /Hearts of Iron IV/logs/variable_dumps/ in the user directory.

Additionally, the following console commands exist for variables. In console commands, the 'current scope' is what is currently selected: the country opened in diplomacy, the state selected, the unit leader that's selected. If left out, assumes to be the player country.

Debugging-related console commands: 折叠

Name	Parameters	Examples	Description

set_var	`<variable>`
	The variable to get the value of.
	`<value>`
	What to set the variable to.
		set_var my_variable 10	Changes the value of the variable within the specified scope.

get_var	`<variable>` The variable to get the value of.	get_var my_variable	Prints the value of the variable in console.
list_vars	`<scope>`Optional. Changes the scope from the currently-selected scope to what's specified.	list_vars	Lists the variables and their values as within the currently-selected scope.

Token-valued variables[编辑 | 编辑源代码]

When one uses a literal of the form `token:some_tokenizable_value`, what is actually returned is an integer variable (commonly with a large value), with an internal flag set so that it is interpreted as an index into a central table of token strings that exists within the game's memory. This can be placed into a variable and passed around freely, it can be compared for equality to other token variables, and it can even be treated as a numeric in arithmetic even if usages for such cases are rare if nonexistent. Variables that have the token flag set can be distinguished in the output of the `list_vars` console command by dint of being automatically translated into their underlying raw-token strings. For example, `some_var_name = token:some_tokenizable_value (27012)`.

When the variable contains a token value, it is possible to use the `GetTokenKey` localisation function (such as `[?some_var_with_token_value.GetTokenKey]`) to get the token string value of the variable, `token:` prefix not included. This is useful for pasting in meta-effects and meta-triggers, however note that the `token:` prefix is still required if using it within the `set_(temp_)variable` operator. With some token-valued variables (specifically, those backed by named database objects of a tokenizable datatype), it is possible to use `GetTokenLocalizedKey` to retrieve the value of the localisation key with the same name as the token itself. This can be used to produce and/or define human-readable names, but it can also be taken advantage of to pass around arbitrary localisation values.

At present (1.12), known tokenizable datatypes include (but are probably not limited to):

Ideas: These are generally the most flexible tokenizable datatype to use for a token that represents some arbitrary value by using a dummy idea. They can also carry an effectual payload in their `on_add` clauses, which can be used to pass around effects, or better yet, to return a set of variables encoding information relevant to the token value.

Ideologies/Ideology groups

Technologies

Equipment types - Sometimes this is a bit inconsistent with regards to the archetype/non-archetype distinction, so be wary.

Operations

Characters/Unit leaders: These are finicky in the game's core. When a character is promoted to a unit leader position, all tokenizations of that character's token will begin to return a different value, representing the token for the unit leader with the same identifier, which is linked to, but not the same as, the original character. The unit leader version of the token returns the same `GetTokenKey`

and `GetTokenLocalizedKey`, but it is *not numerically equal to the character version of the token*. This can come into play if code compares the equality of two character-pointing variables.

Decisions

Buildings

Advanced use of tokens[编辑 | 编辑源代码**]**

Using tokenizable values in variables is the key to a number of advanced techniques that enable relatively seamless abstraction and metaprogramming, with a minimum of scripted-localisation boilerplate code. These techniques are, however, difficult to learn how to apply and can be unintuitive for those not already experienced. Therefore, caution is advised.

It is often useful to define related dummy tokenizable database entries, especially ideas, to represent arbitrary (possibly enumerable) custom datatypes. In most cases, ideas are preferable, however there can be uses for other token types, such as technologies if the code is to implement something that can't be done with a regular idea. The reason for idea preference is little to no downstream effects simply from existing, ease to define in bulk with minimal syntactic overhead, and ease to separate different custom datatypes by using custom idea categories and by placing the definitions for different custom datatypes in different files within /Hearts of Iron IV/common/ideas/. Decisions or technologies are common alternatives for custom datatypes with different usages, but generally datatypes other than these have too much downstream baggage (i.e. their very existence has consequences elsewhere, such as custom building types appearing automatically in certain UIs) to be worth using for dummy values.

Using a systematic naming scheme, custom tokenizable datatypes can also convey multiple related localisation keys, similarly to the approach that the base game does for country naming with separate `TAG`, `TAG_DEF`, and `TAG_ADJ` keys. An example of that would be creating an idea `foo_bar` to act as the key identifier of the custom data value, whose localisation value carries the human-readable name of the data value, and another idea `foo_bar_description` (**NOT `foo_bar_desc`, this collides with the localisation key generated for `foo_bar`**) to carry the localisation value for the description that would be associated with the data value identified by `foo_bar`. This description localisation can be accessed with a meta-effect or meta-trigger (including in scripted localisation), such as `set_temp_variable = { description_var = token:[IDEA_TOKEN]_description }` (where `IDEA_TOKEN = "[?idea_var_name.GetTokenKey]"`). This allows accessing `[?description_var.GetTokenLocalizedKey]` in localisation where `description_var` is visible, such as the `localization_key` following the scripted localisation `trigger` block where the pasting meta-trigger is called.

As far more advanced usage, it is possible to rig up the dummy tokenizable values with arbitrary localisation payloads that carry fragments of Clausewitz code which can then be pasted together in meta-effects (or triggers) to approximate true token string manipulation or at least concatenation, as there's presently no way to split strings, token or otherwise. This is probably most useful for encoding a set of suffixes that can be applied to an idea token to dynamically navigate within the custom datatype's naming scheme. For example, if the mod has a custom datatype representing positions within a government such as the Secretary of State and the Attorney General, such that it might have a base idea `position_attorney_general`, a corresponding description idea (suffixed with `_description`), a singular name idea (suffixed with `_singular`, carrying the name of the data value in singular form, like "Attorney General"), and a plural name idea (suffixed with `_plural`, carrying the name in plural form, like "Attorneys General"), it is possible to select which name to use by having a dummy idea called `enum_position_name_singular` with the localisation value `"_singular"`, and another called `enum_position_name_plural` with the localisation value `"_plural"`. After that, the mod may pass one of the `enum_position` ideas in a variable and then paste its `GetTokenLocalizedKey` after the `GetTokenKey` of the base idea in a meta-effect/trigger to produce either `token:position_attorney_general_singular` or `token:position_attorney_general_plural`. This is a contrived example, and is also possible to be done using a scripted localisation system with hard-coded entries to access the singular vs. plural ideas, but this approach has far more use in more complex systems where creating a scripted localisation file can prove to be far too bulky.

From 1.12 onwards, anything defined in /Hearts of Iron IV/common/script_enums.txt can be freely tokenized, even if defined in a totally custom enumeration block, seperate from the ones hard-coded into the game. It is therefore theoretically possible to define arbitrary tokens of a custom enum type in this file. However, for compatibility reasons, using `script_enums` for custom enumeration types should be avoided, as this will ensure that the mod will not be compatible with any other mod that overwrites that file, no matter which mod gets priority on the file. As before, it's preferable to have a custom idea category with a set of dummy ideas in their own file that will not be overridden by other mods representing the values the custom enum can take on. This pattern is a useful replacement for passing around arbitrary integers that have some enumerable meaning when interpreted by code, especially if those integers are interpreted to yield names in scripted localisation, in which case this pattern can substantially remove the need for scripted localisation boilerplate by allowing defining the localised name for each enum member directly in a localisation file, as similarly to defining the localised name for an idea.

Technically, any raw token used in the language can actually be tokenized: for instance, `token:set_stability` (that is, a token literal of the name of the set_stability effect) is perfectly valid, and yields a token for which `GetTokenKey` and `GetTokenLocalizedKey` both return `set_stability`. However, this is rarely useful (especially since it does not apply to scripted effect and trigger names), and arguably counts as a bug; it mostly just provides an interesting peek into the internals of the tokenization system, indicating that it most likely delegates to a pre-existing string de-duplication system built into the Clausewitz engine.

Modifier tokens[编辑 | 编辑源代码]

> *This section is transposed from Modifiers § Modifier definitions*

Modifier definitions allow the creation of a custom modifier, which can be accessed as a variable when you wish to use it. After being defined, they function entirely like a new variable, being possible to read as a variable with the same `modifier@modifier_name` procedure. They will not have any effect by default and function only as a way to change the variable's value in an additive way with modifier blocks.
Each modifier token is defined within /Hearts of Iron IV/common/modifier_definitions/*.txt files as a separate code block, where the name of the code block serves as the ID of the modifier definition. There are the following arguments that can go inside of it:

`color_type = good` decides the colour of the modifier's value itself. There are three values, `good`, `bad`, and `neutral`. `neutral` is permamently yellow, while `good` turns the positive values **green** and negative values **red**. `bad` is the reversal of `good`.

`value_type = percentage` decides the way the number will show up in the tooltip. There are the following values this argument can be set to:
`number` will make the exact value of the modifier show up. 0.02 will show up like "Modifier definition: 0.02".

`percentage` will make the modifier show up as a percentage on the scale from 0 to 1. 0.2 will show up like "Modifier token: 20%".

`percentage_in_hundred` will make the modifier show up as a percentage on the scale from 0 to 100. 10 will show up like "Modifier token: 10%".

`yes_no` shows up as a boolean: anything larger than 1 will shows up as `yes`, while 0 will show up as `no`.

`precision = 1` decides how many numbers will be shown after the period in the tooltip. For example, if it is set to 1, the tooltip will show 0.341 as if it was 0.3. Note that the game inherently does not support more than 3 decimal numbers, so it cannot be larger than 3.

`postfix` decides what will be added after the value of the modifier in the tooltip. Allowed values are 'none' (Default), 'days', 'hours', and 'daily'.

`category` decides on the category of the modifier. By default, the category is 'all', which makes it be in every single category. Certain tooltips will only show modifiers if they belong to a certain category. It is possible to set multiple categories for the same modifier definition by defining them one after another.

> The allowed values are 'none', 'all', 'country', 'state', 'unit_leader', 'army', 'naval', 'air', 'peace', 'politics', 'ai', 'defensive', 'aggressive', 'war_production', 'military_advancements', 'military_equipment', 'autonomy', 'government_in_exile', and 'intelligence_agency'.

The modifier definition's ID is also used as the localisation key needed to change the name of the modifier depending on the currently turned on language.

Examples[编辑 | 编辑源代码**]**

```
modifier_definition_example = {
    color_type = good
    value_type = number
    precision = 1
    postfix = daily
```

```
    category = country
    category = state
}
modifier_definition_example_2 = {
  color_type = bad
  value_type = percentage
}
```

Using in variables[编辑 | 编辑源代码]

The function of modifier definitions is to modify the value of a game variable, which can be read by other variables. You can set a variable to be equal to the sum of all values of the same modifier token in the current scope by doing `set_variable = { var_name = modifier@modifier_token_name }`. This example will set var_name to be equal to the total value of modifier_token_name.

Note that, unlike countries and states, unit leaders use leader_modifier@modifier_definition_name or unit_modifier@modifier_definition_name in their scopes.

Arrays[编辑 | 编辑源代码]

Arrays are similar to variables, but they instead store a collection of variables.
An array in the game is structured with indexes starting from 0. `arrayname^0` as a variable refers to the first item of `arrayname`, `arrayname^1` refers to the second item in the array, and so on. In order to obtain the total amount of items within the given array, `arrayname^num` is used. An array is assumed to have no elements by default, if not already created, with add_to_array or add_to_temp_array usually being used to create an array.

Temporary and regular arrays exist, with the same distinction between them as in variables: temp arrays can be set and modified in triggers and effects, are unscoped, and cleared after the effect/trigger block ends; regular arrays can only be set or modified in effects, are linked to a specific scope, and persist forever until cleared manually.

A single element of an array is considered a variable in every regard: `set_variable = { arrayname^0 = 10 }` will set the first element to be equal to 10 (with `set_temp_variable` used for temporary arrays), `check_variable = { arrayname^0 > 10 }` is also valid syntax, and `[?arrayname^0]` will print the value of the element in localisation. However, the element must already exist for it to be edited. **There is no simple way to print an entire array in localisation**, however a recursive scripted localisation can be used, which modifies the value of a temporary variable by one until hitting the array's size each time that it's used.

Arguments[编辑 | 编辑源代码]

The following arguments exist for arrays. Similarly to variables, temporary versions exist by replacing `array` with `temp_array`.

Arguments for modifying arrays: 折叠

Name	Parameters	Examples	Description	Notes

add_to_array	`array = <array>`
	The array to modify.
	`value = <decimal>/<variable>`
	The variable to add.
	`index = <integer>`
	The index to place the variable on in the array. Optional, defaults to the end of the array.
		add_to_array = {
		array = global.my_countries
		value = THIS.id
		}
		
		add_to_temp_array = {
		temp_states = THIS }
			Adds an element to the variable either at the end or the specified index.	Shortened version exists with `add_to_array = { <array> = <value> }`.

remove_from_array	`array = <array>`
	The array to modify.
	`value = <decimal>/<variable>`
	The variable to remove. Optional.
	`index = <integer>`
	The index to remove the variable from in the array. Optional.
		remove_from_array = {
		array = global.my_countries
		index = 0
		}
		
		remove_from_temp_array = {
		temp_states = THIS }
			Removes an element from the variable either with the specified value or the index.	Shortened version exists with `remove_from_array = { <array> = <value> }`. If neither value nor index are specified, then the last element is deleted.

clear_array	`<array>` The array to clear.	clear_array = global.my_countries clear_temp_array = temp_states	Clears the array, removing every element inside.	

resize_array	`array = <array>`
	The array to modify.
	`value = <decimal>/<variable>`
	The variable to add to the array if the size is larger than the array's current size. Optional, defaults to 0.
	`size = <integer>`
	The amount of elements inside of the array after the resizing.
		resize_array = {
		array =
		global.countries_by_states
		value = 10
		size = global.countries^num
		}
		
		resize_temp_array = {
		temp_states = 20 }
			Resizes the array, removing or adding elements in the end if necessary.	Shortened version exists with `resize_array = { <array> = <size> }`.

find_highest_in_array	`array = <array>`
	The array to modify.
	`value = <variable>`
	The temporary variable where the largest value will get stored.
	`index = <variable>`
	The temporary variable where the index of the largest value will get stored.
		find_highest_in_array = {
		array =
		global.countries_by_states
		value = temp_largest_country
		index = temp_country_index
		}
			Finds the largest value in the array and assigns its value and index to a temporary variable.	Either value or index are optional to specify.

find_lowest_in_array	`array = <array>`
	The array to modify.
	`value = <variable>`
	The temporary variable where the smallest value will get stored.
	`index = <variable>`
	The temporary variable where the index of the smallest value will get stored.
		find_lowest_in_array = {
		array =
		global.countries_by_states
		value = temp_largest_country
		index = temp_country_index
		}
			Finds the smallest value in the array and assigns its value and index to a temporary variable.	Either value or index are optional to specify.

Meanwhile, the following triggers exist that are array-related. As with variables, these are the exact same for temporary and regular variables.

Array-related triggers: 折叠

Name	Parameters	Examples	Description	Notes

is_in_array	`array = <array>`
	The array to check.
	`value = <decimal>/<variable>`
	The value to check for.
		is_in_array = {
		array = global.my_countries
		value = THIS.id
		}
		
		is_in_array = { temp_states = THIS }
			Checks if any value within the array is the same as the specified value.	Shortened version exists with `is_in_array = { <array> = <value> }`.

any_of	`array = <array>`
	The array to check.
	`value = <variable>`
	The name of the temporary variable where the value of the currently-selected
		any_of = {
		array = temp_numbers
		value = v
		index = i
		multiply_temp_variable = { v = i }
		check_variable = { v > political_power } # Checks
			Checks if any value within the array fulfills the triggers, halting and returning true if that's the case.	The scope does not change from the one this is checked in by default, requiring one of scopes.

Name	Parameters	Examples	Description	Notes
	element is stored. index = <variable> The name of the temporary variable where the index of the currently-selected element is stored. <triggers> An AND trigger block.	if the value multiplied by the index is larger than the political power of the country this is checked in. }		

any_of_scopes	array = <array>
	The array to check.
	tooltip = <localisation key>
	The localisation key used for the trigger.
	<triggers>
	An AND trigger block.
		any_of_scopes = {
		array = global.majors
		tooltip = has_more_states_than_any_other_major_tt
		NOT = { tag = PREV }
		check_variable = { num_owned_controlled_states > PREV.num_owned_controlled_states }
		}
			Checks if any value within the array fulfills the triggers, halting and returning true if that's the case, scoping into each element in the array.	Appending _NOT to the tooltip's key (such as has_more_states_than_any_other_major_tt_NOT in the example) results in the localisation key used if this any_of_scopes is put inside of NOT = { ... }. Since this changes scopes, Scopes#PREV would, unless there's deeper scoping, refer to the scope this trigger is checked in.

all_of	array = <array>
	The array to check.
	value = <variable>
	The name of the temporary variable where the value of the currently-selected element is stored.
	index = <variable>
	The name of the temporary variable where the index of the currently-selected element is stored.
	<triggers>
	An AND trigger block.
		all_of = {
		array = temp_numbers
		value = v
		index = i
		multiply_temp_variable = { v = i }
		check_variable = { v > political_power } # Checks if the value multiplied by the index is larger than the political power of the country this is checked in.
		}
			Checks if every value within the array fulfills the triggers, halting and returning false if any one doesn't.	The scope does not change from the one this is checked in by default, requiring one of scopes.

all_of_scopes	array = <array>
	The array to check.
	tooltip = <localisation key>
	The localisation key used for the trigger.
	<triggers>
	An AND trigger block.
		all_of_scopes = {
		array = global.majors
		tooltip = has_more_states_than_every_other_major_tt
		OR = {
		tag = PREV
		check_variable = { num_owned_controlled_states < PREV.num_owned_controlled_states }
		}
		}
			Checks if every value within the array fulfills the triggers, halting and returning false if any one doesn't, scoping into each element in the array.	Appending _NOT to the tooltip's key (such as has_more_states_than_every_other_major_tt_NOT in the example) results in the localisation key used if this any_of_scopes is put inside of NOT = { ... }. Since this changes scopes, Scopes#PREV would, unless there's deeper scoping, refer to the scope this trigger is checked in.

The following effects also exist in addition used, with arrays. While some are nominally unrelated to arrays, they are often used in conjunction.

Array-related effects: 折叠

Name	Parameters	Examples	Description	Notes

for_each_loop	array = <array>
	The array to check.
	value = <variable>
	The name of the temporary variable where the value of the currently-selected element is stored.
	index = <variable>
	The name of the temporary variable where the index of the currently-selected element is stored.
	break = <variable>
	The temporary variable that can be set to be not 0 to instantly break the loop.
	<effects>
	An effect block.
		for_each_loop = {
		array = temp_numbers
		value = v
		break = break
		random_list = {
		10 = { add_political_power = v }
		10 = {
		divide_temp_variable = { v = 100 }
		add_stability = v
		}
		10 = {
		divide_temp_variable = { v = 100 }
		add_war_support = v
		}
		}
		}
			Runs an effect once for every element in the array.	The scope does not change from the one this is checked in by default, requiring one of scopes.

for_each_scope_loop	array = <array>
	The array to check.
	break = <variable>
	The temporary variable that can be set to be not 0 to instantly break the loop.
	<effects>
	An effect block.
		for_each_scope_loop = {
		array = global.majors
		if = {
		limit = {
		NOT = { tag = ROOT }
		}
		random_owned_controlled_state = {
		transfer_state_to = ROOT
		}
		}
		}
			Runs the effects for every scope within the array.	Equivalent to a every_<...> effect scope type, with additional break. Since this changes scopes, Scopes#PREV would, unless there's deeper scoping, refer to the scope this trigger is checked in.

random_scope_in_array	array = <array>
	The array to check.
	break = <variable>
	The temporary variable that can be set to be not 0 to instantly break the loop.
	limit = { <triggers> }
	An AND trigger block deciding which scopes can be picked.
	<effects>
	An effect block.
		random_scope_in_array = {
		array = global.countries
		break = break
		limit = {
		is_dynamic_country = no
		exists = no
		any_state = {
		is_core_of = PREV # Is core of the currently-checked country
		}
		}
		random_core_state = {
		transfer_state_to = PREV # Transfers to the currently-selected country.
		}
		}
			Runs the effects for a random scope within the array.	Equivalent to a random_<...> effect scope type, with additional break. Since this changes scopes, Scopes#PREV would, unless there's deeper scoping, refer to the scope this trigger is checked in.

while_loop_effect	limit = { <triggers> }
	An AND trigger block deciding when the effect would repeat.
	break = <variable>
	The temporary variable that can be set to be not 0 to instantly break the loop.
	<effects>
	An effect block.
		while_loop_effect = {
		limit = {
		any_country = {
		num_owned_controlled_states > 10
		}
		}
		random_country = {
		limit = {
		num_owned_controlled_states > 10
			Runs the effects as long as the limit is fulfilled.	The limit is only checked at the beginning and once the effect block is executed. Can't run for more than 1000 times by default[5].

Name	Parameters	Examples	Description	Notes

		}
		random_other_country = {
		limit = {
		num_owned_controlled_states < 10
		}
		PREV = {
		random_owned_controlled_state = {
		transfer_state_to = PREV.PREV
		}
		}
		}
		}
		}		

for_loop_effect	`start = <decimal>/<variable>`
	The value at which the evaluation starts. Default to 0.
	`end = <decimal>/<variable>`
	The value at which the evaluation ends. Default to 0.
	`compare_type = <compare type>`
	How the currently-evaluated value must compare with the `end` for the for loop to continue. Defaults to `less_than`, leading to the end value never being picked.
	`add = <variable>`
	How much is being added to the `start` value for every iteration. Defaults to 1.
	`break = <variable>`
	The temporary variable that can be set to be not 0 to instantly break the loop.
	`value = <variable>`
	The name of the temporary variable where the current value of evaluation is stored.
	`<effects>`
	An effect block.
		for_loop_effect = {
		end = 4
		value = temp
		if = {
		limit = {
		NOT = {
		count_triggers = {
		amount = 2
		check_variable = { temparr^temp < temparr^0 }
		check_variable = { temparr^temp < temparr^1 }
		check_variable = { temparr^temp < temparr^2 }
		check_variable = { temparr^temp < temparr^3 }
		}
		}
		}
		add_to_array = { potential_max = temp }
		}
		}
			Runs the effects as long as the limit is fulfilled.	The limit is only checked at the beginning and once the effect block is executed. Can't run for more than 1000 times by default[5]. Comparison types are the same as used in check_variable.

Scorers[编辑 | 编辑源代码**]**

Scorers are defined within /Hearts of Iron IV/common/scorers/country/*.txt files, serving as a sorted list of targeted countries, based on how well they did in the specified MTTH block. The scorer has the primary country which the scorer is used for and targeted countries.
Each scorer is a separate block within the file, with the name of the block being the name of the scorer. The scorer then has the `targets = { ... }` block that encompasses everything else about the scorer.
Scorers target countries similarly to targeted decisions but with differences, with the following arguments being used within of `targets = { ... }`:

`targets = { TAG, TAG }` - A list of country tags, including other dual scopes such as overlord, that can be targeted by this scorer. Unlike targeted decisions, this seems to be comma-separated.

`target_array = global.majors` - The array to target. The default scope is the primary country.

`target_root_trigger = { ... }` - The trigger block which the primary country must fulfill for the target to work.

`target_trigger = { ... }` - The trigger block that target country must fulfill for the target to work.

`targets_dynamic = yes` - If set, makes the dynamic countries possible targets. Defaults to no.

`target_non_existing = yes` - If set, makes non-existing countries possible targets. Defaults to no.

`score = { ... }` - This is the MTTH block itself that ranks the countries in the list. By default, THIS (the default scope) is the target, FROM is the primary country initialising the check.

For example, the following is a scorer that sorts every major country depending on the amount of states they own:

```
majors_owned_states = {
  targets = {
    target_array = global.majors    # Global is not the default scope, scope into it.
    targets_dynamic = yes
    target_non_existing = yes      # As implausible as this is, this is likely technically possible by using the set_major effect.
    score = {
      factor = num_owned_states
    }
  }
}
```

After defining a scorer, there is the way to use it: an argument, similar to arrays and variables. Similarly to them, regular and temporary versions exist, which decides whether the variable/array it's writing to is regular or temporary. Both can only be used inside of country scope. Regular versions can only be used in effects, and temporary versions can be used in effects and triggers.

Scorer-related arguments: 折叠

Name	Parameters	Examples	Description	Notes

get_highest_scored_country	`scorer = <scorer>`
	The scorer to use.
	`var = <variable>`
	The variable where the result will be stored.
		get_highest_scored_country = {
		var = majors_owned_states
		value = largest_major
		}
		get_highest_scored_country_temp = {
		var = majors_owned_states
		value = largest_major
		}
			Sets the specified variable to be the highest-scoring country within the scorer, as checked for the country where this is used.	

Name	Parameters	Examples	Description	Notes

get_sorted_scored_countries	`scorer = <scorer>`
	The scorer to use.
	`array = <variable>`
	The array where the countries will be stored.
	`scores = <variable>`
	The array where the scores will be stored.
		get_sorted_scored_countries = {
		scorer = majors_owned_states
		array = major_owners
		values = major_states
		}
		
		get_sorted_scored_countries_temp = {
		scorer = majors_owned_states
		array = major_owners
		values = major_states
		}
			Creates two arrays of countries and their corresponding values, sorted in the ascending order.	Values with the same index are connected: in the example, major_owners^0 is the country that got major_states^0 points in the majors_owned_states scorer.

Example[编辑 | 编辑源代码]

Scorers can be helpful to create a sorted list of scopes by a certain element, such as political power. For example, the following can be used to find the countries with the most infantry equipment and send an event to the top three that are at peace with ROOT:

Within the scorer file:

```
infantry_equipment_sort = {
  target_array = global.countries
  targets_dynamic = yes
  score = {
    factor = num_equipment@infantry_equipment
  }
}
```

Within an effect block:

```
ROOT = {
  get_sorted_scored_countries_temp = {
    scorer = infantry_equipment_sort
    array = temp_inf_eq_countries
  }
  for_each_scope_loop = {
    array = temp_inf_eq_countries
    break = break
    if = {
      limit = {
        NOT = {
          tag = ROOT
          has_war_with = ROOT
        }
      }
      add_to_temp_variable = { v = 1 }
      country_event = equipment_please.0
    }
    if = {
      limit = {
        check_variable = { v = 3 }
      }
      set_temp_variable = { break = 1 }
    }
  }
}
```

Game variables[编辑 | 编辑源代码]

Game variables are read-only variables that are used to read certain aspects of the game, such as the current political power that the player has. These cannot be modified using any variable-modifying argument, but they can serve as the value within them (such as `add_to_temp_variable = { temp = GER.army_experience }` adding the amount of army experience that GER has to the `temp` temporary variable. Some are targeted, requiring one more argument in the target's place. For example, `modifier` has a modifier token as the target, so it would be used as `set_temp_variable = { temp = modifier@consumer_goods_factor }`, where `consumer_goods_factor` is the name of the modifier used as the argument.

Remember that to get global-scoped game variables, it is necessary to scope into global as `global.year` rather than leaving out the scope. A list of game variables can be found within /Hearts of Iron IV/documentation/dynamic_variables_documentation, alongside that a large portion of purely numeric Triggers have corresponding game variables. The following game variables exist in the game (Some duplicates are omitted):

Global-scoped game variables: 折叠

Variable	Target	Description/Notes
year	None.	The current year, such as 1936.
date	None.	The ID of the current date. This can be localised in a multitude of different ways, using the localisation functions beginning with `GetDateString`. This variable functions by having one game hour be 0.001, with 0 of the variable corresponding to the date 1st January -5000, 1:00.
num_days	None.	The amount of days passed since the start date. Start date is considered the NDefines.NGame.START_DATE define.
difficulty	None.	The current difficulty value on the scale from 0 (For Civilian) to 4 (For Elite).
threat	None.	The current world tension value on the scale from 0 to 1.

Country-scoped general game variables: 折叠

Variable	Target	Description/Notes
modifier	Modifier's name, such as `@political_power_gain`.	The total value of the current modifier as the country has in total. Targeted modifiers have separate definitions.
id	None.	The internal ID of the country. Equivalent to not specifying anything, like `set_temp_variable = { temp = TAG }`.
max_manpower	None.	Total population of the country. Deprecated, recommended to use max_manpower_k instead as to avoid overflows.
max_manpower_k	None.	Total population of the country in thousands.

Variable	Target	Description/Notes
party_popularity	Ideology group or `ruling_party`, such as `@ruling_party`.	The popularity of the specified political party in the range from 0 to 1.
party_popularity_100	Ideology group or `ruling_party`, such as `@neutrality`.	The popularity of the specified political party in the range from 0 to 100.
highest_party_ideology	`exclude_ruling_party` or nothing.	The ideology group that has the highest party popularity within the country, optionally excluding the ruling party.
highest_party_popularity	`exclude_ruling_party` or nothing.	The popularity of the ideology group that has the highest party popularity within the country, optionally excluding the ruling party. The popularity is on the scale from 0 to 1.
current_party_ideology_group	None.	Ideology group that the country currently has.
original_tag	None.	The tag from which the current country originates. Same as the regular tag for non-dynamic countries.
opinion	Country.	The opinion of the current country towards the specified one.
days_decision_timeout	Decision, such as `@my_decision`.	The remaining amount of days for a completed decision before it gets removed.
days_mission_timeout	Mission.	The remaining amount of days for an ongoing mission before it times out.
political_power	None.	Total current political power of the country. `has_political_power` can be used in the same way.
political_power_daily	None.	Current amount of daily gain of political power, equivalent to `political_power_growth`.
stability	None.	Current stability of the country on the scale from 0 to 1.
has_war_support	None.	Current war support of the country on the scale from 0 to 1.
power_balance_value	None.	The current power balance value.
power_balance_daily	None.	The current daily change in power balance.
power_balance_weekly	None.	The current weekly change in power balance.
capital	None.	The capital state of the country.
autonomy_ratio	None.	How close the country is to taking the next autonomy level. 0 if can be downgraded, 1 if can be upgraded, -1 if independent.
overlord	None.	The country that the current country is a subject of.
num_subjects	None.	The number of subjects that the country has as an overlord.
core_compliance	Country.	Average compliance within cores of the target country occupied by the current country.
core_resistance	Country.	Average resistance within cores of the target country occupied by the current country.
has_collaboration	Country.	The collaboration of the target country with the current country within target's cores currently occupied by the current country.
faction_leader	None.	The leader of the faction the current country is in. Returns nothing if not in a faction.
num_faction_members	None.	The number of members in the faction with the current country.
fuel_k	None.	Current fuel of country in thousands.
max_fuel_k	None.	Maximum fuel of country in thousands.
fuel_ratio	None.	Ratio of the current fuel compared to the maximum amount, from 0 to 1.
host	None.	The host of the current country. Returns nothing if not exiled.
legitimacy	None.	The legitimacy of the currently-exiled country. Returns -1 if not exiled.
num_controlled_states	None.	The amount of states that are controlled but not necessarily owned by the current country.
num_core_states	None.	The amount of states that are national territory of the current country.
num_owned_controlled_states	None.	The amount of states that are owned and controlled by the current country.
num_owned_states	None.	The amount of states that are owned but not necessarily controlled by the current country.
num_occupied_states	None.	The number of states occupied by the country.
resource	Resource, such as `@steel`.	The amount of surplus resource of the specified type, which isn't exported or used in production.
resource_consumed	Resource, such as `@aluminium`.	The amount of resource of the specified type currently used for equipment production.
resource_exported	Resource, such as `@oil`.	The amount of resource of the specified type currently set to be exported with the min_export modifier.
resource_imported	Resource, such as `@tungsten`.	The amount of resource of the specified type currently set to be imported from trade with other countries.
resource_produced	Resource, such as `@rubber`.	The amount of resource of the specified type produced from the country's controlled states, buildings on them, and resource rights.
alliance_strength_ratio	Country.	The ratio of land strength between the current country's faction and the target's faction.
alliance_naval_strength_ratio	Country.	The ratio of naval strength between the current country's faction and the target's faction.
enemies_naval_strength_ratio	None.	The estimated navy strength between the current country and all its enemies.
enemies_strength_ratio	None.	The estimated army strength between the current country and all its enemies.
amount_research_slots	None.	The current amount of research slots the country has.
original_research_slots	None.	The amount of research slots that the country had in the beginning of the game, as set in the history file.
any_war_score	None.	The highest war score that the country has among wars it's in, on the scale from 0 to 100.
casualties	None.	The amount of casualties a country has suffered in all of its wars.
casualties_k	None.	The amount of casualties a country has suffered in all of its wars in thousands.
days_since_capitulated	None.	The amount of days since the country has capitulated. Not intended to be used if the country has never capitulated, returning an 'extremely large' value[6].
convoy_threat	None.	Returns, in the range from 0 to 1, the current danger that enemy convoys pose to the current country. Uses NDefines.NNavy.NAVAL_CONVOY_DANGER_RATIOS as the baseline.
mine_threat	None.	Returns, in the range from 0 to 1, the current danger that enemy mines pose to the current country. Uses NDefines.NNavy.NAVAL_MINE_DANGER_RATIOS as the baseline.
foreign_manpower	None.	The amount of foreign garrison manpower that the country has.
garrison_manpower_need	None.	The amount of manpower needed by garrisons.
has_added_tension_amount	None.	The amount of world tension currently added by the country.
manpower_per_military_factory	None.	The amount of manpower that the country has for each military factory.
num_of_available_civilian_factories	None.	Amount of available civilian factories.
num_of_available_military_factories	None.	Amount of available military factories.
num_of_available_naval_factories	None.	Amount of available naval factories.

Variable	Target	Description/Notes
num_of_civilian_factories	None.	Amount of civilian factories.
num_of_civilian_factories_available_for_projects	None.	Amount of civilian factories available for a new project to use.
total_constructed_civilian_factory	None.	Amount of civilian factories that have been constructed after the game's start by the country.
num_of_factories	None.	Amount of total factories.
num_of_military_factories	None.	Amount of military factories.
num_of_naval_factories	None.	Amount of naval factories.
num_of_controlled_factories	None.	Amount of factories within controlled states.
num_of_owned_factories	None.	Amount of factories within owned states.
num_of_nukes	None.	Amount of nukes.
num_researched_technologies	Number of technologies a tag has researched.	
num_tech_sharing_groups	None.	How many tech sharing groups a nation is a member of.
surrender_progress	None.	How close the country is to surrendering on the scale from 0 to 1.

Country-scoped intelligence-related game variables: 折叠

Variable	Target	Description/Notes
army_intel	Country.	Army intel against the target country.
navy_intel	Country.	Navy intel against the target country.
air_intel	Country.	Air intel against the target country.
civilian_intel	Country.	Civilian intel against the target country.
encryption_strength	None.	Total encryption strength of the country. Only without the La Résistance DLC.
decryption_speed	None.	Total decryption speed of the country. Only without the La Résistance DLC.
cryptology_defense_level	None.	cryptology defense level of the country.
agency_upgrade_number	None.	The number of upgrade done in the intelligence agency.
decryption_progress	Country.	Decryption progress against the target country.
network_national_coverage	Country.	Network national coverage the current country has over the target country.
num_fake_intel_divisions	None.	The amount of fake intel division that the country has.
num_free_operative_slots	None.	The amount of operative slots that are currently empty and an operative can be recruited in.
num_of_operatives	None.	The number of operatives the country controls.
num_operative_slots	None.	The number of available operative slots a country has. If this differs from the number of operative, this does not mean the country can recruit an operative, but that it will eventually be able to.

Country-scoped military-related game variables: 折叠

Variable	Target	Description/Notes
manpower	None.	Manpower that the country has, both in the army and reserved. Deprecated, recommended to use manpower_k instead as to avoid overflows.
manpower_k	None.	Manpower that the country has, both in the army and reserved, in thousands.
max_available_manpower	None.	Manpower that the country has, both in the army and reserved, taking into account those that are in the process of being drafted as the result of a recent conscription amount change. Deprecated, recommended to use max_available_manpower_k instead as to avoid overflows.
max_available_manpower_k	None.	Manpower that the country has, both in the army and reserved, taking into account those that are in the process of being drafted as the result of a recent conscription amount change, in thousands.
target_conscription_amount	None.	The target conscription amount of the country.
amount_manpower_in_deployment_queue	None.	Amount of manpower currently in the deployment view.
conscription_ratio	None.	Conscription ratio of the country compared to target conscription ratio.
current_conscription_amount	None.	The current conscription amount of the country.
command_power	None.	The current command power of country.
command_power_daily	None.	The current daily gain of command power by country.
army_experience	None.	Army experience that the country has.
navy_experience	None.	Naval experience that the country has.
air_experience	None.	Air experience that the country has.
num_armies	None.	Amount of land divisions.
num_armies_in_state	State such as `@123`.	Amount of land divisions within the target state.
num_armies_with_type	Subunit type such as `@light_armor`.	Amount of land divisions that are majority of the same subunit type.
num_battalions	None.	Amount of battalions within land divisions the country has.
num_battalions_with_type	Subunit type such as `@cavalry`.	Amount of battalions within land divisions the country has that have the specified subunit type.
num_deployed_planes	None.	Amount of deployed planes.
num_deployed_planes_with_type	Subunit type such as `@fighter`.	Amount of deployed planes with equipment type.
num_divisions	None.	Amount of land and naval divisions.
num_equipment	Equipment type or archetype such as `@infantry_equipment`.	Amount of equipment within the country's stockpile.
num_equipment_in_armies	Equipment type or archetype such as `@artillery_equipment`.	Amount of equipment in country that are used in armies. If there's a large amount of equipment possible, use num_equipment_in_armies_k.
num_equipment_in_armies_k	Equipment type or archetype such as `@support_equipment`.	Amount of equipment in country that are used in armies, in thousands.
num_target_equipment_in_armies	Equipment type or archetype such as `@motorized_equipment_1`.	Amount of equipment in country that are wanted in armies. If there's a large amount of equipment possible, use num_target_equipment_in_armies_k.
num_target_equipment_in_armies_k	Equipment type or archetype such as `@anti_air_equipment_1`.	Amount of equipment in country that are wanted in armies, in thousands.
num_ships	None.	Amount of ships.
num_ships_with_type	Subunit type such as `@carrier`.	Amount of ships with the specified subunit type.
land_doctrine_level	None.	The currently-researched land doctrine level.

A large portion of the AI-related game variables can have values seen in-game by using the `aiview` console command.

Country-scoped AI-related game variables: 折叠

Variable	Target	Description/Notes
ai_attitude_allied_weight	Country.	The total amount of the `attitude_allied` AI attribute.
ai_attitude_friendly_weight	Country.	The total amount of the `attitude_friendly` AI attribute.
ai_attitude_hostile_weight	Country.	The total amount of the `attitude_hostile` AI attribute.
ai_attitude_is_threatened	None.	Returns 1 if the AI is threatened, 0 otherwise.
ai_attitude_neutral_weight	Country.	The total amount of the `attitude_neutral` AI attribute.
ai_attitude_outraged_weight	Country.	The total amount of the `attitude_outraged` AI attribute.
ai_attitude_protective_weight	Country.	The total amount of the `attitude_protective` AI attribute.
ai_attitude_threatened_weight	Country.	The total amount of the `attitude_threatened` AI attribute.
ai_attitude_wants_ally	None.	Returns 1 if the AI wants to ally any country, 0 otherwise.
ai_attitude_wants_antagonize	None.	Returns 1 if the AI wants to antagonize any country, 0 otherwise.
ai_attitude_wants_ignore	None.	Returns 1 if the AI wants to ignore any country, 0 otherwise.
ai_attitude_wants_protect	None.	Returns 1 if the AI wants to protect any country, 0 otherwise.
ai_attitude_wants_weaken	None.	Returns 1 if the AI wants to weaken any country, 0 otherwise.
ai_strategy_activate_crypto	Country.	Total sum of the AI strategy values of the type `activate_crypto` towards the target country.
ai_strategy_alliance	Country.	Total sum of the AI strategy values of the type `alliance` towards the target country.
ai_strategy_antagonize	Country.	Total sum of the AI strategy values of the type `antagonize` towards the target country.
ai_strategy_befriend	Country.	Total sum of the AI strategy values of the type `befriend` towards the target country.
ai_strategy_conquer	Country.	Total sum of the AI strategy values of the type `conquer` towards the target country.
ai_strategy_consider_weak	Country.	Total sum of the AI strategy values of the type `consider_weak` towards the target country.
ai_strategy_contain	Country.	Total sum of the AI strategy values of the type `contain` towards the target country.
ai_strategy_declare_war	Country.	Total sum of the AI strategy values of the type `declare_war` towards the target country.
ai_strategy_decrypt_target	Country.	Total sum of the AI strategy values of the type `decrypt_target` towards the target country.
ai_strategy_dont_defend_ally_borders	Country.	Total sum of the AI strategy values of the type `dont_defend_ally_borders` towards the target country.
ai_strategy_force_defend_ally_borders	Country.	Total sum of the AI strategy values of the type `force_defend_ally_borders` towards the target country.
ai_strategy_ignore	Country.	Total sum of the AI strategy values of the type `ignore` towards the target country.
ai_strategy_ignore_claim	Country.	Total sum of the AI strategy values of the type `ignore_claim` towards the target country.
ai_strategy_influence	Country.	Total sum of the AI strategy values of the type `influence` towards the target country.
ai_strategy_invade	Country.	Total sum of the AI strategy values of the type `invade` towards the target country.
ai_strategy_occupation_policy	Country.	Total sum of the AI strategy values of the type `occupation_policy` towards the target country.
ai_strategy_prepare_for_war	Country.	Total sum of the AI strategy values of the type `prepare_for_war` towards the target country.
ai_strategy_protect	Country.	Total sum of the AI strategy values of the type `protect` towards the target country.
ai_strategy_send_volunteers_desire	Country.	Total sum of the AI strategy values of the type `send_volunteers_desire` towards the target country.
ai_strategy_support	Country.	Total sum of the AI strategy values of the type `support` towards the target country.
ai_irrationality	None.	The AI's irrationality value.
ai_wants_divisions	None.	The total amount of divisions that the AI aims for.

State-scoped game variables: 折叠

Variable	Target	Description/Notes
id	None.	The internal ID of the state. While regular `set_variable = { my_state_pointer = 123 }` would still work as a pointer to the state, this can remove ambiguity within the code whether the number is used as a state ID or the number that matches up with the state ID.
building_level	Building such as `@nuclear_reactor`.	The level of a building with the target type.
arms_factory_level	None.	Military factory level in the state
industrial_complex_level	None.	civilian factory level in the state
infrastructure_level	None.	infrastructure level in the state
damaged_building_level	Building such as `@industrial_complex`.	Damaged building level of a building with target type.
non_damaged_building_level	Building such as `@arms_factory`.	Non-damaged building level of a building with target type.
modifier	Modifier's name, such as `@political_power_gain`.	The total value of the current modifier as the country has in total. Targeted modifiers have separate definitions.
controller	None.	Controller of the state.
owner	None.	Owner of the state.
distance_to	State.	Distance between the current state and the target state.
resource	Resource type such as `@chromium`	Amount of the resource produced within state.
compliance	None.	Current compliance level of the state.
compliance_speed	None.	Current compliance speed of the state.
days_since_last_strategic_bombing	None.	Amount of days since the state was last strategic bombed.
resistance	None.	Current resistance level of the state.
resistance_speed	None.	Current resistance speed of the state.
resistance_target	None.	Current resistance target of the state.
state_population	None.	Current population of the state. Recommended to use state_population_k when possible to avoid overflows.
state_population_k	None.	Current population of the state in thousands.
state_and_terrain_strategic_value	None.	Current state and terrain strategic value, decided by buildings and terrain of provinces.
state_strategic_value	None.	Current state strategic value, decided by buildings.

Unit leader-scoped game variables: 折叠

Variable	Target	Description/Notes

Variable	Target	Description/Notes
leader_modifier	Modifier's name, such as `@navy_max_range`.	The total value of the specified modifier as the unit leader has in total, applied towards the leader. This is represented with `non_shared_modifier = { ... }` in unit leader traits.
unit_modifier	Modifier type, such as `@army_attack_factor`	The total value of the specified modifier as the unit leader has in total, applied towards the units. This is represented with `modifier = { ... }` in unit leader traits.
sum_unit_terrain_modifier	Modifier type, such as `@sickness_chance`	The total sum of the modifier value applied by each province that this unit leader has divisions on.
army_attack_level	None.	The attack level of the leader.
army_defense_level	None.	The defense level of the leader.
attack_level	None.	The attack level of the leader.
coordination_level	None.	The coordination level of the leader.
defense_level	None.	The defense level of the leader.
logistics_level	None.	The logistics level of the leader.
maneuvering_level	None.	The maneuvering level of the leader.
planning_level	None.	The maneuvering level of the leader.
skill_level	None.	The general skill level of the leader.
average_stats	None.	The average stats of the unit leader.
avg_combat_status	None.	The average status of each engagement that units led by the leader are participating in.
avg_defensive_combat_status	None.	The average status of each defensive engagement that units led by the leader are participating in.
avg_offensive_combat_status	None.	The average status of each offensive engagement that units led by the leader are participating in.
avg_unit_planning_ratio	None.	The average ratio of planning bonus that each unit has.
avg_units_acclimation	Acclimatization type, such as `@hot_climate`	Average unit acclimatization for the specified climate, which is defined in /Hearts of Iron IV/common /acclimatation.txt
has_orders_group	None.	Returns 1 if the general has any order groups assigned to them, 0 otherwise.
num_traits	None.	Amount of traits that the unit leader has.
num_assigned_traits	None.	Amount of assigned traits the leader has.
num_basic_traits	None.	Amount of basic traits that the unit leader has.
num_personality_traits	None.	Amount of personality traits that the unit leader has.
num_status_traits	None.	Amount of status traits that the unit leader has.
num_max_traits	None.	Amount of maximum assignable traits the leader can have.
num_terrain_traits	None.	Amount of terrain traits that the unit leader has.
num_battalions	None.	Amount of battalions that the army leader has within units under their control.
num_battalions_with_type	Subunit type such as `@artillery_brigade`	Amount of battalions with sub unit type, sub unit type is defined in target, example: num_battalions_with_type@light_armor
num_battle_plans	None.	Amount of battle plans that the unit leader is assigned to.
num_orders_groups	None.	Number of order groups assigned to the unit leader.
num_equipment	Equipment type or archetype such as `@light_tank_chassis`.	Amount of equipment fulfilled within units led by the unit leader.
num_target_equipment	Equipment type or archetype such as `@anti_tank_equipment`	Amount of equipment required within units led by the unit leader.
num_units_with_type	Subunit type such as `@modern_armor`	The amount of units controlled by the unit leader that are predominantly filled with subunits that are assigned the specified type.
num_armored	None.	The amount of units controlled by the unit leader that are predominantly filled with subunits that are assigned the `armored` type.
num_artillery	None.	The amount of units controlled by the unit leader that are predominantly filled with subunits that are assigned the `artillery` type.
num_cavalry	None.	The amount of units controlled by the unit leader that are predominantly filled with subunits that are assigned the `cavalry` type.
num_infantry	None.	The amount of units controlled by the unit leader that are predominantly filled with subunits that are assigned the `infantry` type.
num_mechanized	None.	The amount of units controlled by the unit leader that are predominantly filled with subunits that are assigned the `mechanized` type.
num_motorized	None.	The amount of units controlled by the unit leader that are predominantly filled with subunits that are assigned the `motorized` type.
num_rocket	None.	The amount of units controlled by the unit leader that are predominantly filled with subunits that are assigned the `rocket` type.
num_special	None.	The amount of units controlled by the unit leader that are predominantly filled with subunits that are assigned the `special` type.
num_ships	None.	Amount of ships controlled by the unit leader
num_ships_with_type	Subunit type such as `@destroyer`.	Amount of ships controlled by the unit leader, ship type is defined in target, example: num_ships_with_type@carrier
num_units	None.	Amount of units controlled by the unit leader.
num_units_crossing_river	None.	Amount of units currently passing through a river.
num_units_in_combat	None.	Amount of units currently fighting under control of this unit leader.
num_units_defensive_combats	None.	Amount of units in defensive combat under control of this unit leader.
num_units_defensive_combats_on	Provincial terrain, such as `@marsh`	Amount of units that are currently defensively fighting on the specified terrain under control of this unit leader.
num_units_offensive_combats	None.	Amount of units in offensive combat under control of this unit leader.
num_units_offensive_combats_against	Provincial terrain type such as `@urban`	Amount of units that are currently offensively fighting on the specified terrain under control of this unit leader.
num_units_in_state	State.	Amount of units controlled by the unit leader within the target state.
num_units_on_climate	Acclimatization type, such as `@cold_climate`	Amount of units under control of the unit leader that are on provinces with a climate that requires the specified acclimatization type.
unit_ratio_ready_for_plan	None.	The percentage of units, on the scale from 0 to 1, under control of this unit leader that have the full planning bonus.

Operative-scoped game variables: 折叠

Variable	Target	Description/Notes

Variable	Target	Description/Notes
intel_yield_factor_on_capture	None.	The rate at which intel is extracted from this operative by an enemy country.
operation_country	None.	The country that the operative is assigned to work in. 0 if not assigned to a country.
operation_state	None.	The state that the operative is assigned to work in. 0 if not assigned to a country.
operation_type	None.	The operation that the operative to work on.
operative_captor	None.	The country that the operative is captured by.
own_capture_chance_factor	None.	The chance that the operative has to be captured.
own_forced_into_hiding_time_factor	None.	The chance that the operative has to be forced into hiding.
own_harmed_time_factor	None.	The chance that the operative has to be harmed.

Game arrays[编辑 | 编辑源代码]

Global-scoped game arrays: 折叠

Array	Description/Notes
countries	Every single country, regardless if they exist or not, including the 50 dynamic countries even if they do not have an original tag defined.
majors	Every country that is currently major.
states	Every state that exists in the game.
ideology_groups	Every ideology group that exists in the game.
operations	Every operation that exists in the game.
province_controllers	An array of controller countries for each province. In this case, the province ID serves as the index: the controller of province 1234 is province_controllers^1234.

Country-scoped game arrays: 折叠

Array	Description/Notes
allies	Countries that are considered an ally, including fellow faction members, subjects, and the overlord.
army_leaders	Army leaders recruited by the country.
enemies	Countries that the current country is at war with.
enemies_of_allies	Countries that are considered an ally of any country that is at war with the current country.
faction_members	Countries that are in the same faction as this one.
occupied_countries	Countries that are currently occupied by this one.
subjects	Countries that are subject to this one.
owned_controlled_states	States that are owned and controlled by the current country.
owned_states	States that are owned, but not necessarily controlled, by the current country.
potential_and_current_enemies	Countries that are considered enemies and that 'may' become enemies (due to being an ally of an enemy or by having a wargoal)
controlled_states	States that are controlled, but not necessarily owned, by the current country.
core_states	States that are considered national territory of the current country.
neighbors	Countries that are considered neighbours to the current country, having at least one border within controlled provinces.
neighbors_owned	Countries that are considered neighbours to the current country, having at least one owned state that borders any state that the current country owns.
researched_techs	Technologies that were researched by the current country.
navy_leaders	Navy leaders recruited by the country.
operatives	Operatives recruited by the country.
exiles	The exiled governments that the current country is hosting.

State-scoped game arrays: 折叠

Array	Description/Notes
core_countries	Countries that consider this state to be national territory.

References[编辑 | 编辑源代码]

↑ `NDefines.NGame.EVENT_TIMEOUT_DEFAULT = 13` in Defines

↑ `NDefines.NAI.FOCUS_TREE_CONTINUE_FACTOR = 1.5` in Defines

↑ `NDefines.NGame.END_DATE = "1949.1.1.1"` in Defines

↑ `NDefines.NMilitary.UNIT_EXP_LEVELS = { 0.1, 0.3, 0.75, 0.9 }` in Defines

↑ 跳转至：5.0 5.1 `NDefines.NGame.MAX_EFFECT_ITERATION = 1000` in Defines

↑ /Hearts of Iron IV/documentation/triggers_documentation