ACC 2.x
GamesData.ini - entries formatting

INI file sections

There are two kinds of sections in the GamesData.ini (further referenced only as „ini“):

- `[GamesData]` – contains only one valid value named Version. It contains version of protocol that will be used to read and interpret this ini. This value should be always set to $00010000 or 65536.
  Note – as of now, checking of this number is not implemented and reader assumes this value to be $00010000.

- `[Game.X]` – contains all necessary informations to support a particular game. X is a number written in decimal notation.
  Warning – first section of this kind must have number 0, and all subsequent sections must have increasing numbering (+1). This is so because the reading is done in following manner (simplified code):

  ```
  GamesData.Version := ini.ReadInteger('GamesData','Version');
  MaxGames := 0; // note this is error, it should be set to -1
  // get highest game number
  For i := 0 to High(Integer) do
    If not ini.SectionExists('Game.%i') then
      begin
        MaxGames := i - 1;
        BreakCycle;
      end;
  // load games
  For i := 0 to MaxGames do LoadDataForParticularGame(ini,i);
  ```

  If there are two sections with the same number, then reading behavior is not defined (it is dependent on ini files handler implementation, which at current ACC implementation is part of OS).
  Ordering of sections (numbers) inside the ini should have no effect on reading.
  There can theoretically be 2147483647 sections.
  Number of section have no inherent meaning and is not tied to any particular game or its trait, it is there only to simplify reading.

  This section can have following valid values: `Identificator`, `GameInfo`, `Process`, `CCStatus`, `CCSpeed`, `TruckSpeed`, `Special[%n]` (%n is number in decimal notation). All of these fielsd are mandatory, except for `Special`, which is optional.

  Valid values (their strings) are checked for first stored character before parsing takes place. If they start with „@“, they are treated as encrypted, otherwise they are parsed without decryption. Value `TruckSpeed` (and only this value) can be set to contain only „0“, in such case it is ignored.

  Warning – none of the values can be empty! If it is, an exception is raised and whole section is ignored.

Game.X section values formatting

Note – order of values inside section has no meaning.

`Identificator`

String containing textual identifier of the game. In current implementation, it is ignored, but should be filled nevertheless.
Should contains some information about a game it identifies.
Can have any length, but should not be longer than 64 characters (sofl limit). Use only latin alphabet and these characters: dash (-), forward slash (/), underscore (_), comma (,), period (.).
Do not use diacritics. Must be unique in entire file. Case insensitive.

Example:

```
ncs_g_18WoS_Con_1.02_en_cd_0001
```

Identifies game 18 Wheels of Steel: Convoy, version 1.02, english retail CD.

`GameInfo`

It is composed from two strings separated by two ampersands.

General structure:

```
GameName&&GameInfo
```

`GameName` contains name of a game without any other information (like version, language, etc.).

`GameInfo` contains additional informations about a game (version, language, distribution means, whether it is fully or partially supported by the ACC – see further, …).

As these texts are shown in the program window and in notification area, they should not exceed 127 characters combined.

Both these texts can contain following placeholders that will be replaced by corresponding strings from translation:

- `%v` – version
- `%ps` – partial support
- `%fs` – full support

Note – difference between partial and full support is that when the game is fully supported, ACC can access actual speed of a vehicle.

Example:

```
Euro Truck Simulator 2&&1.11.1 Multi, DD & CD %v (%ps)
```

In this example, a game is partially supported multilanguage Euro Truck Simulator 2 of version 1.11.1, distributed either on CD or trough digital distribution.

Process

This value contains informations the ACC needs to identify running game in the system.

General structure:

```
version^ModuleInfo[1]&ModuleInfo[2]&...&ModuleInfo[n]
```

...where `ModuleInfo[%n]` block is of the following structure:

```
modsize%modcrc@modfile
```

Following special characters are used as field separators: `^`, `&`, `%`, `@`.
Where `^` marks an end of `version` field, `&` separates individual blocks., `%` marks end of the `modsize` field and `@` marks end of `modcrc` field (see parsing details).

Individual blocks contains information about files that will be checked when identifying a running game. Each block for one file.
Warning – first block must contain informations for main game binary, as this file is searched for in the list of running processes.

`version` contains number that denotes how the remaining string will be parsed. At the moment, only version 0 is implemented. When unsupported version is used, an entire ini section is dropped.
Can be written either as decimal or hexadecimal (preceded by $ character) number of any length.

`modsize` contains size of checked file in bytes. This field is entirely optional and may not be present at all. File is not checked for size when set to 0.
Can be written either as decimal or hexadecimal (preceded by $ character) number of any length.

`modcrc` is CRC32 (polynomial base 0x04C11DB7) checksum of a given file. Mandatory information. File is not actually checked for CRC32 when set to 0.
Can be written either as decimal or hexadecimal (preceded by $ character) number of any length.

`modfile` is name of the checked file (including extension). Mandatory information. Can contain path relative to a main executable (listed in first block => file in the first block must not contain any path, if it does, an silent exception is raised and entire ini section is dropped). Comparisons are case insensitive.
If a file contains only data (meaning it is not a in-memory module), it must be marked by * at the end of its name (**Warning** – this feature is not correctly implemented, do not use data files) – data files are checked only on disk, if it is a module, it must be loaded in game memory before the game is binded and operated on (program waits for such modules to be loaded when searching for running games).

Parsing procedure:

1. entire string is loaded from ini
2. position of `^` mark is found, everything left of this position is parsed as version number
3. `version` and `^` are deleted from the string, what is left is passed to another parsing routine (this routine is selected based on version number)
4. --- following steps are specific for version 0 ---
5. `&` character is added at the end of string
6. blocks are counted (number of `&` characters is interpreted as numbers of blocks)

7. position of `&` mark is found, everything left of this position is copied to a temporary variable (further refered to as `TempStr`) and deleted (mark itself is deleted too)
8. `TempStr` is checked for presence of `%` mark, if present, everything left of it is parsed as number and stored as file size, then deleted (including mark itself)
9. position of mark `@` in `TempStr` is found, everything left of it is parsed as CRC32 and then deleted (including mark itself)
10. what left is checked for errors and then passed as file name, error conditions in current implementation are:
    - first block, file name contains * mark
    - file name does not contain * mark and CRC32 is equal to 0
    - first block, file name contains any path
11. repeat from 7. until there are no `&` marks in the string

Note the lack of error detection throughout parsing!

Example:

```
$0^$96719C3F@core.dll&$D8E4910A@lib\game.dll
```

In this example, there are two modules that has to be loaded in memory for a game to be recognized. The second module is located in *lib* subdirectory relatively to the main (first) module. Size is not checked in both cases.

`CCStatus`, `CCSpeed`, `TruckSpeed`, `Special[%n]` (pointers)

These values are containing informations necessary to obtain an memory address in foreign process (game process).

General structure:

```
type@module+off[1]>off[2]>...>off[n]>coefficient
```

Following special characters are used as field separators: `@`, +, >.
Where `@` marks an end of `type` field, + marks end of `module` field and > marks end of individual offsets (see parsing details).

`type` determines how the variable (value stored in game memory) pointed to by this pointer will be treated. In current implementation, it works in this way:

- When set to 0, the pointer is considered to be a „normal“ pointer – that is, when the variable is obtained from memory it is processed using `coefficient` (see further).
- When set to a value between $100 and $164 (**Note** - there is a range of 100), then number stored in `coefficient` is ignored and instead of it, a new number (32bit floating point value) is obtained from a memory address pointed to by a pointer stored in `Special[%n]`, where index %n is calculated as number stored in `type` minus $100.
  For example, when `type` is set to $105, then new coefficient is obtained from an address pointed to by a pointer stored in `Special[5]`.
  Warning - if required `Special[%n]` does not exist for a given game, the program will crash!
- When set to any other value, the pointer simply will not work, as it will be ignored by the ACC.

Can be written either as decimal or hexadecimal (preceded by $ character) number of any length.

`module` is the number of module whose base address will be used when processing a pointer. This number corresponds to a position of this module in the `Process` value (zero based). Can be written either as decimal or hexadecimal (preceded by $ character) number of any length.

`off[%n]` are individual pointer offsets, ordered from pointer static base towards an actual variable. First offset must always be a base offset – that is, an offset of pointer static base from a module base address.
Can be written either as decimal or hexadecimal (preceded by $ character) number of any length.
If there is need for a negative offset, you can write it in any of the following manner:

- -n (eg. -158, -4)
- -$n (eg. -$51, -$FE0B)
- full-length hexadecimal representation of 32bit signed integer with sign bit set (eg. $FFFFFFF6, $8000078A)

`coefficient` is a number that will be used while processing a value obtained from game memory. At the moment it used in this way:

- for a `CCStatus` value, it is completely ignored (but must be included)
- for a `CCSpeed, TruckSpeed` and `Special[%n]` values:
  - when the value is read from the memory, it is **DIVIDED** by this coefficient before further processing (this is to ensure that obtained value is in expected units of measurement)
  - when the value is written to the memory, it is **MULTIPLIED** by this coefficient (again to ensure it will be in units expected by the game)

It must be written in hexadecimal notation without an $ character.
WARNING - never set value of the `coefficient` to 0! If you don't know what to write there, use value of 3f800000 (1.0).

Parsing procedure:

1. entire string is loaded from ini
2. position of `@` mark is found, everything left of this position is parsed as type number and deleted from the string (including the mark)
3. position of + mark is found, everything left of this position is parsed as module index and deleted from the string (including the mark)
4. offsets are counted (number of > characters is interpreted as numbers of offsets)
5. position of first > mark is found, everything left of this position is parsed as an offset and added to the array of offsets, offset is deleted from the string (including the mark)
6. repeat from 5. count-times (see 4.)
7. what is left is parsed as `coefficient`

Note the lack of error detection throughout parsing!

Example:

```
$100@$1+$4cbf4>$1f4>$3c0>$20>$80>$b0>3f800000
```

It is an 5-level pointer (base offset does not count), base address is taken from second module and coefficient is ignored – it is instead loaded from memory pointed to by a pointer stored in `Special[0]`.