Gamer_Z eXtreme Party 24/7
C++ GameMode

The eXtreme and Fun Modern DeathMatch Mode

Project Lead	Rafał ‘Gamer_Z’ Grasman
Project Contributors	Tom 'IceCube' Hewlett Mariusz 'C2A1' K. Mateusz 'eider' Cichoń
Last Revision	28 February 2015
Published	28 February 2015
Project URL	https://github.com/grasmanek94/eXe

All credits go to the respective owners and/or contributors.

This page has been intentionally left blank.

Table of Contents

Introduction

The purpose of this document is to help new co-developers familiarize themselves with the GameMode they are going to work on. The usage and workings of the most important parts of the code will be explained here. The project is mainly written in C++, but also has parts of the code which contain C code. This guide assumes the co-developer has all the needed knowledge about the C++ Programming Language to understand the code and/or be able to do research on his/her own before asking questions.

Project Layout

To keep projects manageable, a layout for all the files will be used to ease in the search for documents and information.

Physical Layout

The physical layout decides how to physical files for the project are organized. The following layout is being used:

```
+---bin
|   +---Svr
|   |   +---logs
|   |   +---npcmodes
|   |   \---scriptfiles
|   \---tools
+---boost_extract
|   +---boost
|   \---libs
+---lib
|   +---Eigen
|   +---glm
|   \---sqlite
\---src
    +---gamemode
    +---gtasa
    +---libodb
    |   \---odb
    +---sampgdk
    \---streamer
```

root: The solution, automation, versioning, update, resource, readme and documentation.
bin: contains all the executables, data files and all other related files (for example, debug databases).
Svr: contains the server executable, database, the gamemode dll file and all server related files (configs, modes, etc).
logs: contains all the logs generated by the function `gtLog`.
npcmodes: contains the idle bot (which I named 'DoucheBag' because he was causing lots of troubles) used for displaying custom text on the death list.
scriptfiles: contains all the files that are created by external filterscript (eg. TextDraw editor)
tools: contains the C++ ODB executables (mingw compiler) for generating the needed database code. Although this folder is not directly available via the source repository, it can be downloaded fomr the C++ ODB page by CodeSynthesis.
boost_exctract: contains the needed boost libraries, extracted with *bcp* tool.
boost: contains all the headers for the used boost libraries.
libs: contains the needed source files for libraries in boost that don't support the header-only directives.
lib: contains external libraries that are used in the mode (**sqlite3, glm, Eigen**)
src: this is the main folder in which the source code related to the GameMode in one way or another. It contains all the sub-directories which hold the source and header files.
gamemode: contains all the files directly related to the GameMode.
gtasa: contains code that can be used (in)directly in other projects.
libodb\odb: contains the database library source files, headers and related files (C++ ODB).
sampgdk: contains the SampGDK (by Zeex) source and header files (including AMX SDK).
streamer: contains a modified Streamer Plugin by Incognito for this mode (V 2.7.4).

Solution Layout

The solution uses filters to make searching for a specific item easier (and hopefully more logical). The following layout is currently being used:

```
+---boost_src
+---DLL
+---GTASA
|   +---CarColor
|   \---Math
+---libodb
|   +---ODB-MAIN
|   +---ODB-SQLITE
|   \---SQLITE3
+---packethook
|   +---exe24+
|   +---Hook
|   \---RakNet
+---PawnGameMode
+---SAMP
|   +---GDK
|   |   \---Helper
|   \---SDK
|       \---amx
+---streamer
+---TESTS
```

boost_src: contains all the boost source file for compilation (don't use PCH in these files!).
DLL: contains all the files related to the generated DLL file (don't use PCH in these files!).
GTASA: contains the files required to generate real GTA Vehicle colors (instead of totally random ones, don't use PCH in these files!). The CarColor folder contains the CarColor generator and the Math folder contains the random class that is used for pseudorandom numbers.
libodb: contains all the database library source and header files. MAIN contains the C++ ODB base source, SQLITE contains the C++ ODB sqlite part and SQLITE3 contains the needed SQLite files so ODB will work (it does need the SQLite library).
packethook: contains the low-level fixes / security measures, exe24+ headers to control it from the server side and a part of the RakNet source and header files required to operate on the SA-MP networking layer.
PawnGameMode: here is the old eXe24 GameMode, which can be used to extract various needed information from.
SAMP: contains the Samp**GDK** library source and header files, AMX **SDK** and (**Helper**) the _GetPlayerName and _GetPlayerIP functions (to be used once, e.g. in OnPlayerConnect).
streamer: contains the edited Incognito Streamer plugin for this mode along with a custom header to be able to use it's functions.
TESTS: all tests are located here (to confirm the working of all functions).

```
\---gamemode
    +---Achievements
    |   +---GoldCoins
    |   +---Manager
    |   +---StatsUpdate
    |   \---TextDraw
    +---Administrator
    +---AntiCheat&Protections&Fixes
    |   +---AntiCheat
    |   +---RCFix
    |   \---SCMCrashFix
    +---CommandDialogSystem
    |   +---Command
    |   +---Dialog
    |   \---Menu
    +---Database
    |   +---GM
    |   \---ODB
    +---Language
    |   +---Languages
    |   \---System
    +---Libs
    |   +---Extension
    |   +---MapAndreas
    |   \---Whirlpool
    \---MiniGameSystem
        +---Arena
        +---MiniGame
        +---Race
        \---System
```

gamemode: contains all the files directly related to the working of the mode.
Achievements: contains a collection of various files related to statistics and player achievements.
GoldCoins: contains the Golden Coins system, coin locations, and all the code responsible for creating and handling them.
Manager: contains the code for storing and accessing achievements of a player.
StatsUpdate: contains the code that is responsible for sending player and mafia statistics to the website.
TextDraw: contains the TextDraw for the player statistics (id, name, K, D, K/D, Mafia) and the needed code to update the stats TD.
Administrator: contains all commands, code and systems needed for the Staff to do their work (spectate system, admin commands, permission manager, etc).
AntiCheat&Protections&Fixes: Contains all AntiCheat, fixes, protections, patches and related code.
AntiCheat: obviously, contains the AntiCheat system itself.
RCFix: Remote Controlled vehicle exit fix (players couldn't exit these vehicles).
SCMCrashFix: crashfix for single '%'s in Send*Message* functions.
CommandDialogSystem: contains all the code for the **command**-, **dialog**- and **menu** processors.
Database: contains all the **gamemode** related database code (and auto-generated **ODB** code).
Language: contains all the code for the language **system** and the translations in the **languages** filter.
Libs: contains the **Extension, MapAndreas** and **Whirlpool** libraries for the gamemode.
MiniGameSystem: contains all the code for the mini-game **system**, **mini-game**s, **arena**s', race **system** and **races**.

```
\---gamemode
    \---Internals
        +---_Testing
        +---E-Mail
        +---Gates
        +---GlobalVehicleManager
        +---HostBan
        +---House
        +---Logging
        +---Lotto
        +---Other
        +---PerfMon
        +---Sentry
        +---Tutorial
        +---Vehicle
        |   \---Native
        \---Versioning
```

Internals: contains code related to the GameMode but not related to the player (should probably be re-categorized).
_Testing: any experimental code goes here.
E-Mail: the system responsible for sending e-mails.
Gates: gate base class / system allowing easy creation of gates.
GlobalVehicleManager: Vehicle system replacement / a global manager (not used).
HostBan: the system responsible for managing and creating host bans.
House: the house system.
Logging: the logging system.
Lotto: the lotto system.
Other: all uncategorized code, if you can't find something, looking here is your best bet.
PerfMon: performance monitor (cross-platform) and custom sleep manager (windows only).
Sentry: sentry gun system (though sentry guns won't be used anymore, probably).
Tutorial: this would be the tutorial system, this is unused.
Vehicle: all code related to vehicles, vehicle positions, extra vehicle functions, etc**,** the native filter contains the GTA:SA car spawns.
Versioning: automatic gamemode versioning.

```
\---gamemode
    +---Player
    |   +---Account&Permissions
    |   +---Commands
    |   +---Information
    |   +---Mafia
    |   +---Teleport
    |   |   +---Global
    |   |   \---Local
    |   +---TextDraw
    |   |   +---CountDown
    |   |   +---Speedo
    |   |   \---TimedInformation
    |   +---Vehicle
    |   +---Weapon
    |   \---WorldData
    \---World
```

Player: contains all files that are directly related to the player or handling player data / information.
Account&Permissions: contains the login and register system, also enforces command permissions.
Commands: contains commands that are usable by players.
Information: contains the help commands, gamemode info for the player, credits scene and hitman list.
Mafia: contains the mafia system.
Teleport: contains global teleport commands and player-specific (local) commands like /goto.
TextDraw: Contains the **CountDown**, **Speedo** and **TiP** systems, the main function is displaying information by means of TextDraws.
Vehicle: contains vehicle related commands and additional player specific vehicle functions.
Weapon: contains the weapon equipment system for the layer and a weapon information table used across the gamemode (mainly in the **AntiCheat**).
WorldData: contains the player structure, array and player data.
World: contains environment data (objects, map icons, pickups, etc., anything that is in the mode but has no function whatsoever).

Code Structure and Coding Style

The project requires multiple programmers to co-operate together to create a fully functioning program. Because each person has their own personal preferences as to coding structure and style, a few simple rules to follow while coding will be presented in this chapter.

Main Boilerplate License Text

All files that contain code must include the projects' copyright notice / header:

```
/*
	Copyright (c) 2014 - 2015 Rafał "Gamer_Z" Grasman
		* See LICENSE.md in root directory for license
		* Written by Rafał Grasman grasmanek94@gmail.com
		* Development Start Month-Year: October-2014

	_____________________________________________________________________
	Purpose of this file

	_____________________________________________________________________
	Notes

	_____________________________________________________________________
	Dependencies

	_____________________________________________________________________
	Project Contributors
		Tom 'IceCube' Hewlett
		Mariusz 'C2A1' K.
		Mateusz 'eider' Cichoń
*/
```

To apply copyright to other files use alternative ways (properties, details, comments, binary signing).

Header Files

In general, each source file needs a header file. In this project however we use extensions which are the gamemode logic, that use functions provided by other source files. Only the source files that provide functions that can be used by other components in the gamemode should have a corresponding header file with the prototypes that should be exposed.

Include Guards

To prevent multiple inclusion of header files, use the `#pragma once` directive.

Include Order

1. “gamemode.hxx” - main PCH file
2. C system files;
3. C++ system files;
4. Other libraries' header files;
5. Your project's header files;

Preferably, all includes have to be placed in “gamemode.hxx”, this is the main header which all source and header files **must** include as their first include file (libraries and external code are exceptions from this rule).

Examples

Empty header:

```
/*
	Copyright (c) 2014 - 2015 Rafał "Gamer_Z" Grasman
		* See LICENSE.md in root directory for license
		* Written by Rafał Grasman grasmanek94@gmail.com
		* Development Start Month-Year: October-2014

	_____________________________________________________________________
	Purpose of this file

	_____________________________________________________________________
	Notes

	_____________________________________________________________________
	Dependencies

	_____________________________________________________________________
	Project Contributors
		Tom 'IceCube' Hewlett
		Mariusz 'C2A1' K.
		Mateusz 'eider' Cichoń
*/
#pragma once
#include "GameMode.hxx"
```

Empty source file:

```
/*
	Copyright (c) 2014 - 2015 Rafał "Gamer_Z" Grasman
		* See LICENSE.md in root directory for license
		* Written by Rafał Grasman grasmanek94@gmail.com
		* Development Start Month-Year: October-2014


	_____________________________________________________________
	Purpose of this file


	_____________________________________________________________
	Notes


	_____________________________________________________________
	Dependencies


	_____________________________________________________________
	Project Contributors
		Tom 'IceCube' Hewlett
		Mariusz 'C2A1' K.
		Mateusz 'eider' Cichoń
*/
#include "GameMode.hxx"
```

Function Parameter Ordering

When defining a function, parameter order is: inputs, then outputs (and then, if applicable: variable arguments).

Namespaces

Do not use unnamed namespaces in header files.
The *using* directive is **prohibited** for namespaces (the Language namespace is an exception to this rule).

Nested Classes/Structures

Only allowed when the nested class and/or structure is private and only used in the main structure/class.

Nonmember, Static Member, and Global Functions

Prefer nonmember functions within a namespace or static member functions to global functions; use completely global functions rarely. This avoids the pollution of the global namespace.

Local Variables

Place a function's variables in the narrowest scope possible, and initialize variables in the declaration if possible. C++ allows you to declare variables anywhere in a function. We encourage you to declare them in as local a scope as possible, and as close to the first use as possible. This makes it easier for the reader to find the declaration and see what type the variable is and what it was initialized to. In particular, initialization should be used instead of declaration and assignment, e.g.:

```
int i;
i = f();      // Bad -- initialization separate from declaration.

int j = g();  // Good -- declaration has initialization.

vector<int> v;
v.push_back(1);  // Prefer initializing using brace initialization.
v.push_back(2);

vector<int> v = {1, 2};  // Good -- v starts initialized.
```

Variables needed for if, while and for statements should normally be declared within those statements, so that such variables are confined to those scopes. E.g.:

```
while (const char* p = strchr(str, '/')) str = p + 1;
```

There is one caveat: if the variable is an object, its constructor is invoked every time it enters scope and is created, and its destructor is invoked every time it goes out of scope.

```
// Inefficient implementation:
for (int i = 0; i < 1000000; ++i) {
  Foo f;  // My ctor and dtor get called 1000000 times each.
  f.DoSomething(i);
}
```

It may be more efficient to declare such a variable used in a loop outside that loop:

```
Foo f;  // My ctor and dtor get called once each.
for (int i = 0; i < 1000000; ++i) {
  f.DoSomething(i);
}
```

Static and Global Variables

Allowed. The ZCMD system uses this to initialize the commands. Make sure that the order of initialization does not matter if you initiate a global / static class.

Exceptions

We do not use C++ exceptions. We do handle all exceptions by libraries if they do not provide an exception-less interface.

Preincrement and Predecrement

Use prefix form (++i) of the increment and decrement operators with iterators and other template objects.

Use of const

Use const whenever it makes sense (data is never changed / initialized only once).

Preprocessor Macros

Be very cautious with macros. Prefer inline functions, enums, and const variables to macros.

Macros mean that the code you see is not the same as the code the compiler sees. This can introduce unexpected behavior, especially since macros have global scope.

Luckily, macros are not nearly as necessary in C++ as they are in C. Instead of using a macro to inline performance-critical code, use an inline function. Instead of using a macro to store a constant, use a const variable. Instead of using a macro to "abbreviate" a long variable name, use a reference.

Macros can do things these other techniques cannot, and you do see them in the codebase, especially in the lower-level libraries. And some of their special features (like stringifying, concatenation, and so forth) are not available through the language proper. But before using a macro, consider carefully whether there's a non-macro way to achieve the same result.

The following usage pattern will avoid many problems with macros; if you use macros, follow it whenever possible:

- Don't define macros in a header file.
- #define macros right before you use them, and #undef them right after.
- Do not just #undef an existing macro before replacing it with your own; instead, pick a name that's likely to be unique.
- Try not to use macros that expand to unbalanced C++ constructs, or at least document that behavior well.
- Prefer not using ## to generate function/class/variable names.

Break the rules only as a last resort: “*follow it whenever possible*”.

0 and nullptr/NULL

Use 0 for integers, 0.0 for reals, nullptr for pointers (**not** NULL!), and '\0' for chars.

sizeof

Prefer sizeof(varname) to sizeof(type).

Use sizeof(varname) when you take the size of a particular variable. sizeof(varname) will update appropriately if someone changes the variable type either now or later. You may use sizeof(type) for code unrelated to any particular variable, such as code that manages an external or internal data format where a variable of an appropriate C++ type is not convenient.

auto

Use auto to avoid type names that are just clutter. Continue to use manifest type declarations when it helps readability, and never use auto for anything but local variables.

Braced Initializer List

You may use braced initializer lists.

Lambda expressions

Use of lambda expressions is allowed. Do not use default lambda captures; write all captures explicitly.

Boost

Whenever possible, use boost, don't reinvent the wheel. If you need a boost library that is not in available in the project folder use the bcp tool to extract the needed files from boost.

C++11

Use libraries and language extensions from C++11 (formerly known as C++0x) when appropriate.

General Naming Rules

Function names, variable names, and filenames should be descriptive; avoid abbreviation.

Give as descriptive a name as possible, within reason. Do not worry about saving horizontal space as it is far more important to make your code immediately understandable by a new reader. Do not use abbreviations that are ambiguous or unfamiliar to readers outside your project, and do not abbreviate by deleting letters within a word.

```
int price_count_reader;    // No abbreviation.
int num_errors;            // "num" is a widespread convention.
int num_dns_connections;   // Most people know what "DNS" stands for.

int n;                     // Meaningless.
int nerr;                  // Ambiguous abbreviation.
int n_comp_conns;          // Ambiguous abbreviation.
int wgc_connections;       // Only your group knows what this stands for.
int pc_reader;             // Lots of things can be abbreviated "pc".
int cstmr_id;              // Deletes internal letters.
```

File Names

File names should be descriptive but short (one / two words). Source files extension: cxx, header file extension: hxx. Comply with the project physical and solution file structure.

Type Names

Type names start with a capital letter and have a capital letter for each new word, with no underscores: MyExcitingClass, MyExcitingEnum.
The names of all types — classes, structs, typedefs, and enums — have the same naming convention. Type names should start with a capital letter and have a capital letter for each new word. No underscores.

Variable Names

The names of variables and data members are all lowercase, with underscores between words. Data members of classes (but not structs) additionally have trailing underscores. For instance: a_local_variable, a_struct_data_member, a_class_data_member_.

Variable and Array Initialization

Your choice of =, (), or {}.

Inheritance

The list of inherited base classes should be on the same line as the class name (unless the list is really long):

```
	class AntiSobeitProcessor : public Extension::Base
```

Replacing/overriding base class functions

Always use the override keyword! Use it at the end of the line:

```
	bool OnPlayerStateChange(int playerid, int newstate, int oldstate) override
```

Braces

Never omit braces. Always place them on a new line. **Don't do this:**

```
		if (Player[playerid].BanImmunity)
			PlayerAllowedIntoGame(playerid, true);
```

Do this instead:

```
		if (Player[playerid].BanImmunity)
		{
			PlayerAllowedIntoGame(playerid, true);
		}
```

Existing Non-conformant Code

Because these rules have been introduced almost after half a year into the project, it is possible not all code will follow this rules. Please refactor code where appropriate. From this point on use these rules.

Parting Words

Use common sense and *BE CONSISTENT*.
If you are editing code, take a few minutes to look at the code around you and determine its style. If they use spaces around their if clauses, you should, too. If their comments have little boxes of stars around them, make your comments have little boxes of stars around them too.

Server Design

Modularity

Almost all of the code that is in the project is specifically designed for the server, so we drop the 'reusable classes' requirement, but we still try to achieve that with some classes (e.g. in libraries).

The mode is composed of "extensions" (or modules, if you prefer to call it so, each extension has to have its own source file), which contain the logic for what they are doing, and sometime provide header files to make information from within the extension available to the rest of the mode. Because SA-MP is event-driven, each extension can contain any callback that it requires for it to work and do its job. Extensions can interrupt the execution of the callback chain by returning non-default values for callbacks that handle return values (it's also possible for some callbacks that don't handle return values by sa-mp specifications).

Extensions are not limited to itself, it is allowed to create extensions that have their data in other extensions or depend on other extensions. Extensions also have an *execution priority* which tells the extensions library in which order to process all the callbacks (highest priority == first execution). The visualization is available in figure 1. This is the absolute core of how the mode works.

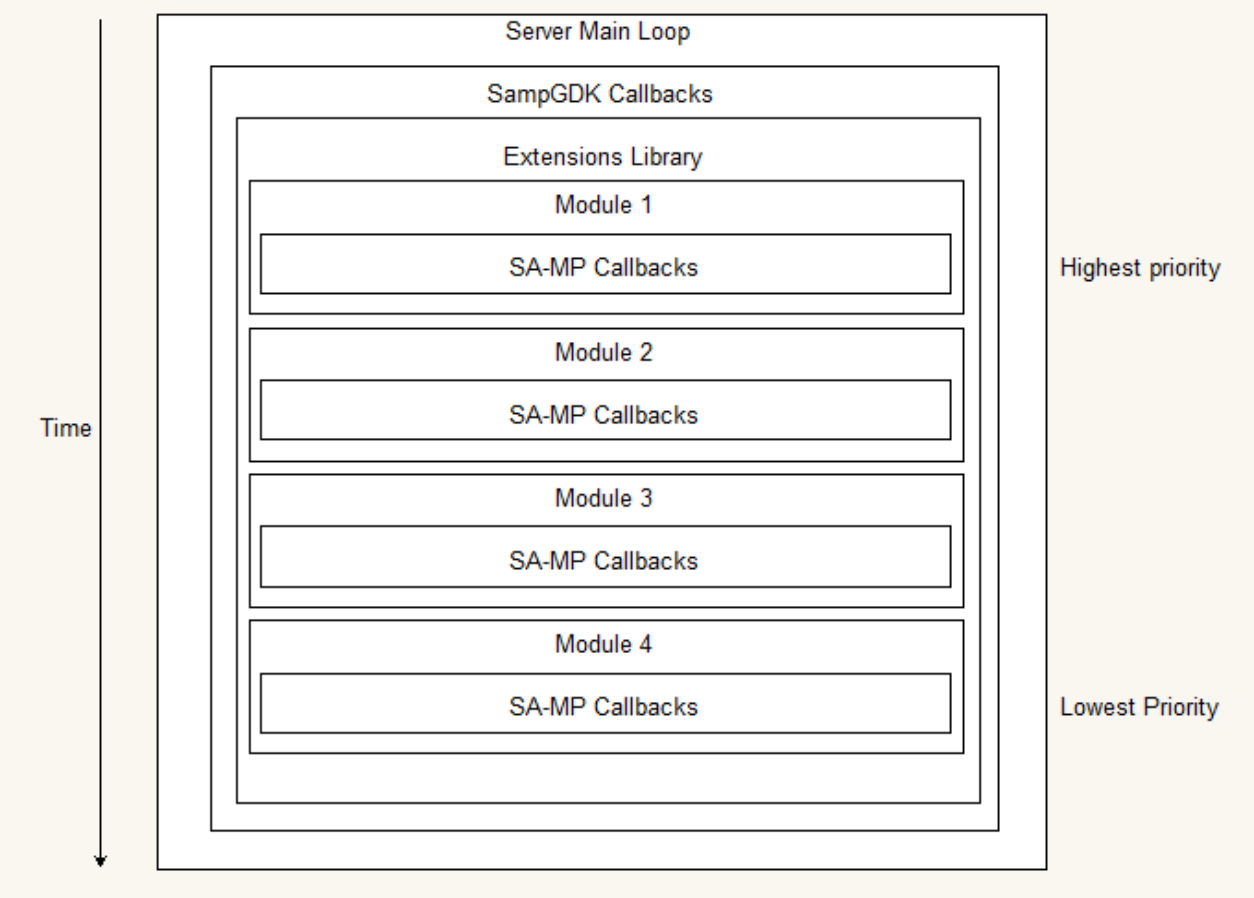

Figure 1 - Visualization of the main design of the server

Code

To make use of the extension system, create the template source file (see section Code Structure and Coding Styles - Examples). When that is done create an empty class and inherit from the extension base:

```
#include "GameMode.hxx"

namespace MyExtensionName
{
	//Inherit from extension base to make callbacks work
	class MyExtensionClassName : public Extension::Base
	{
	public:
		//You can define the priority in the Base constructor
		MyExtensionClassName() : Base(<PRIORITY>)
		{}

		//override any callbacks you like
		bool OnPlayerDisconnect(int playerid, int reason) override
		{
			return true;
		}
	}
	//This is needed so your code will actually EXIST and have an INSTANCE in memory
	MyExtensionClassNameVariable;
};
```

Callbacks Handling Return Values

The list of callbacks which interrupt the execution chain when the specified value is returned:

- `OnPlayerRequestClass` `false`
- `OnRconCommand` `true`
- `OnPlayerSpawn` `false`
- `OnPlayerDeath` `false`
- `OnPlayerText` `false`
- `OnPlayerRequestSpawn` `false`
- `OnPlayerUpdate` `false`
- `OnPlayerWeaponShot` `false`
- `OnPlayerCommandReceived` `false`

A list of all available override-able callbacks is located in ‘extension.hxx’

Database

The database system used in the mode is SQLite3, though it has been abstracted away by the C++ ODB library from codesynthesis. This library allows us to easily switch between database systems (e.g. to MySQL). The server utilizes threading to create a thread where the queries are being executed, the server main loop fetches the results from a thread-safe container. The visualized design is available in figure 2.

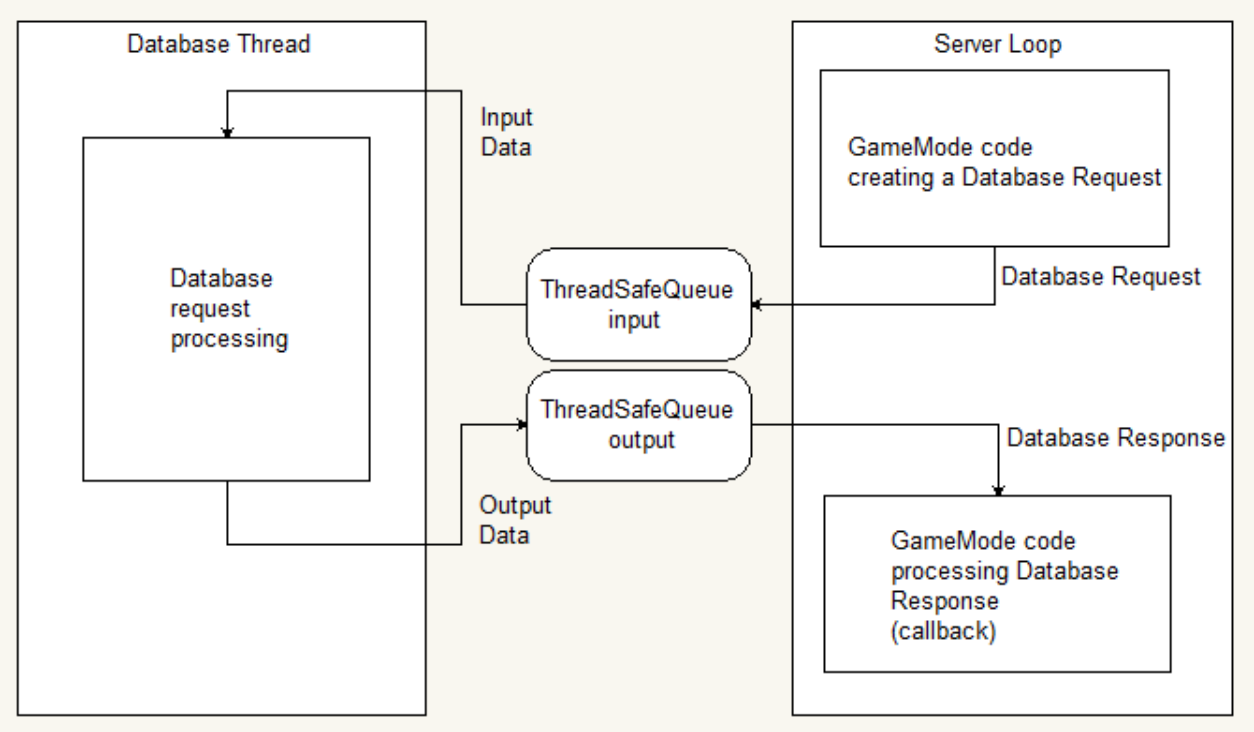

Figure 2 - Database processing chart

You can create database requests with the function `CreateWorkerRequest` and listen for the result with callback `DatabaseOperationReport`. You can find all the needed information in the 'Worker.hxx' file.

Commands

The gamemode does not directly use the OnPlayerCommandText callback. Instead commands are handled by the ZeroCMD++ library, which allows easy creation of commands and access management.

- `OnPlayerCommandReceived`
- `OnPlayerCommandExecuted`
- `OnUnknownCommand`

The library also provides boilerplate definitions to create commands easily:

- `ZCMD` - simple command with permission access and aliases
- `ZCMD3` - simple command with permission access
- `ZCMDF` - command with permission access, aliases and parameter parsing

If you are ever in the need to call another command like the player would submit a command, use `ZCMD_CALL_COMMAND`, you can also check if a command exists with the function `ZCMD_COMMAND_EXISTS`.

Note: Never create a command in a header file!

To create a command, you have to decide which features you need (ZCMD, ZCMD3 or ZCMDF). For example, to create a command with all features the following code will be put in a source file:

```
ZCMDF(
	somecommand, //command name
	PERMISSION_VIP, //lowest required permission to execute command
	RESTRICTION_ONLY_ON_FOOT | RESTRICTION_NOT_IN_A_GAME, //command usage restrictions
	(when? etc) [For command permissions and restrictions see 'CommandPermissions.hxx']
	cmd_alias({ "/somecommand2", "/somecommand3" }),//other names for this command
	"wD")//the parsing format to parse arguments (see 'CommandParamsParser.hxx')
{

	//If the code reached this line the player:
	//	(used command /somecommand or /somecommand2 or /somecommand3) and;
	//	the player is on foot and;
	//	not in a mini-game and;
	//	has the rank 'VIP'

	if (parser.Good() == 2) //successfully parsed 2 parameters
	{	//the provided parameters are in the correct "wD" format
		std::string first_param = parser.Get<std::string>(0);  //Get first argument
		long        second_param    = parser.Get<long>(1); //Get second argument
		//Do something
		return true;
	}
	//Parsing failed
	return true;
}
```

Dialogs

Dialogs, like commands, are handled by a separate library named 'DialogProcessor'.

To create, show and use dialogs, see the following code:

```
//Player has responded to dialog with the name "mydialog_a"
ZERO_DIALOG(mydialog_a)
{
	if (response)
	{
		//clicked left (first) button
		ShowPlayerCustomDialog(playerid, "mydialog_b", DIALOG_STYLE_MSGBOX,
			"Caption", "Info", "Button1", "Button2");
	}
	else
	{
		//canceled or clicked right button
	}
}

//Player has responded to dialog with the name "mydialog_b"
ZERO_DIALOG(mydialog_b)
{

}
```

Language System

The mode features build-in support for multilingual gameplay. It is not allowed to put language-specific strings into the main code. Languages are separated from the mode and each language has its own header file (see LangEnglish.hxx and LangPolish.hxx).

To add a string which you can use in, for example, SendClientMessage, you need to do the following steps:

1. Add a unique string identifier to the language enumerator in 'LangMain.hxx':

```
enum language_string_ids
{
	L_goldcoin_foundnew,
	L_goldcoin_helpmsg,
	L_my_new_string_identifier, //name of your new string identifier
	//------------------
	MAX_LANGUAGE_STRINGS
	//------------------
};
```

2. Translate (you do it or someone else) your string into all language files (see 'LangEnglish.hxx').
3. Translate your new string by using `Translate`, `TranslateP`, `TranslateF`, `TranslatePF` in a function that accepts a string or in a SendclientMessage directly:
   `SendClientMessage(playerid, Color::COLOR_ERROR, L_invalid_playerid);`

If you wish to add a new language, create a new file based on 'LangEnglish.hxx' and include it at the end of 'LanguageSystem.cxx'.

Parting Words

Not all subsystems have been explained. The purpose of this document is to get you up & running in a few minutes to quickly create your additions to the gamemode. Might you still have any questions, visit the GitHub page of this project and ask us for any explanations you want that this document didn't write about. If you wish to create more advanced extensions in the mode, it is recommended to study other code in the mode to learn from it. Some materials (house locations/data, databases and golden coin locations) are not available in this repository - this is because the items are very server-specific and could easily be abused without any option to mitigate abuse.

Libraries, Authors & Credits

This GameMode uses lots of libraries, all the authors, contributors and credits are where they belong and given to whom deserve it. Nonetheless, this document shall give credit to everyone that deserves it too (sorry if I forgot about anyone! Contact me to fix this).

Library	Author(s)
AntiCheat V2.6	**Gamer_Z**
Boost	**Boost.org**
ODB	**Code Synthesis**
MapAndreas Plugin	**Kalcor**
MySQL	**Oracle**
SampGDK	**Zeex**
SampGDK Ext Mgr	**Gamer_Z**
sqlite3	**sqlite.org**
Streamer Plugin	**Incognito**
Whirlpool Plugin	**Y_Less**
ZeroCMD++	**Gamer_Z**
ZeroDLG	**Gamer_Z**
SA-MP	**sa-mp.com**

Additional project contributors, special thanks to, and people & teams/communities I would like to mention (no specific order):

IceCube, Oski20, C2A1, West, BlackPow3R, Arcade, Kurta999, Y_Less, Kalcor, Incognito, Kar, eider, bartekdvd, dugi, JernejL, C2A1, imzdx, _EvO_X_, Megass, Energ, V3X_, Damianos, Leito, PTSRP.pl-team, Boost.org-team, codesynthesis-team, Bucho, KaSkA3eR, KoX, PlayBoY_, Zeex, Abagail, Pottus, GWMPT, Ryder`, Mauzen, ikkentim, SA-MP-team, Danny, Bible, Henkie, Spydah, Jstylezzz, Hiddos, mamoru, mione, legodude, jessejanssen, Kwarde, shadowdog, Tomboeg, justsomeguy, Michael@Belgium, Vince, mickos, Wesley221, Jantjuh, Duck, rbN., Epic_Mickey, milanosie, [MM]IKKE, sjvt, Biesmen, BlackBank3, Gforcez1337, Infinity, Kindred, corne, Kellicia, alpha500delta, Thresh, Giannidw, Karting06, Jochemd, Rzzr, Jansish, Nick_Oostrum, playbox12, Satori_Komeiji, swell, Admigo, RealCop228, MP2, PT, Lorenc_, Nero_3D, BigETI, NaS, Magda_, Natka, the sa-mp-community