NSMB Co-op Development Guide

Introduction

This guide explains how to write code that works correctly with the NSMB Co-op hack. The co-op system allows two players (Mario and Luigi) to play simultaneously on separate consoles connected via local wireless.

The biggest challenge in co-op development is preventing **desyncs** - situations where the two consoles have different game states. This document covers common desync patterns and how to avoid them, along with co-op-specific systems like player spectating.

> **Key Principle:** Any code that affects gameplay state must produce identical results on both consoles, regardless of which console is running it.

Table of Contents

1. Core Concepts - Understanding desyncs and co-op fundamentals
2. Common Anti-Patterns to Avoid - Quick reference of what NOT to do
3. Understanding Desyncs: A Detailed Example - Step-by-step desync analysis
4. Co-op-Safe Patterns and Solutions - Practical coding techniques
5. Special Cases - Exceptions and edge cases
6. Debugging and Troubleshooting - Tools and techniques for finding issues
7. Advanced Systems - Player spectating and complex features

Core Concepts

What is a Desync?

A desync occurs when the two consoles have different game states. For example:

- Console 1 thinks Mario has 3 lives, Console 2 thinks Mario has 2 lives
- Console 1 shows an enemy as alive, Console 2 shows it as defeated
- Console 1 has a different random number sequence than Console 2

Safe vs. Unsafe Operations

- **Safe**: Operations that only affect local display/audio (sounds, UI elements, screen shaking)
- **Unsafe**: Operations that modify gameplay state (player health, enemy behavior, item spawning, particle effects)

The Golden Rule

When writing gameplay logic, always consider: "What happens if both consoles run this code at the same time?"

Common Anti-Patterns to Avoid

Before diving into specific solutions, here are the most common mistakes that cause desyncs:

□ DON'T: Use Game::localPlayerID for gameplay logic

```
// This will desync!
if (shouldTriggerEvent()) {
    Player* player = Game::getPlayer(Game::localPlayerID);
    player->giveReward();
}
```

✅ DO: Loop through all players or use linkedPlayerID

```
// This stays in sync!
if (shouldTriggerEvent()) {
    for (s32 playerID = 0; playerID < Game::getPlayerCount(); playerID++) {
        Player* player = Game::getPlayer(playerID);
        if (playerMeetsCondition(player)) {
            player->giveReward();
        }
    }
}
```

❌ DON'T: Use ViewShaker without playerID parameter

```
// This will desync!
ViewShaker::start(type, viewID);
```

✅ DO: Specify which player should feel the shake

```
// This stays in sync!
ViewShaker::start(type, viewID, playerID, false);
```

❌ DON'T: Use Game::getRandom() for gameplay logic

```
// This will desync!
if ((Game::getRandom() & 0xFF) == 0) {
    spawnEnemy();
```

```
}
```

✅ DO: Use Net::getRandom() for synchronized randomness

```
// This stays in sync!
if ((Net::getRandom() & 0xFF) == 0) {
    spawnEnemy();
}
```

Understanding Desyncs: A Detailed Example

Let's examine how a typical desync occurs using a Goomba collision example:

Let's pretend this is how a Goomba is coded to hurt a player:

```
void Goomba::hurtPlayer() {
    // Game::localPlayerID is the ID of the player for *our* console
    s32 playerID = Game::localPlayerID;

    // Game::getPlayer(id) gives us a pointer to a Player object
    // id = 0 → Mario
    // id = 1 → Luigi
    Player* player = Game::getPlayer(playerID);

    // The local player gets hurt
    player->getHurt();

    // Problem:
    // On Console 0 → local_player_id = 0 → only Mario gets hurt
    // On Console 1 → local_player_id = 1 → only Luigi gets hurt
    //
    // Bad result:
    // Console 0 sees:
```

```
    //   Mario = HURT
    //   Luigi = NOT HURT
    //
    // Console 1 sees:
    //   Mario = NOT HURT
    //   Luigi = HURT
}
```

The Solution

The fix is to use the collision information that's already available:

Usually when a collision with an enemy occurs, the actor is informed of which player collided with it. This information is stored in `this->linkedPlayerID`.

```
void Goomba::hurtPlayer() {
    // this->linkedPlayerID is the ID of the player that collided with the Goomba
    // Let's assume it was Mario (0)
    s32 playerID = this->linkedPlayerID;

    // Game::getPlayer(id) gives us a pointer to a Player object
    // id = 0 → Mario
    // id = 1 → Luigi
    Player* player = Game::getPlayer(playerID);

    // The player that collided with the Goomba gets hurt
    player->getHurt();

    // Good result:
    // Console 0 sees:
    //   Mario = HURT
    //   Luigi = NOT HURT
    //
    // Console 1 sees:
    //   Mario = HURT
    //   Luigi = NOT HURT
}
```

Alternative: Player Loop Pattern

When `this->linkedPlayerID` isn't available, use the player loop pattern:

```
bool Goomba::shouldPlayerGetHurt(Player* player) {
    // ... do any checks to decide if Player should get hurt

    // In this example we assume Mario (0) is in love with the Goomba
    return player->isInLoveWithGoomba(this);
}

void Goomba::hurtPlayer() {
    // Update the logic for all players
    for (s32 playerID = 0; playerID < Game::getPlayerCount(); playerID++) {
        // Game::getPlayer(id) gives us a pointer to a Player object
        // id = 0 → Mario
        // id = 1 → Luigi
        Player* player = Game::getPlayer(playerID);

        if (shouldPlayerGetHurt(player)) {
            // The player that collided with the Goomba gets hurt
            player->getHurt();
        }
    }

    // Good result:
    // Console 0 sees:
    //   Mario = HURT
    //   Luigi = NOT HURT
    //
    // Console 1 sees:
    //   Mario = HURT
    //   Luigi = NOT HURT
}
```

Co-op-Safe Patterns and Solutions

Player Targeting: Finding the Right Player

Use `ActorFixes_getClosestPlayer(this)` instead of `Game::getLocalPlayer()` or `Game::getPlayer(Game::localPlayerID)`.

```
void Volcano::spawnMeteor() {
    // BAD: Always targets the local player
    Player* target = Game::getLocalPlayer();

    // GOOD: Finds the closest player to the volcano
    Player* target = ActorFixes_getClosestPlayer(this);

    // Spawn meteor at target's position
    Vec3 meteorPos = target->position;
    Actor::spawnActor(METEOR_ID, 0, &meteorPos, nullptr, nullptr, nullptr);
}
```

For zone-specific targeting:

```
void SpikeBass::attack() {
    // Find the closest player in a specific zone
    Player* target = ActorFixes_getClosestPlayerInZone(this, zoneID);
    if (target == nullptr) {
        // Fallback to any closest player
        target = ActorFixes_getClosestPlayer(this);
    }

    // Attack the target
    fireProjectileAt(target->position);
}
```

Audio: Console-Specific Sound Effects

> This is one of the few cases where it's safe to use `Game::localPlayerID`, as audio is local to each console - the other console doesn't receive or process your sound effects.

```
void MyHack::updatePlayerFlyState() {
    // Update the logic for all players
    for (s32 playerID = 0; playerID < Game::getPlayerCount(); playerID++) {
        Player* player = Game::getPlayer(playerID);
```

```
        // ... logic to update the fly state

        // Play flight finished jingle
        if (player->finishedFlying) {
            // Only the player that finished flying will hear the jingle
            if (playerID == Game::localPlayerID) {
                SND::playSFX(FLIGHT_FINISHED_SFX, &player->position);
            }
        }
    }
}
```

Another common pattern is to play sound effects when items are collected or power-ups are switched:

```
void RedRing::spawnReward() {
    for (s32 playerID = 0; playerID < Game::getPlayerCount(); playerID++) {
        Player* player = Game::getPlayer(playerID);

        // Determine reward based on power-up
        PowerupState reward = calculateReward(player->currentPowerup);

        // Play sound only for the local player when they get Fire Flower
        if (reward == PowerupState::Fire && playerID == Game::localPlayerID) {
            SND::playSFX(0x17E, &this->position);
        }

        // Spawn the item for this player
        spawnItemForPlayer(reward, playerID);
    }
}
```

Safe Uses of Game::localPlayerID

There are specific cases where using `Game::localPlayerID` is not only safe, but necessary:

1. **Sound Effects**: Audio is local to each console
2. **Visual UI Elements**: Screen-specific UI components like menus and HUD
3. **Liquid Position**: Due to co-op forcing shared areas, liquid levels are stored per-console
4. **File Loading**: Loading graphics or UI resources that are console-specific

```
// □ SAFE: Sound effects
if (playerID == Game::localPlayerID) {
    SND::playSFX(soundID, &position);
}

// □ SAFE: Liquid collision (special case - see liquid section)
if (player->position.y < Stage::liquidPosition[Game::localPlayerID]) {
    // Handle liquid damage
}

// □ SAFE: UI updates
if (playerID == Game::localPlayerID) {
    FS::loadFileLZ77(spectateTextFileID, (u16*)HW_OBJ_VRAM);
}
```

View Shaking: Per-Player Screen Effects

If you want to shake the screen for a specific player, never use the basic `ViewShaker::start` overloads with conditional logic:

```
// BAD: This causes instant desync
if (canShakePlayer(Game::localPlayerID)) {
    ViewShaker::start(type, viewID);
}
```

This causes an immediate desync. Instead, use the 4-argument overload without the conditional check:

```
void SledgeBro::doGroundPound() {
    for (s32 playerID = 0; playerID < Game::getPlayerCount(); playerID++) {
        Player* player = Game::getPlayer(playerID);

        // Check if this specific player should be affected
        if (ActorFixes_isPlayerInShakeRange(player)) {
            ViewShaker::start(3, this->viewID, playerID, false);

            // Play sound only for the local player
            if (playerID == Game::localPlayerID) {
                SND::playSFX(138, &this->position);
```

```
            }

            // Apply gameplay effects to this specific player
            if (!Game::getPlayerDead(playerID)) {
                player->takeDamage();
            }
        }
    }
}
```

Camera and Visibility Checks

Never use `Game::isOutsideCamera(..., Game::localPlayerID)` for gameplay logic.
Use `ActorFixes_isOutsideCamera` or `ActorFixes_isInRangeOfAllPlayers` instead:

```
void Enemy::updateBehavior() {
    // BAD: Only checks against local player's camera
    if (Game::isOutsideCamera(this->position, boundingBox, Game::localPlayerID)) {
        return; // Skip update
    }

    // GOOD: Checks against the closest player's camera
    if (ActorFixes_isOutsideCamera(this, boundingBox)) {
        return; // Skip update
    }

    // Continue with enemy logic...
}
```

For entities that need to stay active when any player can see them:

```
void Enemy::onUpdate() {
    // This ensures the actor only updates if ANY player can see it
    if (!ActorFixes_isInRangeOfAllPlayers(this)) {
        return; // All players are too far away, skip update
    }

    // Continue updating since at least one player can see us
    updateLogic();
```

```
}
```

Rendering Optimization

Use `ActorFixes_safeSkipRender` for 3D animated entities that need to update their models but may not render:

```
class HammerBro : public StageEntity3DAnm {
    bool skipRender() override {
        // This will update the model but only render for players who can see it
        return ActorFixes_safeSkipRender(this);
    }
};
```

Random Number Generation

Use `Game::getRandom()` for local code (UI, effects, sounds).
Use `Net::getRandom()` for gameplay logic that affects game state:

```
void Blockhopper::updateJump() {
    // BAD: Different random numbers on each console = desync
    if ((Game::getRandom() & 0xFF) == 0) {
        doJump();
    }

    // GOOD: Synchronized random numbers across consoles
    if ((Net::getRandom() & 0xFF) == 0) {
        doJump();
    }
}
```

Special Cases

Liquid/Lava Damage: The Exception to the Rule

>

> **Special Case:** Liquids are one of the few exceptions where you **DO** use `Game::localPlayerID`!

This is because the co-op implementation doesn't support per-player liquid levels - if liquid is detected in the level, both players are forced to always be in the same area, so they share the same liquid level. The liquid position is managed per-console, not per-player.

```
void checkLiquidDeath(Player* player) {
    s32 playerID = player->linkedPlayerID;

    // CORRECT: Use localPlayerID for liquid position
    // Both players share the same liquid level since they're in the same area
    if (player->position.y < Stage::liquidPosition[Game::localPlayerID]) {
        player->playSFXUnique(338, &player->position);
        Liquid_doWaves(player->position.x, 1);
        Game::losePlayerLife(playerID);
        Game::setPlayerDead(playerID, true);
    }
}
```

The reason for this exception:

- Liquid positions are stored per-console: `Stage::liquidPosition[Game::localPlayerID]`
- Co-op forces both players to be in the same area at all times
- Each console only tracks one liquid level (the local one)
- Trying to use `Stage::liquidPosition[playerID]` would fail because only index `[Game::localPlayerID]` is valid

StageLayout Data: Incomplete Arrays

> **Critical Issue:** Many parts of the StageLayout have arrays that appear to support both players (`PlayerCount`-sized arrays), but in reality **only the local player's data is populated or valid**.

This means that even though the StageLayout structure contains arrays like:

```
ScreenInfo screenFG[PlayerCount];     // Only [localPlayerID] is valid
ScreenInfo screenBG[PlayerCount];     // Only [localPlayerID] is valid
```

>

> **You MUST use `Game::localPlayerID` when accessing these arrays**, regardless of which player you're working with:

```
// ❌ BAD: Will access invalid/empty data for non-local player
u16 getForegroundID(u32 playerID) {
    return Stage_getFgScreenID(playerID); // This will fail for playerID != localPlayerID
}

// ✅ GOOD: Always use localPlayerID for StageLayout data
u16 getForegroundID(u32 playerID) {
    return Stage_getFgScreenID(Game::localPlayerID); // This works correctly
}
```

Real Examples from the Codebase:

1. **Pipes Background Fix**: The screen foreground data is only available for the local player:
   ```
   // Fixed pipes background - must use localPlayerID instead of playerID
   // Original code caused desyncs by trying to access screenFG[playerID]
   // when only screenFG[localPlayerID] contains valid data
   ```
2. **Volcano Eruption**: Background screen data is local-only:
   ```
   // Check if we're in a volcano level (screen ID 15)
   if (Stage_getFgScreenID(Game::localPlayerID) == 15) {
       ActorFixes_updateVolcanoBackground();
   }
   ```
3. **BG1CNT Register Fix**: Background control data is per-console:
   ```
   // Do not set BG1 CNT with other player's data - we don't have it!
   if (Game::localPlayerID == playerID) {
       // Apply background changes only for local player
   }
   ```

Why This Happens:

- Nintendo's original game was single-player, so many systems only tracked one set of data
- Background/foreground rendering data is managed per-console since each console renders independently
- Screen effects, camera settings, and background animations are local to each console

When to Use localPlayerID for StageLayout:

- Screen/background data access (`screenFG`, `screenBG`, `screenTS`)
- Camera and view-related information that's console-specific
- Background effects and animations
- Tile rendering and display settings

Rotators: Forced View Synchronization

> **Critical Issue:** Levels with **rotators** (rotating/tilting level mechanics) force additional constraints that override normal co-op behavior.

Rotators are detected by checking if any tileset has `screenID == 0xFF00`:

```
bool Stage_areaHasRotator() {
    u32 tilesetCount = Stage::getBlockElementCount(StageBlockID::Tileset);
    for (u32 i = 0; i < tilesetCount; i++) {
        if (Stage::stageBlocks.tileset[i].screenID == 0xFF00)
            return true;
    }
    return false;
}
```

Rotator Restrictions:

When a level has a rotator, **both players must always be in the same view**.

This limitation exists because each console only maintains its own StageLayout data, making it impossible to reliably detect or synchronize rotator states for the other player. Supporting independent rotators for both players would require a major overhaul of the system, so co-op forces both players to remain in the same view to guarantee that all rotator effects apply to a shared, consistent state.

Consequences of Rotators:

- Players cannot split into different areas/views
- Door and pipe transitions affect both players simultaneously

Examples of Rotator Levels:

- W8 Final Castle

Hardcoded Area Values

>

> **Important:** The codebase contains many hardcoded area number checks for specific levels and special behaviors. These represent special-case handling that you need to be aware of when modifying co-op behavior. There are plans to remove these hardcoded constraints by using actors that set flags on the levels and then despawn.

Mini-Mushroom Cutscene Areas (180, 181):

```
// Mini-mushroom cutscene areas
u32& areaNum = *rcast<u32*>(0x02085A94);
if (areaNum == 180 || areaNum == 181) {
    PlayerSpectate::clearSpectators();
    *rcast<u32*>(0x02085ACC) |= 0x20; // toadHouseFlag
    *rcast<u32*>(0x020CA8B4) = 0x1000; // timeLeft

    // Special handling for mini-mushroom collection areas
    if (itemType == 25 && player->currentPowerup == PowerupState::Mini) {
        Stage::exitLevel(1); // Exit to mini world
        return;
    }
}
```

Boss Defeat Cutscenes:

```
// World 2 Boss (42) and World 5 Boss (105)
u32& areaNum = *rcast<u32*>(0x02085A94);
if (areaNum == 42) { // World 2
    switchToCutsceneArea(0);
} else if (areaNum == 105) { // World 5
    switchToCutsceneArea(1);
}
```

Final Castle Special Handling (173):

```
// Hardcoded: prevent rotators from resetting but still get rid of lava in W8 Final Castle
u32& areaNum = *rcast<u32*>(0x02085A94);
if (areaNum == 173 && Game::getPlayer(0)->viewID != 0) {
    Stage_forceAreaReload = 2;
}
```

Boss Arena Loading (19, 175):

```
// Skip loading castle models for specific boss areas
u32& areaNum = *rcast<u32*>(0x02085A94);
if (areaNum == 19 || areaNum == 175) {
    return; // Don't load the model
}
```

Shared Camera Mode (174):

```
// Force shared camera for specific boss fights
u32& areaNum = *rcast<u32*>(0x02085A94);
if (areaNum == 174) {
    PlayerSpectate::sharedCamera = true;
}
```

Common Hardcoded Values:

- **19, 175**: Boss arenas requiring special model handling
- **42**: World 2 Boss (Mummy Pokey)
- **105**: World 5 Boss (Petey Piranha)
- **173**: World 8 Final Castle (rotator + lava)
- **174**: Specific boss fight with shared camera
- **180, 181**: Mini-mushroom cutscene areas

Why These Exist:

- **Memory Management**: Some boss fights need special model loading/unloading
- **Transition Logic**: Areas that exit to special worlds or cutscene areas
- **Camera/View Constraints**: Levels with unique mechanics (rotators, looper)

When Working with New Areas:

1. Check if your area number conflicts with existing hardcoded values
2. Consider whether your area needs special co-op handling
3. Add your own hardcoded checks if needed for area-specific behavior

Debugging and Troubleshooting

Desync Detection System

The codebase includes a `DesyncGuard` system that helps detect when the game state diverges between consoles. Key events that are monitored include:

- Player damage events
- Power-up changes
- Scene transitions
- RNG usage

If you're adding new gameplay systems, consider adding desync check markers at critical points:

```
void myGameplayFunction() {
    // Your gameplay logic here

    // Mark that this function was called to detect desyncs
    DesyncGuard::markDesyncCheck();
}
```

General Debugging Tips

> Remember: When in doubt, loop through all players and apply logic based on each player's individual state rather than assuming anything about the local player!

Key Questions to Ask:

- Does this code behave differently on Console 0 vs Console 1?
- Am I using `Game::localPlayerID` for gameplay logic?
- Are my random numbers synchronized between consoles?
- Will both consoles execute this logic identically?

Advanced Systems

Player Spectate System

The co-op hack includes a spectate system that allows dead players to watch the other player and automatically follow them through level transitions. This system maintains engaging co-op gameplay when one player dies, rather than forcing a restart or breaking the co-op experience.

How Spectating Works

When a player dies in co-op mode, instead of immediately respawning or ending the level, they enter **spectate mode**:

1. **Target Assignment**: The dead player's camera follows the living player

```
// playerID ^ 1 gives us the other player (0 becomes 1, 1 becomes 0)
PlayerSpectate::setTarget(deadPlayerID, deadPlayerID ^ 1);
```

2. **Camera Following**: The spectating player's camera smoothly lerps to follow their target

```
// Camera position updates to match the target player
Player* target = PlayerSpectate::getTargetPlayer(spectatorPlayerID);
target->followCamera(spectatorPlayerID);
```

3. **View Transitions**: When the living player enters doors/pipes, spectators automatically follow

```
// All spectators following transitPlayerID will switch views too
PlayerSpectate::syncSpectatorsOnViewTransition(transitPlayerID);
```

System Components

- **Target Tracking**: Each player has a target they're spectating (`playerTarget[playerID]`)
- **Local Target Cache**: Quick access to who the local player is watching (`localTarget`)
- **Smooth Transitions**: Camera lerping prevents jarring jumps when switching targets
- **Shared Camera Mode**: Special mode for certain levels (like the level looper in W8 Final Castle) where both players share one camera view
- **Entrance Following**: Spectators automatically transition to new areas when their target does

API Reference

```
// Check if a player is spectating someone else
bool PlayerSpectate::isSpectating(u32 playerID);

// Get who a player is currently spectating
Player* PlayerSpectate::getTargetPlayer(u32 playerID);

// Manually set spectate target
PlayerSpectate::setTarget(spectatorID, targetPlayerID);

// Enable smooth camera transitions
PlayerSpectate::setLerping(playerID, true);
```

Spectate Mode Triggers

Entering Spectate Mode:

- **Player Dies (Other Alive)**: Player dies and other player is still alive → Enter spectate mode
- **Level Start (No Lives)**: Player is dead when spawning (e.g., at level start with 0 lives)
- **Warp Cannon**: When using warp cannons, all other players spectate player 0 during the shooting sequence
- **Boss Victory Cutscenes**: Players not directly involved in the victory sequence spectate the player who triggered it
- **Flagpole**: (NOT IMPLEMENTED YET) When one player hits the flagpole, the other player spectates them during the victory sequence

Exiting Spectate Mode:

- **Player Respawns**: The player spawns back into the level → Respawned player returns to normal
- **New Level**: New level starts → Spectate targets reset to self (`PlayerSpectate::clearSpectators()`)
- **Cutscene End**: Boss introduction cutscenes end → Players return to normal

Camera Lerping System

The spectate system includes smooth camera transitions via **lerping** (linear interpolation):

Automatic Lerp Activation:

- **Boss Victory**: When players miss cutscenes or are repositioned during boss victories
- **Flagpole Hit**: (NOT IMPLEMENTED YET) Smooth transition when other player hits the flagpole

Automatic Lerp Deactivation:

The lerping system automatically stops itself when transitions complete:

```
// Position lerping stops when camera reaches close enough to target
if (distanceX < 48fx && distanceY < 48fx) {
    if (distanceX == 0 && distanceY == 0)
        playerLerping[playerID] = false; // Auto-disable
}

// Zoom lerping stops when zoom difference is eliminated
if (distance == 0)
    playerLerpingZoom[playerID] = false; // Auto-disable
```

Key Lerping Properties:

- **Smooth Movement**: Camera doesn't snap instantly to new positions
- **Distance Thresholds**: Lerping ends when within 48 units of target position
- **Separate Zoom**: Zoom lerping is handled independently from position lerping
- **Self-Terminating**: No manual intervention needed - lerping automatically stops when complete

The spectate system ensures that co-op gameplay remains engaging even when one player dies, allowing them to continue following the action and automatically rejoin when appropriate.

Summary

This guide covered the essential principles for writing co-op compatible code in NSMB:

Remember these key points:

- Never use `Game::localPlayerID` for gameplay logic (except liquids and local-only effects)
- Always loop through all players or use `this->linkedPlayerID` for collision-based logic
- Use `Net::getRandom()` for synchronized randomness, `Game::getRandom()` for local effects only
- Specify player IDs explicitly in functions like `ViewShaker::start()`
- Use co-op-safe helper functions like `ActorFixes_getClosestPlayer()`

When in doubt:

- Ask yourself: "Will this code behave identically on both consoles?"
- Test your changes with two consoles to verify synchronization
- Use the debugging tools and desync guards to catch issues early

> Following these patterns will help ensure your code works seamlessly in the co-op environment while maintaining the engaging two-player experience.

Revision #4
Created 2025-08-24 22:32:19 UTC by TheGameratorT
Updated 2025-08-24 23:22:27 UTC by TheGameratorT