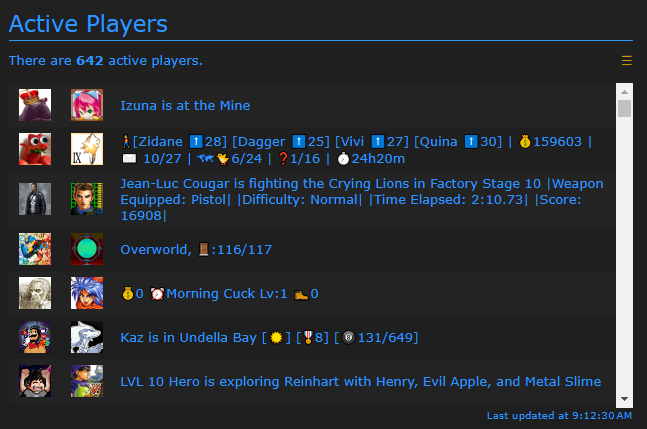
# Tutorial #11 – Rich Presence

## Overview

This tutorial will show a few examples of how to create \*\*Rich Presence\*\* (RP) for various types of games. \*\*Rich Presence\*\* is a public facing string of text that changes dynamically based on the player’s location in the game. If a \*\*Rich Presence\*\* is not defined the emulator will default to the text “Playing [Game name]”. By defining a \*\*Rich Presence\*\* you give other players an idea of where they are in a game. A bonus of using \*\*Rich Presence\*\* is it can expose cheaters, for example, if a player is a much higher level than they should be in the starting area. For more information on the \*\*Rich Presence\*\* see the [RATools Wiki](https://github.com/Jamiras/RATools/wiki/Rich-Presence-Functions).

![Rich Presence](RP\_in\_action.png)<br>

## Dictionaries



A \*\*dictionary\*\* is a special type of data structure that organizes data by \*\*key\*\*/ \*\*value\*\* pairs. The \*\*key\*\* is typically a number or a string which is converted into a \*\*hash\*\* that the computer uses to find where a \*\*value\*\* is in memory. The \*\*hash\*\* function is very fast which makes \*\*dictionaries\*\* ideal for searching through large data sets. For \*\*Rich Presence\*\*, \*\*dictionaries\*\* are used to convert in-game memory values to readable text. For example, the first zone in Sonic the Hedgehog is Green Hill Zone Act 1. By using a \*\*dictionary\*\* to define all the level names you could display “Green Hill Zone Act 1” in the \*\*Rich Presence\*\* instead of the level value of 0x0000. See the [RATools Wiki](https://github.com/Jamiras/RATools/wiki/Variables#dictionaries) for more information of how \*\*dictionaries\*\* works.

```

// To create a dictionary

dictionary = {

key\_1: value\_1,

key\_2: value\_2,

…

key\_n: value\_n

}

// To access a dictionary value

dictionary[key] = value

```

## String Format

The \*\*String Format\*\* function is a common way to build a string with real memory values that are formatted for easy reading. When defining a \*\*Format String\*\*, numbered curly brackets are used as placement markers to tell the computer where to insert a value. The number in the curly brackets corresponds to the parameter number of format function starting at zero. The format function itself is dynamic and can have as many parameters as necessary. For \*\*Rich Presence\*\* the \*\*String Format\*\* function is used to format the text to make the game memory readable. See the [RATools Wiki](https://github.com/Jamiras/RATools/wiki/Built-in-Functions#formatformat\_string-parameters) for more information of how the \*\*Format String\*\* function works.

```format(“Parameter #1: {0}, Parameter #2: {1}, …, Parameter #n {n}, parameter\_1, parameter\_2, … parameter\_n)```

## Value Expressions

The \*\*rich\_presence\_value\*\* function is used to convert a value in game memory into readable text. The benefit of this function is that it supports multiple formats depending on the type of memory you are trying to display. Often you will be showing the raw memory values however you can display memory as various time formats or as a floating point number. See the [RATools Wiki](<https://github.com/Jamiras/RATools/wiki/Rich-Presence-Functions#rich_presence_valuename-expression-format>) for a complete list of the possible formats.

## Lookup Expressions

The \*\*rich\_presence\_lookup\*\* function is used to look up the in game values from a \*\*dictionary\*\*. The in game memory is used as a \*\*key\*\* and it returns to matching \*\*value\*\*. If the \*\*key\*\* is not in the \*\*dictionary\*\* then you can either define a fallback string, or an empty string is used if no fallback is defined. See the [RATools Wiki](https://github.com/Jamiras/RATools/wiki/Rich-Presence-Functions#rich\_presence\_lookupname-expression-dictionary-fallback) for more information on dictionary look ups.

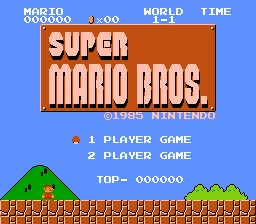
## Macro Expressions

The \*\*rich\_presence\_macro\*\* function is similar to the \*\*rich\_presence\_value\*\* function except that the macros are built-in to the Libretro library. The biggest advantage to using macros is when the game has ASCII or Unicode strings in the memory since the macros handle the text conversion internally. See the [RATools Wiki](https://github.com/Jamiras/RATools/wiki/Rich-Presence-Functions#rich\_presence\_macromacro-expression) for a complete list of the possible macros.

# Example #11A: Super Mario Bros.

The game selected for this example is [Super Mario Bros. (NES)](<https://retroachievements.org/game/1446>). The reason why this game was selected is because it is a well-known game and it was used as an example in the [RADocs](https://docs.retroachievements.org/Rich-Presence/).

![Super Mario Bros. Title Screen](SMB\_Title.png)<br>



```

// Super Mario Bros.

// #ID = 1446

// $000E: Player's State:

function PlayerState() => byte(0x000e)

StatusLookup = {

0: " [Loading]",

1: " taking a vine warp",

2: " entering a warp pipe",

3: " entering a warp pipe",

4: " ",

5: " [Stage Complete]",

6: " [Game Over]",

7: " [Entering Area]",

9: " growing",

0xA: " shrinking",

0xB: " ",

0xC: " powering up",

}

// $0704: Can swim flag

function Swimming() => byte(0x000704)

SwimmingLookup = {

1: " swimming",

}

// $0756: Set to 01 when large mario, 02 when fire-flower mario

function PowerUp() => byte(0x000756)

PowerupLookup = {

0: "Small",

1: "Super",

2: "Fire",

}

// $075C: Act Number (0 based)

function Act() => byte(0x00075C)

// $075F: World (0 based)

function World() => byte(0x00075F)

// $075A: Mario Number of Lives, set to ff when game over

function Lives() => byte(0x00075A)

// $0770: Gameplay Mode, 00=demo/menu, 01=playing,

// 02=end of level

// NB. Set to 02 and wait to advance a world

function GameplayMode() => byte(0x000770)

ModeLookup = {

0: "[Demo] ",

2: "[World Complete] ",

}

// $0776: Pause: 00=normal 81=pausing 01=paused 80=unpausing

function Pause() => byte(0x000776)

PausedLookup = {

0x81: "▌▌ ",

0x80: "▌▌ ",

1: "▌▌ ",

}

// $079f: Invulnerability Counter: starts at 23, counts to 0, 0=normal

function StarCounter() => byte(0x00079F)

function StarBitcount() => bit0(0x00079F) + bit1(0x00079F) + bit2(0x00079F) + bit3(0x00079F) + bit4(0x00079F) + bit5(0x00079F)

StarLookup = {

5: "🌟 ",

4: "🌟 ",

3: "🌟 ",

2: "🌟 ",

1: "🌟 ",

}

// $07FC: Second Quest enabled = 01

function SecondQuest() => byte(0x0007FC)

QuestLookup = {

0x0: "1st",

0x1: "2nd",

}

// Example #11A - Rich Presence for Super Mario Bros.

rich\_presence\_display("{0}{1}{2}{3} Mario in {4}-{5}{6}{7}, 🚶:{8}, {9} Quest",

rich\_presence\_lookup("Mode", GameplayMode(), ModeLookup),

rich\_presence\_lookup("Paused", Pause(), PausedLookup),

rich\_presence\_lookup("Star", StarBitcount(), StarLookup),

rich\_presence\_lookup("Powerup", PowerUp(), PowerupLookup),

rich\_presence\_value("Digit", World() + 1),

rich\_presence\_value("Digit", Act() + 1),

rich\_presence\_lookup("Swimming", Swimming(), SwimmingLookup),

rich\_presence\_lookup("Status", PlayerState(), StatusLookup),

rich\_presence\_value("Digit", Lives() + 1),

rich\_presence\_lookup("Quest", SecondQuest(), QuestLookup)

)

```

## rich\_presence\_display()

The ```rich\_presence\_display()``` function is used to define the default \*\*Rich Presence\*\*. It is used in a similar way to the format() function where the first parameter is the format string and the rest of the parameters are the values inserted into the format string. See the [RATools Wiki](<https://github.com/Jamiras/RATools/wiki/Rich-Presence-Functions#rich_presence_displayformat_string-parameters>) more information on ```rich\_presence\_display()```.

## {0} Mode

The ```ModeLookup`` has two different values. If the memory accessor ```GameplayMode()``` is 0 then it displays “[Demo]” and if the value is 2 then it displays “[World Complete]”. Any other values will result in an empty string.

## {1} Paused

The ```PausedLookup``` has three values for the same display output. If the memory accessor ```Pause()``` is 0x1, 0x80, or 0x80 then it displays the pause emoji. Any other values will result in an empty string.

## {2} Star Power

The ```StarLookup``` will display a star for any value between 1 to 5. The memory accessor ```StarBitcount()``` counts the first 5 bits of the byte and ignores the rest. By counting the bits instead of using the raw values the ```StarLookup``` can be smaller since it only needs to have 5 possible values instead of 31.

## {3} Power Up

The ```PowerupLookup ``` has three values for each power state of Mario. If the memory accessor ```PowerUp()``` is 0 then it displays “Small”, if Mario collects a mushroom the value is 1 then it displays “Super”, and when Mario collects a fire flower then it will display “Fire” Any other values will result in an empty string.

## {4} World

The current world is zero based meaning that when Mario is in world 1 to memory accessor ```World()``` will be 0. To display correctly we add 1 to the value of ```World()```.

## {5} Act

Similar to the world, the current act is also zero based meaning that when Mario is in act 1 to memory accessor ```Act()``` will be 0. We use the same technique to display correctly by adding 1 to the value of ```Act()```.

## {6} Swimming

The ```SwimmingLookup ``` only has one value defined. If the memory accessor ```Swimming ()``` is 1 then it displays “swimming” otherwise an empty string will be displayed.

## {7} Status

The ```StatusLookup``` has 12 different definitions for the ```PlayerState()```memory accessor. This memory is used for both the game state and the current state of Mario. The state information is more useful for achievement code than \*\*Rich Presence\*\* however, it gives some additional context to what the player is currently doing.

## {8} Lives

Like both the world and act, the current number of lives Mario has left is zero based meaning when Mario has 4 lives to memory accessor ```Lives()``` will be 3. We use the same technique as the world and act to display the lives correctly by adding 1 to the value of ``` Lives()```.

## {9} Quest

The ```QuestLookup``` has two values defined for the current loop of the game. Super Mario Bros. is a game that has a second harder loop to play when you beat the final boss, Bowser, for the first time. If the memory accessor ```SecondQuest()``` is 0 then it displays “1st”and if the value is 1 then it displays “2nd”. Any other values will result in an empty string.

# Example #11B: Muppet RaceMania

The game [Muppet RaceMania](<https://retroachievements.org/game/13910>) was selected for this example because it has multiple regions linked to it. Each region has different memory locations so both the Rich Presence and Achievements need to have separate logic for each region supported. Fortunately, RATools can help manage the memory differences and make equivalent logic for each region.

![Muppet RaceMania Title Screen](Muppets\_Title.png)<br>



```

// Muppet RaceMania

// #ID = 13910

// Give each language an unique value

EU = 0

FR = 1

DE = 2

IT = 3

ES = 4

US = 5

// Array of each supported language

Languages = [EU, FR, DE, IT, ES, US]

// $00b8b7: [ASCII] CD-ROM Name

// SCES\_020.01;1 = EU

// SCES\_024.83;1 = FR

// SCES\_024.84;1 = DE

// SCES\_024.85;1 = IT

// SCES\_024.86;1 = ES

// SLUS\_012.37;1 = US

function CDROM() => 0xb8b7

// "SCES\_020.01;1" = EU Disc

EU\_Disc = [0x53,0x43,0x45,0x53,0x5F,0x30,0x32,0x30,0x2E,0x30,0x38,0x3B,0x31]

// "SCES\_024.83;1" = FR Disc

FR\_Disc = [0x53,0x43,0x45,0x53,0x5F,0x30,0x32,0x34,0x2E,0x38,0x33,0x3B,0x31]

// "SCES\_024.84;1" = DE Disc

DE\_Disc = [0x53,0x43,0x45,0x53,0x5F,0x30,0x32,0x34,0x2E,0x38,0x34,0x3B,0x31]

// "SCES\_024.85;1" = IT Disc

IT\_Disc = [0x53,0x43,0x45,0x53,0x5F,0x30,0x32,0x34,0x2E,0x38,0x35,0x3B,0x31]

// "SCES\_024.86;1" = ES Disc

ES\_Disc = [0x53,0x43,0x45,0x53,0x5F,0x30,0x32,0x34,0x2E,0x38,0x36,0x3B,0x31]

// "SLUS\_012.37;1" = US Disc

US\_Disc = [0x53,0x4C,0x55,0x53,0x5F,0x30,0x31,0x32,0x2E,0x33,0x37,0x3B,0x31]

// Dictionary of the disc names

Discs = {

EU: EU\_Disc,

FR: FR\_Disc,

DE: DE\_Disc,

IT: IT\_Disc,

ES: ES\_Disc,

US: US\_Disc

}

// 0xb0f38: Player's character (EU)

// 0xb0328: Personnage du joueur (fr)

// 0xb01dc: Spielercharakter (De)

// 0xb01ec: Personaggio dei giocatore (It)

// 0xb038c: Personaje del jugador (Es)

// 0xb07a4: Player's character (US)

PlayerCharacter = {

EU: byte(0xb0f38),

FR: byte(0xb0328),

DE: byte(0xb01dc),

IT: byte(0xb01ec),

ES: byte(0xb038c),

US: byte(0xb07a4)

}

// Dictionary of all the racers

Racers = {

0: "Kermit",

1: "Miss Piggy",

2: "Gonzo",

3: "Fozzie",

4: "Rizzo",

5: "Beaker",

6: "Rowlf",

7: "Animal",

8: "Janice",

9: "Dr. Teeth",

10: "Floyd",

11: "Zoot",

12: "Bunsen",

13: "Chef",

14: "Sam",

15: "Scooter",

16: "Pepe",

17: "Bobo",

18: "Link",

19: "Robin",

20: "Johnny Fiamma",

21: "Sal",

22: "Bean Bunny",

23: "Clifford",

24: "Doozer"

}

// 0xb0f3c: Player's vehicle (EU)

// 0xb032c: Véhicle du joueur (Fr)

// 0xb01e0: Spielerfahrzeug (De)

// 0xb01f0: Veicolo del giocatore (It)

// 0xb0390: Vehiculo del jugador (Es)

// 0xb07a8: Player's vehicle (US)

PlayerVehicle = {

EU: byte(0xb0f3c),

FR: byte(0xb032c),

DE: byte(0xb01e0),

IT: byte(0xb01f0),

ES: byte(0xb0390),

US: byte(0xb07a8)

}

// All the vehicles in the game

Vehicles = {

0: "Racer",

1: "Monster Truck",

2: "Flying Saucer",

3: "Bubbler",

4: "Cheese",

5: "Moon Buggy",

6: "Bulldozer",

7: "Dragster",

8: "Love Bug",

9: "Hotrod",

10: "Spoetster",

11: "Supercharger",

12: "Hovercraft",

13: "Hotdog",

14: "Plane",

15: "Dune Buggy",

16: "Kart",

17: "Water Buggy",

18: "Spaceship",

19: "Steamroller",

20: "Speedster",

21: "Flamer",

22: "Snow Bike",

23: "Trike",

24: "Dumper"

}

// 0x0b1628: Current course set (EU)

// 0x0b0a18: Set de parcourses courante (Fr)

// 0x0b08d0: Aktuelle Set von Strecken (De)

// 0x0b08e0: Set di percorsi corrente (It)

// 0x0b0a80: Set de pistas actual (Es)

// 0x0b0e38: Current course set (US)

Set = {

EU: byte(0x0b1628),

FR: byte(0x0b0a18),

DE: byte(0x0b08d0),

IT: byte(0x0b08e0),

ES: byte(0x0b0a80),

US: byte(0x0b0e38)

}

// 0x0b1622: Current course (EU)

// 0x0b0a12: Parcours courante (Fr)

// 0x0b08ca: Aktuelle Strecke (De)

// 0x0b08da: Percorso corrente (It)

// 0x0b0a7a: Pista actual (Es)

// 0x0b0e32: Current course (US)

Course = {

EU: byte(0x0b1622),

FR: byte(0x0b0a12),

DE: byte(0x0b08ca),

IT: byte(0x0b08da),

ES: byte(0x0b0a7a),

US: byte(0x0b0e32)

}

// Dictionary of tracks are hashed by Set # \* 16 + Course #

Tracks = {

0x00: "Swamp",

0x01: "Ghost Town",

0x02: "Canyon",

0x03: "Funfair",

0x10: "Happiness Hotel",

0x11: "Mallory Gallery",

0x12: "Trafalgar Square",

0x13: "Resturant",

0x20: "Central Park",

0x21: "New York City",

0x22: "Theatre",

0x23: "Diner",

0x30: "Old London Town",

0x31: "Graveyard",

0x32: "Chamber",

0x33: "Rooftops",

0x40: "Dock",

0x41: "Treasure Island",

0x42: "Lagoon",

0x43: "Mountain",

0x50: "Secret Base",

0x51: "Cape Doom",

0x52: "Laboratory",

0x53: "Muppet Labs",

0x60: "Studio",

0x61: "Arches",

0x62: "Fraggle Rock",

0x63: "Disco"

}

// 0x0c3e10: Player's race position (EU)

// 0x0c3158: Position du joueur (Fr)

// 0x0c3010: Platz im Rennen (De)

// 0x0c3020: Posizione del giocatore (It)

// 0x0c31c0: Posición del jugador (Es)

// 0x0c3572: Player's race position (US)

RacePosition = {

EU: byte(0x0c3e10),

FR: byte(0x0c3158),

DE: byte(0x0c3010),

IT: byte(0x0c3020),

ES: byte(0x0c31c0),

US: byte(0x0c3572)

}

// Dictionary of the place names for each position

Place = {

1: "1st",

2: "2nd",

3: "3rd",

4: "4th",

5: "5th",

6: "6th",

7: "7th",

8: "8th",

9: "9th",

10: "10th",

11: "11th",

12: "12th"

}

// Check if the string hex values equal the hex values at the passed address

function StringCompare(address, string)

{

strArray = []

for hexValue in string

{

array\_push(strArray, byte(address) == hexValue)

address = address + 1

}

return all\_of(strArray, a => a)

}

// Make 25 example achievements using that support multiple languages

for index in range(0,24)

{

// Setup the title and description of

title = Racers[index] + " " + Vehicles[index]

description = "Select the racer " + Racers[index] + " with the vehicle " + Vehicles[index]

triggers = []

for language in Languages

{

// Make a condition to check the current language disc region and racer/vehicle combo

array\_push(triggers,

StringCompare(CDROM(), Discs[language]) &&

PlayerCharacter[language] == index &&

PlayerVehicle[language] == index

)

}

achievement(title, description, 0, any\_of(triggers, t => t))

}

// Example #11B - Rich Presence the supports mulitple regions

for language in Languages

{

disc = Discs[language]

rich\_presence\_conditional\_display(

StringCompare(CDROM(), disc) &&

Set[language] != 0xff &&

Course[language] != 0xff &&

RacePosition[language] >= 1 &&

RacePosition[language] <= 12,

"{0} is driving a {1} on the track {2} in {3} position!",

rich\_presence\_lookup("Racer", PlayerCharacter[language], Racers),

rich\_presence\_lookup("Vehicle", PlayerVehicle[language], Vehicles),

rich\_presence\_lookup("Track", Set[language] \* 0x10 + Course[language], Tracks),

rich\_presence\_lookup("Position", RacePosition[language], Place)

)

}

rich\_presence\_display("The muppets are in the menu!")

```

## Using Dictionaries for Multiple Regions

Similar to the regional strings [Example #10A](.. /10\_Strings/Example\_10A.md) from the last tutorial we will be using the disc name to detect the region. Memory accessors are defined using dictionaries instead of functions so that they can be looped through programmatically. For example, instead of defining the ```PlayerCharacter()``` we store the accessor using the language as the key. To access the North American memory use the command ```PlayerCharacter[US]```. The rational for setting up the memory like this is that you can create alts using the same basic logic. Note that each language is a unique value and the array ```Languages``` contains all of the supported language values.

Included in the above code is a silly example of how to create achievements using dictionaries. The achievements are to check if the racer has selected a certain vehicle. The dictionaries we defined for the rich presence is also used here to build the title and description. The logic for each alt is essentially the same except for the part that checks the disc name. The nice thing about coding an achievement like this is only one alt is ever active at a time.

## rich\_presence\_conditional\_display()

In this example we used the command (rich\_presence\_conditional\_display)[https://github.com/Jamiras/RATools/wiki/Rich-Presence-Functions#rich\_presence\_conditional\_displaycondition-format\_string-parameters) to control what message is displayed depending on the disc name. Each message format is exactly the same and the code will fill in the correct memory address for the region. Remember that order matters when using the conditional display and it will display the string for the first condition that is met even if later conditions would also be true.

## {0} Racer

The ```Racers ``` lookup has all 25 possible racers defined. The memory accessor ```PlayerCharacter[]``` has a one to one match for the values in the lookup.

## {1} Vehicle

The ```Vehicles ``` lookup has all 25 possible vehicles defined. Like the Racer, the memory accessor ```PlayerVehicle[]``` has a one to one match for the values in the lookup.

## {2} Track

The ```Tracks ``` lookup has all 24 possible tracks defined. This value is a composite value of the ```Set[]``` and ```Course[]``` accessors. By combining these values together we can define all of the tracks in one lookup. Note that the value of ```Set[]``` is multiplied by 16 to ensure each track value is unique. The multiplier 16 was chose since in hex the ```Course[]``` would be the lower four bits and ```Set[]``` would be the upper four bits. Other multipliers could be used so long as the possible combinations all yield unique results.

## {3} Position

The ```Place``` lookup has all 12 possible race placements defined. Like the Racer and Vehicle, the memory accessor ```RacePosition[]``` has a one to one match for the values in the lookup. Cardinal values (1, 2, …, 12) could be displayed directly however, a dictionary was chosen to display the ordinal values (1st, 2nd, …, 12th) instead.

# Example #11C: Pitfall! Revisited

The game [Pitfall!](<https://retroachievements.org/game/11191>) was selected for this homework example since most of the values require arithmicatic operations to display correctly. Pitfall! was covered in [Tutorial #4 – Arithmetic Operations](../04\_Arithmetic\_Operations/readme.md) if you would like a refresher on the game.<br>

![Pitfall Title Screen](Pitfall\_Title.png)<br>



## Homework #11

Create a Rich Presence for a Pitfall! using the format: ``` "⏳{0} | 💳{1}/32 | 🧑x{2} | 💯{3}" ```` <br>

Where {0} is time, {1} is treasures obtained, {2} is lives, and {3} is score.

## Useful Memory

To complete the homework problem you’ll need the memory addresses:<br>

0x0000 -> [8-bit] Lives, bit5=1st extra life, bit7=2nd extra life <br>

0x001E -> [8-bit] 00=game active <br>

0x0055 -> [BCD] Score XX0000 <br>

0x0056 -> [BCD] Score 00XX00 <br>

0x0057 -> [BCD] Score 0000XX <br>

0x0058 -> [BCD] Time XX:00 <br>

0x0059 -> [BCD] Time 00:XX <br>

0x005A -> [8-bit] Time (milliseconds) <br>

0x006D -> [8-bit] Treasures flags 1-8 <br>

0x006E -> [8-bit] Treasures flags 9-16 <br>

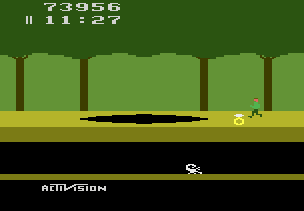
0x006F -> [8-bit] Treasures flags 17-24 <br>

0x0070 -> [8-bit] Treasures flags 25-32 <br>

# Example #11C: Pitfall! Revisited

The Rich Presence for Pitfall! shows the time left, the number of treasures collected, the number of lives left, and the players current score.

![Pitfall Harry is about to collect some gold](Pitfall\_Golden\_Scorpion.png)<br>



```

// Pitfall!

// #ID = 11191

// $0000: [8-bit] Lives

// bit5=1st extra life

// bit7=2nd extra life

function Lives() => bitcount(0x000000)

// $001E: [8-bit] 00=game active

function GameActive() => byte(0x00001E)

// $0055: [BCD] Score XX0000

// $0056: [BCD] Score 00XX00

// $0057: [BCD] Score 0000XX

function ScoreX00000() => high4(0x000055)

function Score0X0000() => low4(0x000055)

function Score00X000() => high4(0x000056)

function Score000X00() => low4(0x000056)

function Score0000X0() => high4(0x000057)

function Score00000X() => low4(0x000057)

// Converts the BCD score values into Base 10 decimal

function Score()

{

return ScoreX00000() \* 100000 +

Score0X0000() \* 10000 +

Score00X000() \* 1000 +

Score000X00() \* 100 +

Score0000X0() \* 10 +

Score00000X()

}

// $0058: [BCD] Time XX:00

// 20=out of game

// $0059: [BCD] Time 00:XX

// $005A: [8-bit] Time (miliseconds)

function TimeX00000() => high4(0x000058)

function Time0X0000() => low4(0x000058)

function Time00X000() => high4(0x000059)

function Time000X00() => low4(0x000059)

function Time0000XX() => byte(0x00005A)

// Converts the BCD time values into frames

// Note that minutes and seconds are BCD and the frames are already Base 10

function Time()

{

return TimeX00000() \* 36000 +

Time0X0000() \* 3600 +

Time00X000() \* 600 +

Time000X00() \* 60 +

Time0000XX()

}

// $006D: [8-bit] Treasures flags 1-8

// $006E: [8-bit] Treasures flags 9-16

// $006F: [8-bit] Treasures flags 17-24

// $0070: [8-bit] Treasures flags 25-32

function Treasures1() => bitcount(0x00006D)

function Treasures2() => bitcount(0x00006E)

function Treasures3() => bitcount(0x00006F)

function Treasures4() => bitcount(0x000070)

function Treasures() {

return Treasures1() +

Treasures2() +

Treasures3() +

Treasures4()

}

// Example #11C - Rich Presence for Pitfall

rich\_presence\_conditional\_display(GameActive() != 0 || Time() == 0,

"Taking a break between adventures"

)

rich\_presence\_display("⏳{0} | 💳{1}/32 | 🧑x{2} | 💯{3}",

rich\_presence\_value("Time", Time(), format="FRAMES"),

rich\_presence\_value("Digit", Treasures()),

rich\_presence\_value("Digit", Lives()),

rich\_presence\_value("Digit", Score())

)

```

## {0} Time

Pitfall Harry has twenty minutes to collect all the treasures. The ```Time()``` function calculates the time left by combining Binary-Coded Decimal (BCD) values for the minutes and seconds displayed on the screen plus the number frames. To display the time in the typical format H:MM:SS.ms the format is set to ```FRAMES```.

## {1} Treasures

Pitfall! Has 32 total treasures to collect with each being marked collected by an array of bitflags. The number of treasures collected is calculated by adding the total number of bits active in the bitflag array using the ```Treasures()``` function.

## {2} Lives

Pitfall Harry has two extra lives from the beginning of each game. Each extra life has its own bitflag so number of extra lives is calculated by adding the bits together with the ```Lives()``` function.

## {3} Score

Starting with 2000 points the score will increase when Harry collects a treasure and decrease when Harry gets hit. Like the time the score is a BCD value, the total score is calculated by multiplying each digit by its decimal place and adding them together with the ```Score()``` function.