# patchlib

*Release 1.1*

**BrettefromNesUniverse**

**Sep 24, 2023**

# SUBPACKAGE DOCUMENTATION:

# patchlib by Nes Universe

`patchlib` is a library of tools for total control of the fundamental elements of each `patch` filetype, and is split into several packages, each documented in detail in this repository, and installed on: PyPi

Or can be installed with:

| Windows | Posix / Unix |
|---|---|
| `pip install get_patchlib` | `pip3 install get_patchlib` |

- *Intended for Python 3.7 and above*
- Uses importlib for complete importing
- Currently can only use `ips` files.

| Name | package | Docs | Version | Size | Docs |
|---|---|---|---|---|---|
| IPS | patchlib.ips | *patchlib.ips_docs* | 1.0 | 17KB | *IPS* |
| BPS | `patchlib.bps` | Not yet | alpha | 5.85KB | None |
| Xdelta | `patchlib.xdelta` | Not yet | None | None | None |
| PPF | `patchlib.ppf` | Not yet | None | None | None |
| UPS | `patchlib.ups` | Not yet | None | None | None |
| APS | `patchlib.aps` | Not yet | None | None | None |
| RUP | `patchlib.rup` | Not yet | None | None | None |
| NPS | `patchlib.nps` | Not yet | None | None | None |
| Luna | `patchilb.luna` | Not yet | None | None | None |

Using `import patchlib` will install all packages with the `importlib` module. `patchlib` provides no context abundant methods and relies on the package being specified.

`Luna` serialized class handling may be implemented at a later point in time once enough of the crucial patch filetypes have been completed.

docs applies only to the working build *and therefore may not reflect the* open source versions. *Contributions are open an encouraged, and will be release once the open build leaves exerpiment phase.*

In either `cmd` or `powershell` for Windows, and `terminal` for Posix/Unix Systems.

---

**Note:** Currently only `patchlib.ips` is implemented, we are already working the next subpackage however and welcome contributions

---

`patchlib.ips` is the best `ips` file handler out there, while slower than it's C++ counterparts. `patchlib` can boast that it offers

- Total control of every byte in the IPS
- Serializable objects for sharing/extended use
- Produces the *smallest* `ips` files of any `ips` handler
- Lightweight filesize sitting only at `17kb`
- extendable scope to `0x100FFFE`, or standard `0xFFFFFF`

CHAPTER

# ONE

# IPS HANDLING

`patchlib` is useful for *all* elements of `ips` handling, even some that are likely to be left unused.

```
from patchlib.ips import *          #import ips library

with open("EXTRA MARIO BROS.ips","rb") as f:
    mod = f.read()

mod = ips(mod, False)               #construct `ips` object from file
```

We pass the `bytes` object into the constructor, and can specify whether or not this `ips` should comply to `24 bit` standards. Set to `True` by default the optional positional arg `legacy` indicates whether or no the `ips` should be allowed to write up to `0x100FFFE` which is the `32bit` limit of `ips`. Since an `ips` is just a series of `instances`, the methods within the `ips` class are just for `instance` handling:

```
#retrieve multiple offsets within a range

header = mod.range(end=16)          #here we retrieve all instances up to the 16th byte
PRG = mod.range(16,0x8010)          #here we retrieve all instances from 16 to 0x8010
CHR = mod.range(0x8010)             #here we retrieve all instances from 0x8010 to the␣
↪end

#retrieve an instance from offset, or many with a shared or unique name
mod.get(1622)                                       #retrieve instance starting at␣
↪offset 1622
mod.get("unnamed instance at 1984 - 2010 | 26")
```

The default name on an `instance` is `unnamed instance at` to signify that it was assigned a name during construction or modification, then we get the `offset` to the `end` describing it's range including it's `size` at the end. The names do not affect how an`instance` works, it merely aids readability of the file.

An `instance``may also be created and removed from an ``ips`

```
#This code will insert the bytearray b"Example" at 1234
mod.create(offset = 1234, data = b"Example")

#or with rle...
mod.create(offset = 1234, data = (10, b"E"))
```

If the space you are attempting to write data is occupied by another *instance* you can use the optional *kwarg overwrite* which will overwrite existing *instances* to make space for the new *instance*

```
mod.create(offset = 1234, data = b"Example", override = True)
```

In the above example we keep as much of the non-clashing *instance* data within the clashing *instance*. This is due to the optional *kwarg sustain* default value being *True*. If the data *should not* be sustained, then this can be executed like so:

```
mod.create(offset = 1234, data = b"Example", override = True, sustain = False)
```

However, in most cases `sustain` will likely be used, furthermore it may also be the case that the overwriting data may exist to modify existing instance data in which case it may make sense to use the optional `kwarg` `merge` which connects the new `instance` with any possible consecutive `instance`. By default `merge` is set to `False` and is only usable when `sustain` is `True`.

```
mod.create(offset = 1234, data = b"Example", override = True, merge= True)
```

If an `instance` needs to be removed, we have the `remove` function. This function does more than just remove an `instance` however, it returns a `dict` of every `attribute` in the `instance`. Further, if the `discriminator` is a string and we have multiple `instances` it will remove them all and return a `tuple` storing all `instances` in ascending order by `offset`.

```
dict_of_data = mod.remove(1234)                                                 ␣
↪                            #removing by offset
tuple_of_datasets = mod.remove("unnamed instance at 1984 - 2010 | 26")      #removing by␣
↪name
```

Other ways to access the `instances` in an `ips` may be index specification or slices, or using the `__iter__` method with `for`:

```
ins = mod.instances[20]
ins = mod.instances[20:30]

ins = [ins for ins in mod if ins.size > 100]
for i in mod:
    if i.size > 100:
                ins = i ; break
```

Once the necessary jobs regarding the `ips` is complete, you can then create a bytes object for it to be wrote to a file with `ips.to_bytes()`.

## Instance Handling The "Total" control of `patchlib.ips` is as advertised, the `instance` class has one method excluding initialization, which is `modify` which acts just like `ips.create` but `offset` and `data` are now optional as they have been predefined at least once.

```
reference = mod.get(1234)
reference.modify(data = (30, b"f"), overwrite = True)
reference.modify(offset = 200, name = "New Name")
```

CHAPTER

TWO

# EXCEPTIONS

- `ScopeError` This `Exception` will raise when the task is infeasible given the limitations of `ips` as a system.
- `OffsetError`` This `Exception` will raise when the task demands an impossible offset during creation or modification.

CHAPTER

THREE

# METHODS

The `apply` method takes an `ips` object and a `bytes` object for the base file.

```
with open("Super Mario Bros. (World).nes", "rb") as f:
        base = f.read()
with open("EXTRA MARIO BROS.ips","rb") as f:
    mod = ips(f.read())
with open("EXTRA MARIO BROS.nes","wb") as f:
    f.write(apply(mod, base))
```

The `build` method takes two `bytes` objects, one for the base file and one for the target file. It also takes an optional positional arg `legacy` which indicates if we should allow writes up to `0x100FFFE`.

```
with open("Super Mario Bros.nes", "rb") as f:
    base = f.read()
with open("My Cool Mario ROMhack.nes","rb") as f:
    target = f.read()
with open("My Cool Mario ROMhack.ips","wb") as f:
    f.write(build(base, target))
```

CHAPTER

FOUR

# INTERNATIONAL PATCHING SYSTEM

## 4.1 What is IPS?

The `IPS` or `International Patching System` filetype was created in 1993 to express "diffs" between a controlled `base` file and a resultant `target` file. MS-DOS at this point was the most popular operating system and some versions could not store a `32 bit` number due to limitations of the time.

The `IPS` filetype was developed by Japanese "ROM-Hackers" who wished to share their modifications in a way that did not promote or commit acts of digital piracy, these files were spread on file hosting/shareware and forum websites before mainstream ROM-Hacking communities were established such as Super Mario World Central which stores the largest archive of Super Mario World ROM-Hacks, and is the most popular use for this filetype.

**Popular IPS tools include:**

- Lunar IPS
- Floating IPS
- EWing IPS Patcher

Lunar IPS is the most popular, but does not support `BPS` like Floating IPS does. All the above are designed for Windows Systems except EWing IPS Patcher which should work on any `Posix` System.

**As well as historic ones such as:**

- IPS
- IPSMac
- JIPS

IPS is designed for 90's Windows Operating Systems such as MS-DOS, IPSMac is designed for early Macintosh systems, and JIPS is simply an `ips` handler designed in Java.

**As well as more developmental IPS tools:**

- IPS Peek
- ips.py
- Chief-Net IPS

IPS Peek and Chief-Net IPS both allow selective patching, meaning that parts of the IPS may be excluded or included when patching by the users choice. ips.py is a Python `ips` tool, however it is not as versatile as `patchlib.ips` which is the *recommended* Python IPS Module.

ROM Patcher JS, however eradicates the usage of executable "diff" appliers/makers as the tool is made entirely in JavaScript and therefore can use mod files and base files in a browser. (The Exception being `xdelta` files that use `Xdelta3` format which requires an x64/ARM environment that most web services cannot offer)

## 4.2 How does it work?

Simple, an IPS file has a static Header reading `PATCH` and a Footer reading `EOF`. The middle of the IPS file is a series of unterminated series of bytes. These "instances" come in two forms in variable size.

**Example of a noRLE Instance:**

`01 23 45 00 05 67 89 AB CD EF`

To make that easier to read, let's break it down. `01 23 45 | 00 05 | 67 89 AB CD EF`

The first three bytes is a `24 bit` number for the target `offset`, the next two bytes is a `16 bit` number for the size of the data. The final series of bytes is the data of which it's length is described by the `16 bit` size number.

**Example of an RLE Instance:**

`01 23 45 00 00 FF FF 64`

The first three bytes are still the `24 bit` target `offset` however you may notice the `16 bit` size bytes are equal to zero, indicating that the data is zero in length. However, when the size is equal to zero we treat this is an `RLE flag` indicating that this `instance` behaves differently.

We read past the two size bytes and take the next `two bytes` as a `16 bit` number describing the `hunk length` of the `RLE` instance. The last byte is the `hunk data` and will be repeated until it's length is equal to the `16 bit` "hunk length".

Spaces in-between offsets beyond the size of the original file size are expected to be zeroes, drastically reducing file size and likely complexity too leading to faster handling. When not storing past the original file the space in between offsets stores the original file contents which are not included in the `ips` for both efficiency and integrity.

## 4.3 How to read/apply one?

Here it will be demonstrated in very simple Python, but annotated well even when the code is verbose.

```
def apply(base : bytes, patch : bytes) -> bytes:
    patch = patch[5:-3]                         #trim header and footer
    changes = {}                                #dictionary to store diffs
    count = 0                                   #create variable used to track progress in
↪"patch"
    while count != len(patch):                  #Until we have read the last byte
        offset = patch[count:count+3]           #read next 3 bytes
        offset = int(offset.hex(),16)           #convert to 24 bit integer
        count += 3
        size = patch[count:count+2]                     #read next two bytes
        size = int(size.hex().16)                       #convert to 16 bit integer
        if not size:                                            #if RLE flag set
            count += 2
            size = patch[count:count+2]             #Acces RLE Length bytes
            size = int(size.hex(),16)               #convert to 16 bit number
            count += 2
            data = patch[count:count+1]
            changes[offset] = (size,data)           #Store RLE instance with offset as␣
↪key
        else:
            data = patch[count:count+size]
```

(continues on next page)

(continued from previous page)

```
            changes[offset] = data                  #Store noRLE instance with offset as␣
↪key
            count += size
    output = b""
    for offset in changes:
        if offset < len(base):                                  #if we are still overwriting
            output += base[len(output):offset]      #Copy base until diff start
        else:
            output += b"\x00" * (offset-len(base))  #Write zeroes until diff start
        if isinstance(changes[offset],tuple):
            output += changes[offset][0]*changes[offset]*1
        else: output += changes[offset]
    output += base[len(output):]                                #if we have not wrote up to␣
↪base, then do so
    return output
```

The code above accepts two *bytes* objects and will return ` byets` object which could be parsed into a *file* object. If you only needed this data for patching then you could :

```
def patchfile(modfile,basefile,outfile):
    def get(File):
        with open(File,"rb") as f:
            return f.read()
    with open(outfile,"wb") as f:
        f.write(patch(get(base),get(mod)))
```

However as *ipsluna* is a module, usage is determined by the user and therefore despite the applications beyond standard usage being nothing short of eccentric does not invalidate the intentions. This is where *iplsuna* exceeds *ips.py*.

**How does `ips` building work?**

*ips* constructing is much more detailed than *ips* applying, as we have to account for the following things:

- `ips` files *should* contain minimal original data.*
- `ips` files *should* not attempt to make an impossibly large file.**
- `ips` files *should* prefer `rle` unless setup is too costly.***
- `ips` files *must* write to the last byte of the new file if bigger , even if zero.

** This does not mean that it won't work, it just means that you may end up creating an unnecessarily large file that contains potentially sensitive data*

*** By default in `patchlib` it is set to `16,777,215 bytes` ( 16.7 MB) however `ips` may reach up to `16,842,750 bytes` by setting `legacy` to `False`*

**** This is merely optimization, no `ips` has to contain `rle` however it should be noted that it is only optimal if the `rle` is of length `9` or higher.*

Now that you know the rules, we can begin to create an `ips` file.

```
def build(base : bytes, target : bytes) -> bytes:
    patch,count = b"", 0

    #Lambdas for operation viability checks
    viability = lambda offset, dist: target[offset].to_bytes(1, "big")*dist ==␣
```

(continues on next page)

(continued from previous page)

```
↪target[offset : offset + dist]
    compare = lambda offset: (base[offset] != target[offset]) if offset < len(base) else␣
↪True

    def rle():              #function for processing rle data
        length = 9
        while compare(count + length) and count + length < len(target) and␣
↪viability(count, length): length += 1
        return length - 1

    def norle():                    #function for processing rle unviable data
        length = 1
        while compare(count + length) and count + length < len(target) and not␣
↪(viability(count + length, 9) and all(compare(count + length + r) for r in range(9))):␣
↪length += 1
        return length

    #while we have not compared the final byte
    while count < len(target):

        #if we are comparing the final byte
        if count == len(target)-1:
        patch += count.to_bytes(3, "big")+b"\x00\x01"+target[count].to_bytes(1, "big")
        count += 1

        #if we have unncessary data
        elif base[count] == target[count] if count < len(base) else target[count] == 0:
            while (base[count] == target[count] if count < len(base) else target[count]␣
↪== 0) if count < len(target) - 1 else False: count += 1

        #now that we have our diff
        else:
            #determinte rle viability
            isrle = viability(count, 9) and all(compare(count + r) for r in range(9))

            length = [norle,rle][isrle]()   #retrieve length to store

            #while length is impossible for a singular instance
            while length > 0xFFFF:
                if isrle: patch += count.to_bytes(3, "big")+b"\x00\x00\xff\xff
↪"+target[count].to_bytes(1, "big")
                else: patch += count.to_bytes(3, "big")+b"\xff\xff
↪"+target[count:count+0xFFFF]
                count += 0xFFFF
                length -= 0xFFFF

            #if data was not a multiple of 0xFFFF
            if length:
                if isrle: patch += count.to_bytes(3, "big")+b"\x00\x00"+length.to_
↪bytes(2, "big")+target[count].to_bytes(1, "big")
                else: patch += count.to_bytes(3, "big")+length.to_bytes(2, "big
↪")+target[count:count+length]
```

(continues on next page)

(continued from previous page)

```
                count += length

    #return data
    return b"PATCH"+patch+b"EOF"
```

This is the *best* `ips` construction code in terms of minimal output and is very optimized.

```
def makepatch(basefile,targetfile,outfile):
    def get(File):
        with open(File,"rb") as f:
            return f.read()
    with open(outfile,"wb") as f:
        f.write(build(get(basefile),get(targetfile)))
```

## 4.4 Why do we sometimes use other patching filetypes?

`bps` for example, uses variable width offsets, and instead of immediate replacement it uses "actions" to move the data and perform selective "range" overwrites in order to achieve a goal with *variable* scope. `ips` has a reach of `16,842,750 bytes`, however a true legal `ips` could not write beyond the `24 bit` maximum and therefore the maximum reach is truly `16,777,215 bytes`.

`ips` also is horribly inefficient at patching large files, some files may contain duplicates of the base code, which is not just horribly inefficient but also provides a security risk for the original file contents. A simple `ips` integrity checker could be constructed to compare base contents to patch contents to see what resemblance there is .

In conclusion, `ips` is designed for an older generation of consoles that were small and simplistic, as the scope of technology gradually increases we may see `bps` become irrelevant. Currently, and for much time, it is irrational to assume that `bps` can be made redundant however as it can reach up to a theoretical `2 exabytes` in reach.

## 4.5 Why do we still use `ips` if better filetypes exist?

Easiest question of them all, `ips` was just there when it needed to be. Because of `ips`'s common usage and popularity when ROMhacking was more niche than it was the filetype has been the face of early ROMhacking, `ips` is actually quite space efficient for most of these hacks, it fit's its scope perfectly.

In some cases, you may opt for `bps` over `ips` if the scope of the project would benefit from it, however for minor edits within the size of the base file there is commonly zero reason not to choose `ips` unless the file you are modding requires a higher reach.

## 4.6 Why should I use `patchlib` over `ips.py`?

The main reason you should choose `patchlib` over `ips.py` is because *it does what ***every*** other **advanced** patching tool does*. After being passed the raw contents of an `ips` or initialising a blank canvas, `patchlib` offers **total** control of the `ips`. Each instance (diff) has the `size`, `data`, `rle flag`, and `diff-reach` stored in the `instance` class as well as a `name` attribute which can be used to annotate an `ips`.

The benefit to all of this is that now we can *smartly* interact with the instances, we can access them with a variety of functions such as `get`, `range``or by accessing the ``instances` attribute within the `ips` class which stores each `instance` by order of `offset`. We can also modify the individual `instance` with the `modify` method.

Moreover, the project is being actively worked on - and updates and new features should be expected. The code exceeds all known IPS tools and is not even at a release build yet, and it has full docs on the PyPI and active developers in immediate contact on the Discord!

## 4.7 Should I make my own `ips` handling tool?

There is very minimal reason to do this. As it stands, even when `ips` filetypes are being manipulated at a deep level, the tools provided are often not even fully used as rarely does the user exceed common `building` and `applying`. There is generally a surplus of tools, should you create your own `ips` tool there should be a reason for this, `patchlib`'s existence is to provide total control in a `Python 3`.

`JIPS` is forgivable as it runs in a Java runtime, meaning that it can run on devices that do not support `Python 3`. Because `JIPS` uses `Java`, the whole ideology being that it can run in *any* environment, this tool is very helpful to those who do not have an Operating System which any dedicated tool can support. The same *would* go for `ips.py` if `patchlib` did not render it redundant. If you wish to make a tool, ensure that the benefits are not found immediately in someone else's tools alone. Once you can confirm there is a point to doing this baring scope, usability and cause, making an `ips` handler makes complete sense.

## 4.8 Can I contribute towards `patchlib`?

Yes! `patchlib` GitHub allows for forks to be made and anyone with some `Python` skill can be included in the Project! In fact, there are many elements of the project left **totally** untouched that you could begin working on! If you are interested feel free in contacting on the Discord!

## 4.9 Is it not better just to make your own filetype?

This should be overall somewhat discouraged for these reasons:

- `ips` is standardized, people may not want to use your files/tools
- It is quite likely that `bps` could solve this, people *will* use that instead
- It creates some sort of proprietary sense to it, which may deter users.
- If tool sharing is too slow for demand, users may share original files

If people do not want to use your tools then the project's popularity will be stunted, if people construct a `bps` between the base and result file then nobody will feel obliged to use *your format or tool*. In the world of common base files it is natural to assume a universal format for manipulation, for this we opt for universal filetypes, limiting control only works for immediate distribution.

CHAPTER

FIVE

# PRODUCER'S NOTICE

Hi there! My name is Brette and I am the Developer of this module and I have a little story to tell if you continue reading about how this project came to be.

I was originally given the task of an auto-patch Discord bot that would process these patches against a comapred ROM, for this I needed a basic understanding of how the `ips` filetype worked and I spent about a week consitently researching this.

After much time into the projects development I had got in contact witht he owner of the Super Mario Bros 3 Modding Discord and suggested that I make a bot which countered Piracy and if a ROM was present, it would create an `ips` file for it, while it has the drawbacks of essentially creating other games (assuming they are larger than Super Mario Bros 3) this is to be expected and appreciated as impossible for me to control.

The original `ips` contrusction code was incredibly crude, producing unoptimized large `ips` files and did not solve the early `EOF` issue in which the offset 4542278 was misinterpreted as the footer which serves to terminate patching. It took me about two weeks to create this bot another week spent on the `ips` construction code and another week spent misunderstanding the `discord.py` API.

After this I had learned much more Python, as of writing this I am yet to have reached adulthood and am a hobbyist developer who as of writing this can only code in Python and MOS 6502 Assembly. During this time period the AI Assistant ChatGPT was released to the public and I soon found a better teacher than I ever had, while it often suggested minor optimizaitons - the effortless nature in which I recieved answers for any queery I had made learning a fast and fun process which improved my worth ethic, mentality and undoubtly the product quality too.

After reviewing my code on both my Discord bots I had concluded that the code was simply inadequate, I am no perfectionist but if there is an identifiable problem then there is a conjurable solution. After at least a dozen attempts at creating optimized build code, countless tests on `instance` handling with all of it's unnecessarily helpful and complicated optional kwargs I had created the best `ips` handler the world has seen.

I have much to be greatful for, if it were not for the incredibly awkardly wrote `ips` filetype docs that I stumbled on when researching for file structure I would never have gotten to where I am. No one should go through the struggle that I did to get where I am, for this I create the best tool that I can and I offet detailed explanatory documentation on the filetype and the module such that no one should ever be confused about this again.

In it's current state, `patchlib` only supports `ips`, however there are immediate plans to approach the different `patch` types with the same ideology and structure as `patchlib.ips`. If you have found this tool helpul whatsoever and have read up until this part, please feel free to join the Discord Server where you can see all of my products, including the aforementioned bots that use this module!

CHAPTER

SIX

# FREQUENTLY ASKED QUESTIONS

**Q: Is this a ROMhacking module?**

A: *No, this module is not purly designed for ROMhacking, while is no doubt would be useful to anyone involving themself with ROMhacking this project has no immediate intended integration with ROMhacking at all. In fact, there are many reasons to use this project outside of modifying Video game Intellectual Property. This filetype could be used to update files in your software or creating a config file modificaiton script*

**Q: Is it legal to ROMhack?**

A: *In answering this I, Brette the creator, cannot be found liable for the potential crimes you commit. However, I can say in full confidence that ROMhacking's legality has always been a grey area, while I cannot think of one person who has been imprisoned for ROMhacking - there have been many times where creators were told that they would have to stop their project due as it is often illegal to create a local copy of a game and modify it*

**Q: If ROMhacking is illegal why is it commonplace?**

A: *Simply, in as much as it is illegal - that does not mean that it must be resolved. While it is true that creating a copy of a game is illegal as long as you do not share the game it is almost like you never created the copy. Likewise these mods also do not contain any illegal data (typically) and therefore are perfectly legal - It is a world that cannot be controlled but in truth is so harmless that there really is no need to*

**Q: How do I use this tool?**

A: *After installing a Python 3 environment you should be able to run teh command* `pip install patchlib` *or if you are using a* `posix` *system you may wish to use* `pip3 install pathlib`. `ips` *files must be processed into an* `ips` *object before handling, after this the methods of the* `ips` *and* `instance` *can be used to manipulate the data as needed.*

**Q: I have found an error in the code, what can I do?**

A: *One of the very helpful things that you can do is commit an update, or if you cannot code you can fill out an* "issue form". *The source code for this project is, however, publicly available and can be accessed by anyone should an immediate change need to be made that you cannot wait for.*

**Q: The Module is slow overall, or with a certain Feature.**

A: *The project is highly optimized where necessary, should you find a way to make it run faster then do so - however this is a Python module and should be expected to be bottlenecked so. If project does not perform fast enough to your liking then you will need to switch to a faster differnet alternative. Currently* `patchlib` *is only a Python module, this may change in the future however there are no current plans for this.*

**Q: I have a suggestion for `patchlib` , how can I be heard?**

A: *If you have something to say about* `patchlib` *then you can speak on the* Discord Server *or* Submit a Feature Request.

CHAPTER

SEVEN

# RESOURCES

As I began ROMhacking and developing this tool I of course became aware of communtiies and pre-existing documentation that I will store in these docs too as they are public domain articles. I will also link websites that have more information on this sort of thing and relevant communities in which you may ask more questions.

| | |
|---|---|
| Download ZeroSoft IPS Guide | Visit Zophar's Domain |
| Download blakesmith BPS Guide | Visit blakesmith's GitHub repository |