Thank you for using the **Large Bitmask System**!

You can also check out my other plugin, **BulletPro**, made for handling any projectile in a 2D game: Link on Asset Store

Table of Contents: *(click on any entry to reach it)*

What is a Large Bitmask?

Sometimes you want to handle bitmasks that are **bigger than 32 bits**, so you cannot use an *enum* or an *uint* variable.
Sometimes you even need something **bigger than 64 bits**, so a *ulong* variable is no more helpful.

This plugin provides you with a single struct named **LargeBitmask**, whose size is entirely up to you. It could be 8 bits, or 80 bits, or 800 bits.

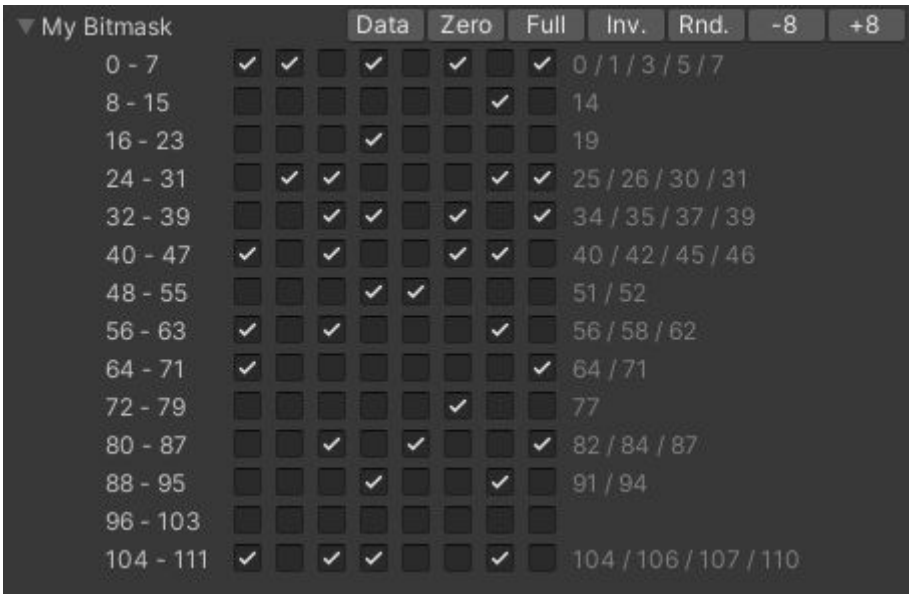
- **It's unlimited and resizable**.
- From the inspector, it has a friendly editing interface (see below).
- From your code, it's easy to edit just like a boolean array :
  `myMask[42] = true;`
- But it's also optimized and compatible with bitwise operations.
  `myMask &= anotherMask;`

Reading the following manual isn't really needed to get started, but it goes in-depth into all the **utility functions** provided by the LargeBitmask struct.

In the inspector:

The inspector of your LargeBitmask variable looks like this:

Unfolding this single line shows the actual bits, that you can turn on or off.



The top buttons function as follows:

- The **Data** button prints info in the console. It says how many bits your mask has, and how many are turned on.
- The **Zero** button sets all bits to false. It unticks all the boxes.
- The **Full** button sets all bits to true. It ticks all the boxes.
- The **Inv.** button toggles all bits. It turns true to false and vice versa.
- The **Rnd.** button gives a random value (on or off) to all the bits.
- The **-8** button deletes the last 8 bits, so your mask gets shorter.
- The **+8** button creates 8 more bits at the end, so your mask gets longer.

Each row represents a byte (which equals 8 bits). Next to each row are written the indexes or all the bits that are set to true.

In your code:

For the following examples, we're going to assume that a `LargeBitmask` named `myMask` has been declared.

```
public LargeBitmask myMask;
```

Operators:

- The LargeBitmask struct is fully compatible with all the bitwise operators.
  | & ^ ~ << >>
- Two variables of type LargeBitmask can also be concatenated with the + operator. 11011 + 10001 = 1101110001.
- LargeBitmask variables can be compared with the usual == and != operators.

Main properties:

- A simple indexer gives you access to the bit as a boolean value.
```
if (myMask[42]) DoStuff();
myMask[42] = true;
```

- The LargeBitmask is serialized as a byte array. You can also directly access the byte array:
```
myMask.bytes
```

- You have multiple ways to know the size (length) of your bitmask:
```
myMask.sizeInBits; // returns myMask.bytes.Length * 8
myMask.sizeInBytes; // returns myMask.bytes.Length
```

Constructor:

There are five different possible ways to initialize a LargeBitmask.

- Simply create an empty LargeBitmask with a set number of bits.
```
int numberOfBits = 641;
myMask = new LargeBitmask(numberOfBits);
// actual bit-size will be rounded up to the next multiple of 8.
// in this example, it would create a 648-bit mask.
```

- Same as above, but also provide an array of ints that say which bits should be set to true.
```
int numberOfBits = 120;
int[] trueBits = new int[]{ 4, 8, 15, 16, 23, 42 };
myMask = new LargeBitmask(numberOfBits, trueBits);
```

- By duplication of an existing LargeBitmask:
```
myMask = new LargeBitmask(anotherMask);
```

- Turning a single byte into an 8-bit LargeBitmask.
```
byte someSingleByte = 137;
myMask = new LargeBitmask(someSingleByte);
```

- Turning a byte array of any size into a LargeBitmask.
```
byte[] someByteArray = new byte[64];
// fill the byte array manually...
myMask = new LargeBitmask(someByteArray);
```

Byte indexing:

It should be noted that in a LargeBitmask, bits from a byte are indexed **from the leftmost bit to the rightmost bit**:

```
byte[] someByteArray = new byte[] { 128, 64, 32 };
myMask = new LargeBitmask(someByteArray);
int[] trueBits = myMask.GetNumberOfTrueBits();
Debug.Log(trueBits[0]); // prints 0
Debug.Log(trueBits[1]); // prints 9
Debug.Log(trueBits[2]); // prints 18
```

Getters:

- `myMask.GetNumberOfTrueBits()` returns how many bits in the mask are set to true.
- `myMask.GetIndexesOfTrueBits()` returns the indexes of all these bits, in an array of int variables.
- `myMask.ToString()` returns a string nicely formatted for debugging purposes.
- `myMask.ToString(true)` enables multiline in the string output.

Setters:

- `myMask.Resize(42);` extends or trims the LargeBitmask so its new size is 42 bits.

- Mass editing:
  ```
  myMask.MakeFalseAll(); // sets every bit to false
  myMask.MakeTrueAll(); // sets every bit to true
  myMask.InvertAll(); // sets every bit to its own opposite
  myMask.RandomizeAll(); // sets every bit to true or false at random
  ```

- Mass editing restricted to a range of bits:
  ```
  myMask.MakeFalseInRange(500, 6); // sets every bit to false from
  index 500 (incl.) to index 506 (excl.)
  myMask.MakeTrueInRange(500, 6);
  myMask.InvertInRange(500, 6);
  myMask.RandomizeInRange(500, 6);
  ```

- Mass editing for an array of specific indexes:
  ```
  int[] indexes = new int[] { 4, 8, 15, 16, 23, 42 };
  myMask.MakeFalseInList(indexes);
  myMask.MakeTrueInList(indexes);
  myMask.InvertInList(indexes);
  myMask.RandomizeInList(indexes);
  ```