

Bitap algorithms used in M0

A flexible macro processor
18 July 2026

by Marco de Beurs
(m0macroprocessor@gmail.com)

This document is for use with M0, a package containing an implementation of the m0 macro language.

Copyright © 2025, 2026 Marco de Beurs

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled “GNU Free Documentation License.”

Table of Contents

1	Introduction	1
1.1	For the reader	1
1.2	Main characteristics of the bitap algorithm.....	1
2	Algorithms in m0.....	3
2.1	Vertical bitap.....	3
2.2	Horizontal bitap.....	4
2.3	Two pass macro recognition	6
2.4	Two pass macro recognition in m0.....	7
2.5	Further possibilities for two pass macro recognition.....	7
3	Bitap for patterns and variable length macros in m0.....	9
3.1	Patterns.....	9
3.2	Zero or one character	9
3.3	One or more characters.....	10
3.4	Pattern part depending on another pattern part.....	12
3.5	Regular expression possibilities using multiple parts.....	12
3.6	Variable length macros	12

1 Introduction

1.1 For the reader

This document provides insight in the algorithms used to detect macros and patterns in the `m0` macro processor.

Useful for the curious and persons who want to use these for similar purposes.

Bitap algorithms are already known in different implementations. It is also known as the shift-or, shift-and or Baeza-Yates–Gonnet algorithm (see e.g. wikipedia Bitap algorithm (https://en.wikipedia.org/wiki/Bitap_algorithm)).

In Chapter 2 [Algorithms in `m0`], page 3, the bitap algorithms used in `m0` in general and for detecting macros are described. In Chapter 3 [Bitap for patterns in `m0`], page 9, the implementation of the bitap algorithm for the patterns is described.

1.2 Main characteristics of the bitap algorithm

In many search algorithms the characters in a text are compared with the characters to be found using a compare instruction. A compare instruction is not used in the bitap algorithm, instead:

The bitap algorithm uses a look-up table instead of a compare.

The advantage of a look-up table is that multiple characters can be checked in a single position with a single look-up. The table has an index the size of the possible values of a character. When using all possible values in a byte the index size is thus 256.

The result of a look-up is either true or false, so that a single bit value is sufficient for the result. To check a word it is only necessary to use a bit AND to combine all the result of the positions of the word and if the result is true, then the word is found.

The following figure illustrates this:

```

search word: wor

text:  | | | |H|e|l|l|o| |w|o|r|l|d| | | |
search      w o r
bits      0 0 0 = 0
search      w o r
bits      0 0 0 = 0
search      w o r
bits      0 0 0 = 0
search      w o r
bits      0 1 0 = 0
search      w o r
bits      0 0 0 = 0
search      w o r
bits      0 0 0 = 0
search      w o r
bits      1 1 1 = 1   found: wor

```

To check if a word has been found:

A bit AND is used on the look-up results to test if the word is found.

This AND is the "and" in the name of the shift-and. The shift is used to put the bits over each other. How this works is explained in the following parts.

2 Algorithms in m0

Two implementations of the bitap algorithm are used in m0. The first is called vertical bitap in this document. The second is called horizontal bitap in this document. This is also the version that is normally discussed in the articles explaining the algorithm.

The following sections introduce these bitap algorithms and their use for macro recognition in m0.

2.1 Vertical bitap

The look-up result only requires a single bit. The word size in computers is normally far larger; nowadays 64 bits is common. It is possible to use all the bits in the cpu word so that multiple words can be searched in one table look-up. With 64 bits in a cpu word, it is thus possible to search up to 64 different search words. Every bit position in the cpu word is used for a search word. This is displayed in the following figure:

cpu word:		1	2	3	4	5	6	...
bit position								
1	search 1:	H	e	l	l	o		
2	search 2:	w	o	r	l	d		
3	search 3:	h	a	i				
4	search 4:	s	o	m	e	t	h	i
5		.						
.		.						
.		.						
64	search 64:	l	a	s	t	o	n	e

The cpu word is displayed in a vertical position in the figure. Therefore this is called the vertical bitap in this document.

What can be seen from using the algorithm in this way is:

- The maximum number of look-ups in a position in the text is depending on the size of the largest search word.
- Up to 64 search words can be checked with a single lookup times the size of the largest search word.
- Every search word can have multiple characters in a single position. This functionality has no performance influence. It is thus easy to use a wildcard (any value) in a position in a search word.
- The size of the complete table = size of index (256) x size of largest word.
- Not depending on any convention to separate words (such as space). It allows to easily search in a part of a word in the text.

In a macro processor a large number of macros could exist. Using this algorithm it would be possible to quickly find all the macros in the text. This algorithm is therefore used in m0, but not without a first pass using the horizontal bitap explained in the next part.

2.2 Horizontal bitap

Multiple times the vertical bitap will need to use the same character in the text for look-up. This is similar to a naive string search, and its efficiency can be improved.

A solution to this is using the bit positions in a cpu word for the search word, instead of using the bit positions for different words.

The following figure illustrates this:

```

search word: wor

text:      | | | |H|e|l|l|o| |w|o|r|l|d| | | |
           |
           v
search      r o w
bits        0 1 0
           |
           v
search      r o w
bits        0 0 0
           |
           v
search      r o w
bits        0 0 1
           |
           v
search      r o w
bits        0 1 0
           |
           v
search      r o w
bits        1 0 0

```

In the figure can be seen that the separate bits of the search word will become true when the character in the text matches. To test if the whole search word has been found it is necessary to check if the bits will sequentially become true. If this is the case up to the last bit, then the search word is found.

This can easily be accomplished by a left shift of the previous result with a bit AND of the new check. What is needed is:

- A result register witch holds the result of the tests.
- A mask for setting the first bit of a word.

The following figure illustrates this:

search word: wor

text: | | | |H|e|l|l|o| |w|o|r|l|d| | | |

|

v

search	r o w
check bits	0 1 0
mask	0 0 1
result << 1	0 0 0
result OR= mask	0 0 1
result AND= check	0 0 0

|

v

search	r o w
check bits	0 0 0
mask	0 0 1
result << 1	0 0 0
result OR= mask	0 0 1
result AND= check	0 0 0

|

v

search	r o w
check bits	0 0 1
mask	0 0 1
result << 1	0 0 0
result OR= mask	0 0 1
result AND= check	0 0 1

|

v

search	r o w
check bits	0 1 0
mask	0 0 1
result << 1	0 1 0
result OR= mask	0 1 1
result AND= check	0 1 0

|

v

search	r o w
check bits	1 0 0
mask	0 0 1
result << 1	1 0 0
result OR= mask	1 0 1
result AND= check	1 0 0

~

search word found

The cpu word is displayed in a horizontal position in this figure. Therefore this is called the horizontal bitap in this document.

What can be seen from using the algorithm in this way is:

- Only one look-up per position in the text.
- Up to 64 characters divided over several words can be checked with a single lookup using a 64 bit cpu word.
- Every search word can have multiple characters in a single position. This functionality has no performance influence. It is thus easy to use a wildcard (any value) in a position in a search word.
- The size of the complete table = size of index (256).
- Not depending on any convention to separate words (such as space). It allows to easily search in a part of a word in the text.

This algorithm is used in m0 for a first pass for recognizing macros and directly in patterns. The following parts will show how the two versions are combined in m0.

2.3 Two pass macro recognition

For performant macro recognition in a macro processor it is necessary to be able to recognise a large number of macros and also to not do too much unnecessary work.

The horizontal bitap is performant, but only if the macros fit into one or a limited number of cpu words. This will often not be the case.

The vertical bitap can hold a large number of macros, but performance is throttled by unnecessary checks. Similar improvements as used for a naive search could be used. However for m0 a solution is chosen to combine both bitaps in a two pass method.

For the first pass all search words with the same length are combined in a cpu word. This makes use of the ability to have multiple possible characters in a single position. Multiple combined words of different length are put in the cpu word.

This is illustrated in the following figure with the example search words #, `, aa, bb, hai, two, one, four, hello:

size of search word		5	4	3	2 1
cpu word					
examples	o	l	l	e	h r
					u
					o
					f i
					a
					h a
					a a
					#
					o
					w
					t b
					b `
					e
					n
					o

In this example 15 characters of the cpu word are used to hold all search words with a length up to 5. In a single 64 bit cpu word search words with a length up to 10 can be put ($1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 = 55$).

However it is not clear which macro was recognised if multiple search words with the same length are searched. Besides this, the above method leads to false triggers. For example "ab" or "ba" will in the example above also trigger the search word with length 2.

When this first pass triggers, the length of the search word and the position in the text are known. The second pass using the vertical bitap can then efficiently find the macro or check for a false trigger.

The two pass method can thus be summarized as:

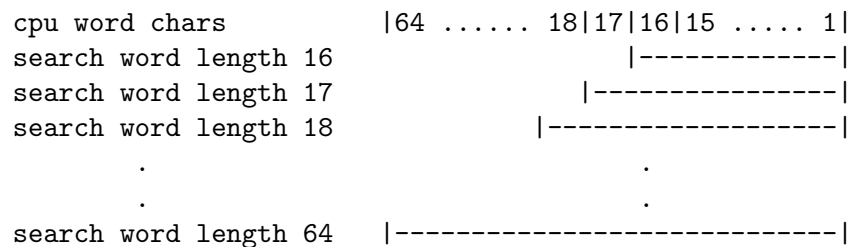
- first a quick search for a probable match,
- secondly finding a precise match or false trigger.

2.4 Two pass macro recognition in m0

Up to a word length of 10 can be put in a single 64 bit cpu word using the two pass method. Using two 64 bit cpu words results in lengths up to 15 ($11 + 12 + 13 + 14 + 15 = 65$, $55 + 65 = 115$) by properly dividing the lengths over the two words.

In m0 macros can have a length up to 64 characters. Using more cpu words to fit all these, results in too many cpu words to be efficient. Most macros will be relatively short and be shorter than 15 characters. Using cpu words for words up to 64 characters length is thus also wasting a lot of space and processing time.

In m0 a third cpu word is used to hold all words with a length of 16 up to 64. These words overlap each other from the lower characters. This solution is illustrated in the following figure:



This leads to a lot of false triggers, but this should not be a big issue if not many macros with a length of more than 15 are used, or if the text does not have a lot of these larger macros.

The vertical bitap is split up in two sets; one for words with a length of up to 15 and a second for words with a length of 16 up to 64.

Of course also other choices can be made regarding the number of cpu words for the first pass and the number of sets for the the vertical bitap.

2.5 Further possibilities for two pass macro recognition

The choices described above lead to:

- 3 cpu words, 2 cpu words for words with length up to 15 and 1 for length 16 – 64.
- 2 vertical bitap sets, one for words up to 15 length and one for words with length 16 – 64.

This is using a relative small amount of memory for holding the macros. In this document this small set is named small-15.

It is also possible to use a vertical bitap set separately for every length of the words. This is named large in this document. If still using the two cpu words for length up to 15, then this is named large-15.

It is also possible to use 1 vertical bitap for 1 cpu word. With again the 2 cpu words for length up to 15 this is named medium-15.

Extending these possibilities leads to the following solutions for different usages:

up to length separate words	small	medium	large
10			cpu words: 2 vertical : 11
15	cpu words: 3 vertical : 2	cpu words: 3 vertical : 3	cpu words: 3 vertical : 16
22	cpu words: 5 vertical : 2	cpu words: 5 vertical : 5	cpu words: 5 vertical : 23
27		cpu words: 7 vertical : 7	cpu words: 7 vertical : 29
31		cpu words: 9 vertical : 9	cpu words: 9 vertical : 32
64			cpu words: 33 vertical : 64

Which choice will be the most performant will depend mostly on the workload. For a huge number of macros in a large text with many macros the large choice will probably be most performant. The performance will also depend on the hardware; a large set requires a large amount of memory that might induce more cpu cache misses and reduce performance.

Till so far m0 uses only the small-15 choice.

3 Bitap for patterns and variable length macros in m0

The patterns used in m0 are normally not large and can fit easily in one cpu word. Therefore the horizontal bitap is sufficient for use in the patterns. The horizontal bitap also allows possibilities that are not possible with the two pass method for macros.

The following sections explain the use of the bitap algorithm for the patterns in m0.

3.1 Patterns

A pattern is a collection of pattern parts. The pattern parts are the search words that must be recognised to trigger a program in m0. The pattern parts can thus be easily used in the horizontal bitap.

This gives the characteristics of the horizontal bitap, which are the most relevant for the patterns:

- Only one look-up per position in the text (good performance).
- Up to 64 characters divided over several words can be checked with a single lookup using a 64 bit cpu word (large enough for most patterns).
- Every search word can have multiple characters (also wildcards) in a single position.
- Not depending on any convention to separate words (such as space).

These are nice features for patterns, but can be extended for more functionality. The following parts show how extra functionalities are added to the horizontal bitap.

3.2 Zero or one character

The possibility to have a character or not on a position is implemented by a mask and an extra shift. The shift in the bitap will normally bring the result from the previous character. If this previous character should have the possibility to not exist, then the result of the character before the previous character should also be used.

This is shown in the following figure using a search for "a[a]?b", whereby the "[a]?" can be a single or no "a" character:

```

search word: a[a]?b

text:                | | |a|b|b| | |
                    |
                    v
search               b ? a
check bit            0 1 1
mask                 0 0 1
mask_z               0 1 0
result << 1           0 0 0
result_z = result AND mask_z 0 0 0
result_z << 1         0 0 0
result OR= mask      0 0 1
result OR= result_z  0 0 1
result AND= check    0 0 1      found the a
                    |
                    v
search               b ? a
check bits           1 0 0
mask                 0 0 1
mask_z               0 1 0
result << 1           0 1 0
result_z = result AND mask_z 0 1 0      the previous a
result_z << 1         1 0 0      shifted the previous a
result OR= mask      0 1 1
result OR= result_z  1 1 1      the previous a sets bit
result AND= check    1 0 0
                    ~
                    found ab

```

Thus the additional instructions to do for the zero or one check are:

- Select the correct bit using an additional mask
- Shift this additional result one further (thereby jumping over one character).
- Set the bit in the result using this additional result before the final AND of the bitap.

3.3 One or more characters

The possibility to have one or more characters in a single position uses a similar method as the zero or one character, but needs a latched memory. This latched memory should be true so long as the character in that position is still correct.

This is shown in the following figure using a search for "a[c]*b", whereby the "[c]*" can be a single or multiple "c" characters:

```

text:                | | |a|c|c|b | |
                    |
                    v
search               b * a
check bit           0 0 1
mask                0 0 1
mask_m              0 1 0
result << 1          0 0 0
result OR= mask      0 0 1
result OR= mem << 1  0 0 1    memory still empty
result AND= check    0 0 1    found a
mem AND= check        0 0 0
mem OR= result AND mask_m 0 0 0

                    |
                    v
search               b * a
check bit           0 1 0
mask                0 0 1
mask_m              0 1 0
result << 1          0 1 0
result OR= mask      0 1 1
result OR= mem << 1  0 1 1
result AND= check    0 1 0
mem AND= check        0 0 0
mem OR= result AND mask_m 0 1 0    memory for c is true

                    |
                    v
search               b * a
check bit           0 1 0
mask                0 0 1
mask_m              0 1 0
result << 1          1 0 0
result OR= mask      1 0 1
result OR= mem << 1  1 0 1
result AND= check    0 0 0    acb is not found
mem AND= check        0 1 0
mem OR= result AND mask_m 0 1 0    memory for c is still true

                    |
                    v
search               b * a
check bit           1 0 0
mask                0 0 1
mask_m              0 1 0
result << 1          0 0 0
result OR= mask      0 0 1
result OR= mem << 1  1 0 1
result AND= check    1 0 0    found accb
mem AND= check        0 0 0    memory is reset
mem OR= result AND mask_m 0 0 0

```

Thus the additional instructions to do for the one or multiple check are:

- Set the bit in the result using the memory before the final AND of the bitap.
- Reset the memory when character is not detected.
- Set the memory using an mask for the memory and when the character has been detected.

A zero or more characters can be detected by combining the zero or one and the one or more detection.

3.4 Pattern part depending on another pattern part

From the previous explanations can be seen that the shift and the AND are the main characteristic of the bitap to sequentially test the characters. The start of a word and thus of a sequence is done by setting a start bit. This is normally done using a bit mask (init bitmask), and can also be done in a different way as the zero or one and the one or more characters show.

It is also possible to start a pattern part when another pattern part is triggered. This can be implemented using a bit mask. In `m0` it is implemented by directly placing the depending pattern part behind the previous part. The default shift will in this case start the depending part, because the start bit is not set by the init bitmask. The two pattern parts are thus working like one word.

This method of starting a pattern part can be extended. The next part discusses this.

3.5 Regular expression possibilities using multiple parts

By using a mask with start bits, which is activated by a triggered pattern part, it is possible to make many different links between pattern parts. Thus when a certain pattern part is triggered, it starts one or multiple other pattern parts or even itself by setting the correct starting bits.

This functionality allows to make the functionality of regular expressions possible using the bitap as described till so far.

3.6 Variable length macros

The variable length macros in `m0` use the same bitap algorithm as the patterns. This enables the variable length macros and gives these macros the same possibilities as the patterns.

It is expected that not too much variable length macros are normally used and that therefore the number of cpu words that is needed is limited.