

CAN bus and stm32 - part two - filters and two CAN

STM32



Hello.

The first part ended with the fact that I promised to talk about setting up filters and the operation of two CAN interfaces on one stone. Let's start with filters.

Filters

As mentioned earlier, each CAN interface has 14 filters. *Filters are also called filter banks or simply banks.* Each filter has a serial number (it counts from zero), and consists of two 32-bit registers (hereinafter I will call them "filter registers", the first and second). Each filter has its own "filter registers".

In a nutshell, the filter works like this: we write a packet identifier to the first "filter register", and a certain mask to the second, as a result of which CAN will only accept a certain range of frames, and discard all rest. This is the first way. The second way: in one or both "filter registers" we prescribe the specific frame identifiers that we want to receive - everything else will be discarded.

The filter cannot be configured as exclusive, that is, if you want to accept all frames except for one, for example 0x0296, you will have to configure two filters so that they accept frames from 0x0000 to 0x0295 and from 0x0297 to 0x07FF.

For flexible filtering settings, there are four options for using "filter registers" for each filter. The picture from the reference manual illustrates this ...

32-Bit Filter - Identifier Mask

ID	CAN_FxR1[31:24]	CAN_FxR1[23:16]	CAN_FxR1[15:8]	CAN_FxR1[7:0]			
Mask	CAN_FxR2[31:24]	CAN_FxR2[23:16]	CAN_FxR2[15:8]	CAN_FxR2[7:0]			
Mapping	STID[10:3]	STID[2:0]	EXID[17:13]	EXID[12:5]	EXID[4:0]	IDE	RTR

32-Bit Filters - Identifier List

ID	CAN_FxR1[31:24]	CAN_FxR1[23:16]	CAN_FxR1[15:8]	CAN_FxR1[7:0]			
ID	CAN_FxR2[31:24]	CAN_FxR2[23:16]	CAN_FxR2[15:8]	CAN_FxR2[7:0]			
Mapping	STID[10:3]	STID[2:0]	EXID[17:13]	EXID[12:5]	EXID[4:0]	IDE	RTR

16-Bit Filters - Identifier Mask

ID	CAN_FxR1[15:8]	CAN_FxR1[7:0]					
Mask	CAN_FxR1[31:24]	CAN_FxR1[23:16]					
ID	CAN_FxR2[15:8]	CAN_FxR2[7:0]					
Mask	CAN_FxR2[31:24]	CAN_FxR2[23:16]					
Mapping	STID[10:3]	STID[2:0]	RTR	IDE	EXID[17:15]		

16-Bit Filters - Identifier List

ID	CAN_FxR1[15:8]	CAN_FxR1[7:0]					
ID	CAN_FxR1[31:24]	CAN_FxR1[23:16]					
ID	CAN_FxR2[15:8]	CAN_FxR2[7:0]					
ID	CAN_FxR2[31:24]	CAN_FxR2[23:16]					
Mapping	STID[10:3]	STID[2:0]	RTR	IDE	EXID[17:15]		

x = filter bank number
ID=Identifier

I have separated the use cases of "filter registers" with red stripes.

The first and second options are suitable for filtering both standard (11 bit) and extended (29 bit) identifiers and the third and fourth are only for standard ones. Now let's look at all this in detail.

The first option is 32-Bit Filter - Identifier Mask

I warn you in advance - it will be difficult, so get ready 🤔

In the first "filter register" (ID on the picture) the frame identifier is written, and in the second (Mask on the picture) mask that will be applied to this identifier and compared with this identifier. It looks confusing, but

will be accepted (*not discarded*) .

Since our CAN interface accepts all frames, and then discards by hardware those that do not match the filter settings, then in the text below, by the word “accept”, I will mean that the frame was not discarded.

For example, we will configure filter No. 0.

Specify the filter number ...

```
sFilterConfig.FilterBank = 0;
```

Specify the filter mode...

```
sFilterConfig.FilterMode = CAN_FILTERMODE_IDMASK;
```

ID and mask mode. I draw your attention to this parameter, if you write another macro here (CAN_FILTERMODE_ IDLIST), then the filter will work differently. Since these macros are similar, be careful.

Specify the scale (*dimension*) of the filter ...

```
sFilterConfig.FilterScale = CAN_FILTERSCALE_32BIT;
```

32 bits, indicates that either standard (11 bits) identifiers or extended (29 bits) identifiers can be filtered.

Further, in the last part, we wrote the following code ...

```
sFilterConfig.FilterIdHigh = 0x0000;    // старшая часть первого "регистра фильтра"
sFilterConfig.FilterIdLow  = 0x0000;    // младшая часть первого "регистра фильтра"

sFilterConfig.FilterMaskIdHigh = 0x0000; // старшая часть второго "регистра фильтра"
sFilterConfig.FilterMaskIdLow  = 0x0000; // младшая часть второго "регистра фильтра"
```

These are two “filter registers”, each of which is conditionally divided into a high and low part. Zeros are *written everywhere to receive all frames.*

Now we will configure the filter to work with standard frame IDs, that is, with 11-bit IDs.

Despite the fact that the filter dimension is 32 bits (CAN_FILTERSCALE_32BIT), we can configure the fi
to work with both standard (11 bits) and extended (29 bits) identifiers. If we had specified a filter size of

discussed below.

Suppose we want to accept ids from **0x0100** to **0x0107**, then in the upper part of the first "filter register" write the initial identifier, with a shift ...

```
sFilterConfig.FilterIdHigh = 0x0100<<5; // старшая часть первого "регистра фильтра"
sFilterConfig.FilterIdLow = 0x0000;    // младшая часть первого "регистра фильтра"
```

We write zeros in the lower part.

The shift is done because we have an 11-bit identifier, and according to the figure, the upper part is allocated for it, the upper part of the "filter register". This is how "butter oil" turned out. 😊

This is the area...

32-Bit Filter - Identifier Mask

ID	CAN_FxR1[31:24]	CAN_FxR1[23:16]	CAN_FxR1[15:8]	CAN_FxR1[7:0]			
Mask	CAN_FxR2[31:24]	CAN_FxR2[23:16]	CAN_FxR2[15:8]	CAN_FxR2[7:0]			
Mapping	STID[10:3]	STID[2:0]	EXID[17:13]	EXID[12:5]	EXID[4:0]	IDE	RTR

→

←

Mapping shows that there should be a standard identifier - STDID - eight bits in the first octet (left) and *it* in the second.

That is, it turned out like this ...

32-Bit Filter - Identifier Mask

ID	0 0 1 0 0 0 0 0	0 0 0	CAN_FxR1[15:8]		CAN_FxR1[7:0]							
Mask	CAN_FxR2[31:24]		CAN_FxR2[23:16]		CAN_FxR2[15:8]		CAN_FxR2[7:0]					
Mapping	STID[10:3]		STID[2:0]		EXID[17:13]		EXID[12:5]		EXID[4:0]		IDE	RTR

→

←

In binary format, the number **0x0100** looks like this - **0000000100000000**, and since the number of standard identifiers cannot exceed **0x07FF** (0000011111111111), no one needs the first five zeros, there they are "thrown out into the cold".

In general, you don't really need all these pictures with the distribution of zeros and ones, however, they be useful for general development and simplification of understanding.

Let's move on to the upper part of the second "filter register". Here we write (again with a shift) the number **0x07F8** ...

```
sFilterConfig.FilterMaskIdHigh = 0x07F8<<5; // старшая часть второго "регистра фильтра"
sFilterConfig.FilterMaskIdLow = 0x0000;    // младшая часть второго "регистра фильтра"
```

This is a twist, an inexperienced reader will say, probably the author mixed something up. But no, everything is correct, and now we will deal with it.

In the low part of the second “filter register”, we again wrote a zero, and the number 0x07F8 (0000011111111000) went into the high part of the second “filter register” like this ...

32-Bit Filter - Identifier Mask

ID	0	0	1	0	0	0	0	0	0	0	0	CAN_FxR1[15:8]	CAN_FxR1[7:0]			
Mask	1	1	1	1	1	1	1	1	0	0	0	CAN_FxR2[15:8]	CAN_FxR2[7:0]			
Mapping	STID[10:3]					STID[2:0]		EXID[17:13]				EXID[12:5]		EXID[4:0]	IDE	RTR

The first (left) five zeros are again thrown out, for the reasons described above.

Thus, the number 0x07F8 is our mask. *Looking ahead a little, I will say that this number must be selected independently, experimentally.* Now let's understand how it works.

Now see what happens. When some identifier arrives, it enters the filtering system, and the following happens: a bitwise operation “AND” (*aka AND, aka &*) is performed between the arriving identifier and the mask, and then this result is compared with what is written in “first register” of the filter, in our case it is number **0x0100**. If the result matches, then the identifier (*frame with this identifier*) is accepted, if not, it is discarded.

To clarify this mechanism, you can describe this mechanism with a simple program ...

```
#include <stdio.h>
#include <stdint.h>

int main()
{
    uint32_t ID = 0x0100;
    uint32_t Mask = 0x07F8;

    for(uint32_t i = 0; i < 0x0800; i++)
    {
        if((i & Mask) == ID)
        {
            printf("0x%04X\n", i);
        }
    }

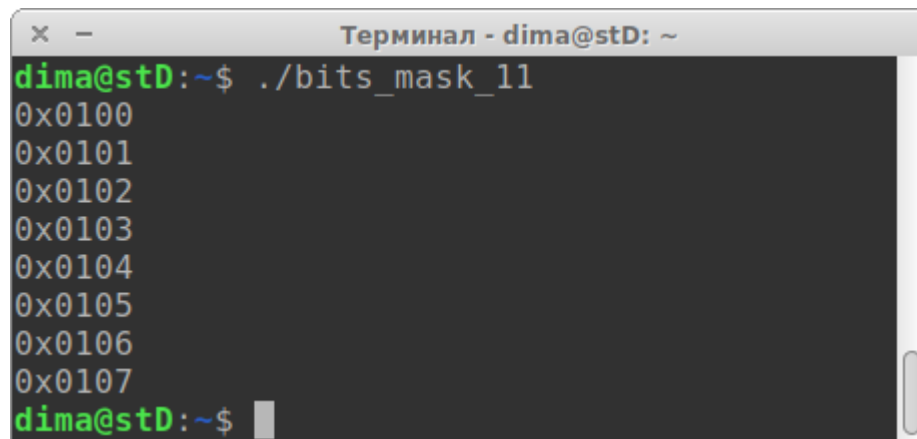
    return 0;
}
```

ID- this is what we wrote in the first "filter register".

Mask— the mask written in the second "filter register"

Cycleforsimulates arriving identifiers, from 0 to 0x07FF (0x0800 - 1, *maximum number of standard identifiers*). In the condition if the arrived identifier is masked, and if the result is equal to 0x0100, the arrived identifier will be accepted by CAN (*printed*) .

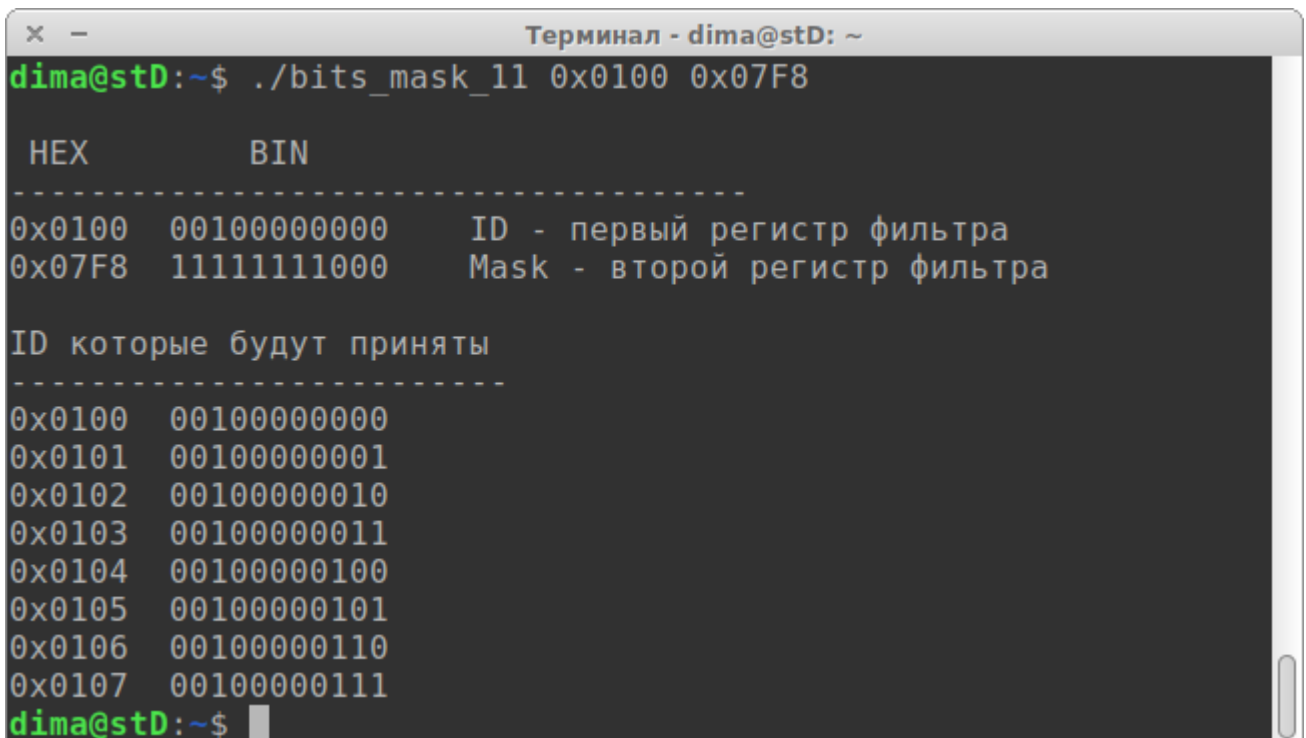
The result of the program will be this output ...



```
Терминал - dima@stD: ~
dima@stD:~$ ./bits_mask_11
0x0100
0x0101
0x0102
0x0103
0x0104
0x0105
0x0106
0x0107
dima@stD:~$
```

Identifiers from **0x0100** to **0x0107** that we planned to accept.

By the link you can download [a program](#) that displays values in HEX and binary format ...



```
Терминал - dima@stD: ~
dima@stD:~$ ./bits_mask_11 0x0100 0x07F8

  HEX      BIN
-----
0x0100  001000000000    ID - первый регистр фильтра
0x07F8  11111111000    Mask - второй регистр фильтра

ID которые будут приняты
-----
0x0100  001000000000
0x0101  001000000001
0x0102  001000000010
0x0103  001000000011
0x0104  001000000100
0x0105  001000000101
0x0106  001000000110
0x0107  001000000111
dima@stD:~$
```

With its help, you can use the "poke" method to select a mask. *The program is written for Linux, but should compile for Windows as well. Or just transfer it to stm.*

If we change the last digit in the mask from 0x07F8 to 0x07FC, then our CAN will accept identifiers from 0x0100 to 0x0103 ...

```
Терминал - dima@stD: ~
dima@stD:~$ ./bits_mask_11 0x0100 0x07FC

HEX      BIN
-----
0x0100    001000000000    ID - первый регистр фильтра
0x07FC    11111111100    Mask - второй регистр фильтра

ID которые будут приняты
-----
0x0100    001000000000
0x0101    001000000001
0x0102    001000000010
0x0103    001000000011
dima@stD:~$
```

It is not necessary to configure the filter so that the received identifiers are sequential in ascending order you can do whatever you like, it all depends on the ID and Mask. For example, if you write 0x0200 in the ID and 0x03FD in the mask, then the identifiers 0x0200, 0x0202, 0x0600 and 0x0602 will be accepted ...

```
Терминал - dima@stD: ~
dima@stD:~$ ./bits_mask_11 0x0200 0x03FD

HEX      BIN
-----
0x0200    010000000000    ID - первый регистр фильтра
0x03FD    01111111101    Mask - второй регистр фильтра

ID которые будут приняты
-----
0x0200    010000000000
0x0202    010000000010
0x0600    110000000000
0x0602    110000000010
dima@stD:~$
```

To understand how to select a mask, take a look at the last picture. If the bit in the mask is equal to one, then the corresponding bit in the arriving identifier must also be equal to one. If the bit in the mask is equal to zero, then it does not matter what the corresponding bit in the arriving identifier is equal to.

The first bit in the mask (from left to right) is zero, so it doesn't matter what the first bit of the identifiers that will be accepted can be zero or one. The second bit in the mask is one, so only identifiers with a second bit equal to one are accepted. The tenth bit in the mask is zero, which means it doesn't matter what this bit is for the identifiers that will be accepted.

To reinforce understanding. let's make a mask in which all bits except the last three will be equal to one

```
Терминал - dima@stD: ~
dima@stD:~$ ./bits_mask_11 0x0200 0x07F8

HEX          BIN
-----
0x0200  010000000000  ID - первый регистр фильтра
0x07F8  11111111000  Mask - второй регистр фильтра

ID которые будут приняты
-----
0x0200  010000000000
0x0201  010000000001
0x0202  010000000010
0x0203  010000000011
0x0204  010000000100
0x0205  010000000101
0x0206  010000000110
0x0207  010000000111
dima@stD:~$
```

Voila. We now accept IDs in the range 0x0200 to 0x0207. Note that the mask is the same as for identifiers 0x0100 through 0x0107. It doesn't mean anything, it's just a word.

In general, I think more or less everything is clear. Take some identifier, come up with a mask, feed it to program and see what identifiers will be accepted.

Now let's mentally return to the beginning of the article, remember that we wanted to accept identifiers from **0x0100** to **0x0107**, and finish setting up filter #0.

This will result in the following code...

```
sFilterConfig.FilterBank = 0;
sFilterConfig.FilterMode = CAN_FILTERMODE_IDMASK;
sFilterConfig.FilterScale = CAN_FILTERSCALE_32BIT;
sFilterConfig.FilterIdHigh = 0x0100<<5; // старшая часть первого "регистра фильтра"
sFilterConfig.FilterIdLow = 0x0000; // младшая часть первого "регистра фильтра"
sFilterConfig.FilterMaskIdHigh = 0x07F8<<5; // старшая часть второго "регистра фильтра"
sFilterConfig.FilterMaskIdLow = 0x0000; // младшая часть второго "регистра фильтра"
sFilterConfig.FilterFIFOAssignment = CAN_RX_FIFO0;
sFilterConfig.FilterActivation = ENABLE;

if(HAL_CAN_ConfigFilter(&hcan, &sFilterConfig) != HAL_OK)
{
    Error_Handler();
}
```


you remember, CAN has two receive buffers CAN_RX_FIFO0 and CAN_RX_FIFO1. In this case, we specified CAN_RX_FIFO0, which means that all received frames that passed through filter No. 0 will fall into CAN_RX_FIFO0. It follows that by setting up several filters, we can specify which buffer the received frames will be dumped into. Now let's set up another filter.

Let's take the last example and we will accept identifiers from **0x0200** to **0x0207** through filter #1...

```
sFilterConfig.FilterBank = 1;
sFilterConfig.FilterIdHigh = 0x0200 << 5; // старшая часть первого "регистра фильтра"
sFilterConfig.FilterIdLow = 0x0000;       // младшая часть первого "регистра фильтра"
sFilterConfig.FilterMaskIdHigh = 0x07F8 << 5; // старшая часть второго "регистра фильтра"
sFilterConfig.FilterMaskIdLow = 0x0000;    // младшая часть второго "регистра фильтра"

if(HAL_CAN_ConfigFilter(&hcan, &sFilterConfig) != HAL_OK)
{
    Error_Handler();
}
```

We indicate only the filter number and write the values into the "filter registers". All other settings will be automatically taken from the settings of filter No. 0.

Thus, the initialization of CAN and filters will look like this ...

```

...
/* USER CODE BEGIN CAN_Init 0 */
CAN_FilterTypeDef sFilterConfig;
/* USER CODE END CAN_Init 0 */

hcan.Instance = CAN1;
hcan.Init.Prescaler = 4;
hcan.Init.Mode = CAN_MODE_NORMAL;
hcan.Init.SyncJumpWidth = CAN_SJW_1TQ;
hcan.Init.TimeSeg1 = CAN_BS1_13TQ;
hcan.Init.TimeSeg2 = CAN_BS2_2TQ;
hcan.Init.TimeTriggeredMode = DISABLE;
hcan.Init.AutoBusOff = ENABLE;
hcan.Init.AutoWakeUp = DISABLE;
hcan.Init.AutoRetransmission = ENABLE;
hcan.Init.ReceiveFifoLocked = DISABLE;
hcan.Init.TransmitFifoPriority = ENABLE;

if (HAL_CAN_Init(&hcan) != HAL_OK)
{
    Error_Handler();
}

/* USER CODE BEGIN CAN_Init 2 */
sFilterConfig.FilterBank = 0;
sFilterConfig.FilterMode = CAN_FILTERMODE_IDMASK;
sFilterConfig.FilterScale = CAN_FILTERSCALE_32BIT;
sFilterConfig.FilterIdHigh = 0x0100<<5; // старшая часть первого "регистра фильтра"
sFilterConfig.FilterIdLow = 0x0000; // младшая часть первого "регистра фильтра"
sFilterConfig.FilterMaskIdHigh = 0x07F8<<5; // старшая часть второго "регистра фильтра"
sFilterConfig.FilterMaskIdLow = 0x0000; // младшая часть второго "регистра фильтра"
sFilterConfig.FilterFIFOAssignment = CAN_RX_FIFO0;
sFilterConfig.FilterActivation = ENABLE;

if(HAL_CAN_ConfigFilter(&hcan, &sFilterConfig) != HAL_OK)
{
    Error_Handler();
}

sFilterConfig.FilterBank = 1;
sFilterConfig.FilterIdHigh = 0x0200 << 5; // старшая часть первого "регистра фильтра"
sFilterConfig.FilterIdLow = 0x0000; // младшая часть первого "регистра фильтра"
sFilterConfig.FilterMaskIdHigh = 0x07F8 << 5; // старшая часть второго "регистра фильтра"
sFilterConfig.FilterMaskIdLow = 0x0000; // младшая часть второго "регистра фильтра"

if(HAL_CAN_ConfigFilter(&hcan, &sFilterConfig) != HAL_OK)
{
    Error_Handler();
}

```

If we want to put frames passed through filter #1 into the CAN_RX_FIFO1 buffer, then this must be

everywhere so as not to get confused, it will not be worse.

```
sFilterConfig.FilterBank = 1;
sFilterConfig.FilterIdHigh = 0x0200 << 5; // старшая часть первого "регистра фильтра"
sFilterConfig.FilterIdLow = 0x0000; // младшая часть первого "регистра фильтра"
sFilterConfig.FilterMaskIdHigh = 0x07F8 << 5; // старшая часть второго "регистра фильтра"
sFilterConfig.FilterMaskIdLow = 0x0000; // младшая часть второго "регистра фильтра"

sFilterConfig.FilterFIFOAssignment = CAN_RX_FIFO1;

if(HAL_CAN_ConfigFilter(&hcan, &sFilterConfig) != HAL_OK)
{
    Error_Handler();
}
```

Thus, frames from **0x0100** to **0x0107** will fall into the **CAN_RX_FIFO0** buffer, and frames from **0x0200** to **0x0207** into the **CAN_RX_FIFO1** buffer.

Further, if there is a need, you can set up as many more filters as you need - the maximum is No. 13.

The filter mode (**CAN_FILTERMODE_IDMASK**) and dimension (**CAN_FILTERSCALE_32BIT**) can also be configured for each filter separately, but more on that later.

Now consider filtering 29-bit identifiers. You remember that we are still analyzing the first use case of "filter registers", which involves filtering 11 and 29 bit identifiers 🤔

Since the filter mode and size settings are the same here as in the previous examples, we will only configure the filter number and "filter registers", and specify the **CAN_RX_FIFO1** buffer (can be **CAN_RX_FIFO0**, if you like). Of course, if you are setting up only one filter, then you need to write all the values. In this case, it doesn't matter at all what filter number you specify, you can use any from 0 to 13.

Let's set up filter #2, and we will accept frames with identifiers in the range from **0x00010000** to **0x00010007** ...

```
sFilterConfig.FilterBank = 2;
sFilterConfig.FilterIdHigh = (uint16_t)(0x00010000 >> 13); // старшая часть первого "регистра
sFilterConfig.FilterIdLow = (uint16_t)(0x00010000 << 3) | 0x04; // младшая часть первого "регистра
sFilterConfig.FilterMaskIdHigh = (uint16_t)(0x1FFFFFF8 >> 13); // старшая часть второго "регистра
sFilterConfig.FilterMaskIdLow = (uint16_t)(0x1FFFFFF8 << 3) | 0x04; // младшая часть второго "регистра

sFilterConfig.FilterFIFOAssignment = CAN_RX_FIFO1;

if(HAL_CAN_ConfigFilter(&hcan, &sFilterConfig) != HAL_OK)
{
    Error_Handler();
}
```

Here, we use the "filter registers" in full, a shift is also made so that the bits are in the right places, and | added to the lower parts | 0x04.

ID and Mask will fall into the STID and EXID area and a unit will be written to the **IDE**

32-Bit Filter - Identifier Mask

ID	CAN_FxR1[31:24]		CAN_FxR1[23:16]		CAN_FxR1[15:8]		CAN_FxR1[7:0]					
Mask	CAN_FxR2[31:24]		CAN_FxR2[23:16]		CAN_FxR2[15:8]		CAN_FxR2[7:0]					
Mapping	STID[10:3]		STID[2:0]		EXID[17:13]		EXID[12:5]		EXID[4:0]		IDE	RTR

bit using | 0x04 (00000100) . This will tell the system that the filter is configured to work with an extended identifier. *That is, whenever we set up a filter to work with an extended frame, we add | 0x04. We don't touch the RTR bit* , but if we want to configure the filter to work with Remote Frame , then we need to write one to it. This applies to both standard and extended frames.

Operations with ID and mask here are exactly the same as in the previous example, and here is the same program , only for 29-bit identifiers ...

```

Терминал - dima@stD: ~
dima@stD:~$ gcc -Wextra -Wall bits_mask_29.c -o bits_mask_29
dima@stD:~$ ./bits_mask_29 0x00010000 0x1FFFFFFF8

HEX          BIN
-----
0x00010000  00000000000001000000000000000000  ID - первый регистр фильтра
0x1FFFFFFF8  11111111111111111111111111111000  Mask - второй регистр фильтра

ID которые будут приняты
-----
0x00010000  00000000000001000000000000000000
0x00010001  00000000000001000000000000000001
0x00010002  00000000000001000000000000000010
0x00010003  00000000000001000000000000000011
0x00010004  00000000000001000000000000000100
0x00010005  00000000000001000000000000000101
0x00010006  00000000000001000000000000000110
0x00010007  00000000000001000000000000000111
dima@stD:~$

```

Then you already know what to do with it.

You can set up multiple filters for extended frames, and combine them with the standard ones, as we just did. Now our CAN accepts standard identifiers from 0x0100 to 0x0107 and from 0x0200 to 0x0207, addi

CAN_RX_FIFO1.

On this, the first option is completed, the most difficult is behind, then it will be easy.

The second option is 32-Bit Filters - Identifier List

32-Bit Filters - Identifier List

ID	CAN_FxR1[31:24]	CAN_FxR1[23:16]	CAN_FxR1[15:8]	CAN_FxR1[7:0]			
ID	CAN_FxR2[31:24]	CAN_FxR2[23:16]	CAN_FxR2[15:8]	CAN_FxR2[7:0]			
Mapping	STID[10:3]	STID[2:0]	EXID[17:13]	EXID[12:5]	EXID[4:0]	IDE	RTR

Everything is simple here, the filter can be configured to receive only one or two specific identifiers, they simply written to the first and second “filter registers”, also with a shift. The only important difference from the previous version is the filter mode, here it will be CAN_FILTERMODE_ IDLIST .

Filter #3 will accept two standard identifiers — 0x06D9 and 0x04C6...

```
sFilterConfig.FilterBank = 3;
sFilterConfig.FilterMode = CAN_FILTERMODE_IDLIST;
sFilterConfig.FilterIdHigh = 0x06D9 << 5;
sFilterConfig.FilterIdLow = 0x0000;
sFilterConfig.FilterMaskIdHigh = 0x04C6 << 5;
sFilterConfig.FilterMaskIdLow = 0x0000;
sFilterConfig.FilterFIFOAssignment = CAN_RX_FIFO0;

if(HAL_CAN_ConfigFilter(&hcan, &sFilterConfig) != HAL_OK)
{
    Error_Handler();
}
```

And filter No. 4 will accept two extended identifiers - 0x000006D9 and 0x000004C6 ...

```
sFilterConfig.FilterBank = 4;
sFilterConfig.FilterMode = CAN_FILTERMODE_IDLIST;
sFilterConfig.FilterIdHigh = (uint16_t)(0x000006D9 >> 13);
sFilterConfig.FilterIdLow = (uint16_t)(0x000006D9 << 3) | 0x04;
sFilterConfig.FilterMaskIdHigh = (uint16_t)(0x000004C6 >> 13);
sFilterConfig.FilterMaskIdLow = (uint16_t)(0x000004C6 << 3) | 0x04;
sFilterConfig.FilterFIFOAssignment = CAN_RX_FIFO0;

if(HAL_CAN_ConfigFilter(&hcan, &sFilterConfig) != HAL_OK)
{
    Error_Handler();
}
```

If you need to accept only one identifier, then in the second "filter register" you need to register the same in the first ...

```
sFilterConfig.FilterIdHigh = (uint16_t)(0x000006D9 >> 13);
sFilterConfig.FilterIdLow = (uint16_t)(0x000006D9 << 3) | 0x04;
sFilterConfig.FilterMaskIdHigh = (uint16_t)(0x000006D9 >> 13);
sFilterConfig.FilterMaskIdLow = (uint16_t)(0x000006D9 << 3) | 0x04;
```

It turns out that CAN will filter the same thing twice, but apparently there is no other option, since if you just zeros there, the identifier 0x0000 will be accepted.

That's all with the second option.

The third option - 16-Bit Filters - Identifier Mask

16-Bit Filters - Identifier Mask

ID	CAN_FxR1[15:8]	CAN_FxR1[7:0]
Mask	CAN_FxR1[31:24]	CAN_FxR1[23:16]
ID	CAN_FxR2[15:8]	CAN_FxR2[7:0]
Mask	CAN_FxR2[31:24]	CAN_FxR2[23:16]
Mapping	STID[10:3]	STID[2:0] RTR DE EXID[17:15]

This option, like the first one, works with a mask, but can only filter standard (*11 bit*) frames. The advantage of this option over the first one is that one filter can be configured for two ranges of identifiers. This is achieved by using the "filter registers" in their entirety, not just the higher parts.

We will accept frames in the ranges from **0x0320** to **0x0323** , and from **0x0480** to **0x0487** .

Set up filter #5...

```
sFilterConfig.FilterBank = 5;
```

Filter mode for working with a mask...

```
sFilterConfig.FilterMode = CAN_FILTERMODE_IDMASK;
```

An important difference from the previous examples is that the filter dimension is set to 16 bits...

```
sFilterConfig.FilterScale = CAN_FILTERSCALE_16BIT;
```

In the high and low parts of the first "filter register" we write the initial IDs of the first and second ranges

```
sFilterConfig.FilterIdHigh = 0x0320 << 5; // ID 1  
sFilterConfig.FilterIdLow = 0x0480 << 5; // ID 2
```

And in the upper and lower parts of the second "filter register" we write the corresponding masks ...

```
sFilterConfig.FilterMaskIdHigh = 0x07FC << 5; // Mask 1  
sFilterConfig.FilterMaskIdLow = 0x07F8 << 5; // Mask 2
```

It will be entirely...

```
sFilterConfig.FilterBank = 5;  
sFilterConfig.FilterMode = CAN_FILTERMODE_IDMASK;  
sFilterConfig.FilterScale = CAN_FILTERSCALE_16BIT;  
sFilterConfig.FilterIdHigh = 0x0320 << 5; // ID 1  
sFilterConfig.FilterIdLow = 0x0480 << 5; // ID 2  
sFilterConfig.FilterMaskIdHigh = 0x07FC << 5; // Mask 1  
sFilterConfig.FilterMaskIdLow = 0x07F8 << 5; // Mask 2  
  
sFilterConfig.FilterFIFOAssignment = CAN_RX_FIFO0;  
  
if(HAL_CAN_ConfigFilter(&hcan, &sFilterConfig) != HAL_OK)  
{  
    Error_Handler();  
}
```

We check in the program ...

```
Терминал - dima@stD: ~
dima@stD:~$ ./bits_mask_11 0x0320 0x07FC

HEX      BIN
-----
0x0320  01100100000    ID - первый регистр фильтра
0x07FC  11111111100    Mask - второй регистр фильтра

ID которые будут приняты
-----
0x0320  01100100000
0x0321  01100100001
0x0322  01100100010
0x0323  01100100011
dima@stD:~$ ./bits_mask_11 0x0480 0x07F8

HEX      BIN
-----
0x0480  10010000000    ID - первый регистр фильтра
0x07F8  11111111100    Mask - второй регистр фильтра

ID которые будут приняты
-----
0x0480  10010000000
0x0481  10010000001
0x0482  10010000010
0x0483  10010000011
0x0484  10010000100
0x0485  10010000101
0x0486  10010000110
0x0487  10010000111
dima@stD:~$
```

If you need to accept only one range, then instead of the second identifier and mask, you need to write the same as for the first ...

```
sFilterConfig.FilterIdHigh = 0x0320 << 5; // ID 1
sFilterConfig.FilterIdLow = 0x0320 << 5; // ID 2
sFilterConfig.FilterMaskIdHigh = 0x07FC << 5; // Mask 1
sFilterConfig.FilterMaskIdLow = 0x07FC << 5; // Mask 2
```

Again, it turns out that CAN will do extra work, but if you write zeros there, then all standard frames will be received in general.

I hope everything is clear here.

This option is similar to the second, only again, we can filter not two specific identifiers, but four. And as may have guessed, they can only be standard.

We will accept identifiers **0x0690** , **0x0693** , **0x0696** and **0x0699** .

Set up filter #6 and change the mode to CAN_FILTERMODE_IDLIST...

```
sFilterConfig.FilterBank = 6;
sFilterConfig.FilterMode = CAN_FILTERMODE_IDLIST;
sFilterConfig.FilterScale = CAN_FILTERSCALE_16BIT;
sFilterConfig.FilterIdHigh = 0x0690 << 5; // ID 1
sFilterConfig.FilterIdLow = 0x0693 << 5; // ID 2
sFilterConfig.FilterMaskIdHigh = 0x0696 << 5; // ID 3
sFilterConfig.FilterMaskIdLow = 0x0699 << 5; // ID 4

sFilterConfig.FilterFIFOAssignment = CAN_RX_FIFO0;

if(HAL_CAN_ConfigFilter(&hcan, &sFilterConfig) != HAL_OK)
{
    Error_Handler();
}
```

Again, if you enter 0x0000 instead of some identifier, then frame 0x0000 will be received.

With the last option, as well as with the analysis of filters, it's over.

Finally, it remains to say that the filter configured as CAN_FILTERSCALE_32BIT has a higher priority than CAN_FILTERSCALE_16BIT.

So if we set up two filters like this...

```

sFilterConfig.FilterBank = 0;
sFilterConfig.FilterMode = CAN_FILTERMODE_IDMASK;
sFilterConfig.FilterScale = CAN_FILTERSCALE_16BIT; // отличие
sFilterConfig.FilterIdHigh = 0x0000;
sFilterConfig.FilterIdLow = 0x0000;
sFilterConfig.FilterMaskIdHigh = 0x0000;
sFilterConfig.FilterMaskIdLow = 0x0000;
sFilterConfig.FilterFIFOAssignment = CAN_RX_FIFO0; // буфер 0
sFilterConfig.FilterActivation = ENABLE;

if(HAL_CAN_ConfigFilter(&hcan, &sFilterConfig) != HAL_OK)
{
    Error_Handler();
}

sFilterConfig.FilterBank = 1;
sFilterConfig.FilterMode = CAN_FILTERMODE_IDMASK;
sFilterConfig.FilterScale = CAN_FILTERSCALE_32BIT; // отличие
sFilterConfig.FilterIdHigh = 0x0000;
sFilterConfig.FilterIdLow = 0x0000;
sFilterConfig.FilterMaskIdHigh = 0x0000;
sFilterConfig.FilterMaskIdLow = 0x0000;
sFilterConfig.FilterFIFOAssignment = CAN_RX_FIFO1; // буфер 1

if(HAL_CAN_ConfigFilter(&hcan, &sFilterConfig) != HAL_OK)
{
    Error_Handler();
}

```

... all frames will be added to buffer 1.

Similarly, filter numbers take precedence. The lower the filter number, the higher its priority. In the example below, all frames will be added again to buffer 1.

```

sFilterConfig.FilterBank = 1;
sFilterConfig.FilterMode = CAN_FILTERMODE_IDMASK;
sFilterConfig.FilterScale = CAN_FILTERSCALE_32BIT;
sFilterConfig.FilterIdHigh = 0x0000;
sFilterConfig.FilterIdLow = 0x0000;
sFilterConfig.FilterMaskIdHigh = 0x0000;
sFilterConfig.FilterMaskIdLow = 0x0000;
sFilterConfig.FilterFIFOAssignment = CAN_RX_FIFO0; // буфер 0
sFilterConfig.FilterActivation = ENABLE;

if(HAL_CAN_ConfigFilter(&hcan, &sFilterConfig) != HAL_OK)
{
    Error_Handler();
}

sFilterConfig.FilterBank = 0;
sFilterConfig.FilterMode = CAN_FILTERMODE_IDMASK;
sFilterConfig.FilterScale = CAN_FILTERSCALE_32BIT;
sFilterConfig.FilterIdHigh = 0x0000;
sFilterConfig.FilterIdLow = 0x0000;
sFilterConfig.FilterMaskIdHigh = 0x0000;
sFilterConfig.FilterMaskIdLow = 0x0000;
sFilterConfig.FilterFIFOAssignment = CAN_RX_FIFO1; // буфер 1

if(HAL_CAN_ConfigFilter(&hcan, &sFilterConfig) != HAL_OK)
{
    Error_Handler();
}

```

And if you change the filter numbers, then buffer 0. That's it.

Two CANs on one stone

The only feature of CAN2 is that it cannot work when CAN1 is disabled, in all other respects the CAN2 settings are the same as for CAN1, except for the filter numbers. To make CAN2 work, just activate CAN and configure at least one filter for it.

That is, the initialization of both CANs will be like this ...

```

////////// CAN 1 //////////
static void MX_CAN1_Init(void)
{
    CAN_FilterTypeDef sFilterConfig;

    hcan1.Instance = CAN1;
    hcan1.Init.Prescaler = 4;
    hcan1.Init.Mode = CAN_MODE_NORMAL;
    hcan1.Init.SyncJumpWidth = CAN_SJW_1TQ;
    hcan1.Init.TimeSeg1 = CAN_BS1_15TQ;
    hcan1.Init.TimeSeg2 = CAN_BS2_2TQ;
    hcan1.Init.TimeTriggeredMode = DISABLE;
    hcan1.Init.AutoBusOff = ENABLE;
    hcan1.Init.AutoWakeUp = DISABLE;
    hcan1.Init.AutoRetransmission = DISABLE;
    hcan1.Init.ReceiveFifoLocked = DISABLE;
    hcan1.Init.TransmitFifoPriority = ENABLE;

    if(HAL_CAN_Init(&hcan1) != HAL_OK)
    {
        Error_Handler();
    }

    /* USER CODE BEGIN CAN1_Init 2 */

    //////////// Filter CAN 1 ////////////
    sFilterConfig.FilterBank = 0;
    sFilterConfig.FilterMode = CAN_FILTERMODE_IDMASK;
    sFilterConfig.FilterScale = CAN_FILTERSCALE_32BIT;
    sFilterConfig.FilterIdHigh = 0x0000;
    sFilterConfig.FilterIdLow = 0x0000;
    sFilterConfig.FilterMaskIdHigh = 0x0000;
    sFilterConfig.FilterMaskIdLow = 0x0000;
    sFilterConfig.FilterFIFOAssignment = CAN_RX_FIFO0;
    sFilterConfig.FilterActivation = ENABLE;

    if(HAL_CAN_ConfigFilter(&hcan1, &sFilterConfig) != HAL_OK)
    {
        Error_Handler();
    }
}

////////// CAN 2 //////////
static void MX_CAN2_Init(void)
{
    CAN_FilterTypeDef sFilterConfig;

    hcan2.Instance = CAN2;
    hcan2.Init.Prescaler = 4;
    hcan2.Init.Mode = CAN_MODE_NORMAL;
    hcan2.Init.SyncJumpWidth = CAN_SJW_1TQ;
    hcan2.Init.TimeSeg1 = CAN_BS1_15TQ;
    hcan2.Init.TimeSeg2 = CAN_BS2_2TQ;
    hcan2.Init.TimeTriggeredMode = DISABLE;

```

```

hcan2.Init.AutoRetransmission = DISABLE;
hcan2.Init.ReceiveFifoLocked = DISABLE;
hcan2.Init.TransmitFifoPriority = ENABLE;

if(HAL_CAN_Init(&hcan2) != HAL_OK)
{
    Error_Handler();
}

/* USER CODE BEGIN CAN2_Init 2 */

////////// Filter CAN 2 //////////
sFilterConfig.FilterBank = 14;
sFilterConfig.FilterMode = CAN_FILTERMODE_IDMASK;
sFilterConfig.FilterScale = CAN_FILTERSCALE_32BIT;
sFilterConfig.FilterIdHigh = 0x0000;
sFilterConfig.FilterIdLow = 0x0000;
sFilterConfig.FilterMaskIdHigh = 0x0000;
sFilterConfig.FilterMaskIdLow = 0x0000;
sFilterConfig.FilterFIFOAssignment = CAN_RX_FIFO0;
sFilterConfig.FilterActivation = ENABLE;
sFilterConfig.SlaveStartFilterBank = 14;

if(HAL_CAN_ConfigFilter(&hcan2, &sFilterConfig) != HAL_OK)
{
    Error_Handler();
}
}

```

For CAN1 filters from 0 to 13, for CAN2 filters from 14 to 27. CAN2 has an additional parameter - sFilterConfig.SlaveStartFilterBank = 14, telling the system which number the CAN2 filters start with. Filters are added in the same way as described above.

Of course, CAN's can be configured for different speeds. Actually I don't know what else to say.

Despite the fact that CAN2 is called Slave in the manuals, this is a completely independent interface, it simply cannot work without CAN1 enabled, since it is CAN1 that includes the entire system of buffers, filters, mailboxes, etc. Apparently this is some feature of the design of the stone. Probably at first they did plan to make two CANs, and then they changed their mind and adjusted 😊

if. If CAN1 is not used and a transceiver is not connected to it, then the RX leg needs to be pulled up to plus, you can use an internal pull-up.

Both CANs start up like this ...

```

HAL_CAN_Start(&hcan1);
HAL_CAN_ActivateNotification(&hcan1, CAN_IT_RX_FIFO0_MSG_PENDING | CAN_IT_ERROR | CAN_

HAL_CAN_Start(&hcan2);
HAL_CAN_ActivateNotification(&hcan2, CAN_IT_RX_FIFO0_MSG_PENDING | CAN_IT_ERROR | CAN_

```

If CAN1 is not used, then it does not need to be started.

CAN_IT_RX_FIFO0_MSG_PENDING- it can be CAN_IT_RX_FIFO 1_MSG_PENDING, depending on which buffer you specified in the filter settings (*CAN_RX_FIFO0* or *CAN_RX_FIFO1*) .

Callback for errors...

```
void HAL_CAN_ErrorCallback(CAN_HandleTypeDef *hcan)
{
    if(hcan->Instance == CAN1)
    {
        uint32_t er = HAL_CAN_GetError(&hcan1);
        sprintf(trans_str,"ER CAN_1 %lu %08lX", er, er);
        trans_to_usart1(trans_str);
    }
    else if(hcan->Instance == CAN2)
    {
        uint32_t er = HAL_CAN_GetError(&hcan2);
        sprintf(trans_str,"ER CAN_2 %lu %08lX", er, er);
        trans_to_usart1(trans_str);
    }
}
```

Callback for receiving if both CANs have buffer CAN_RX_FIFO0...

```
void HAL_CAN_RxFifo0MsgPendingCallback(CAN_HandleTypeDef *hcan)
{
    /////////////////////////////////// CAN1 ///////////////////////////////////
    if(HAL_CAN_GetRxMessage(&hcan1, CAN_RX_FIFO0, &RxHeader_1, RxData_1) == HAL_OK)
    {

    }

    /////////////////////////////////// CAN2 ///////////////////////////////////
    if(HAL_CAN_GetRxMessage(&hcan2, CAN_RX_FIFO0, &RxHeader_2, RxData_2) == HAL_OK)
    {

    }
}
```

If buffer CAN_RX_FIFO1 is used for both CANs...

```

void HAL_CAN_RxFifo1MsgPendingCallback(CAN_HandleTypeDef *hcan)
{
    /////////////////////////////////// CAN1 ///////////////////////////////////
    if(HAL_CAN_GetRxMessage(&hcan1, CAN_RX_FIFO1, &RxHeader_1, RxData_1) == HAL_OK)
    {

    }

    /////////////////////////////////// CAN2 ///////////////////////////////////
    if(HAL_CAN_GetRxMessage(&hcan2, CAN_RX_FIFO1, &RxHeader_2, RxData_2) == HAL_OK)
    {

    }

}

```

Please note that in the function name ...RxFifo 0 ... has changed to ...RxFifo 1 ...

If different buffers are configured for different CANs ...

```

void HAL_CAN_RxFifo0MsgPendingCallback(CAN_HandleTypeDef *hcan)
{
    /////////////////////////////////// CAN1 ///////////////////////////////////
    if(HAL_CAN_GetRxMessage(&hcan1, CAN_RX_FIFO0, &RxHeader_1, RxData_1) == HAL_OK)
    {

    }

}

```

```

void HAL_CAN_RxFifo1MsgPendingCallback(CAN_HandleTypeDef *hcan)
{
    /////////////////////////////////// CAN2 ///////////////////////////////////
    if(HAL_CAN_GetRxMessage(&hcan2, CAN_RX_FIFO1, &RxHeader_2, RxData_2) == HAL_OK)
    {

    }

}

```

If CAN1 is not used, then nothing needs to be written for it.

Well, shipping...

```
if(HAL_CAN_GetTxMailboxesFreeLevel(&hcan1) > 0)
{
    HAL_CAN_AddTxMessage(&hcan1, &TxHeader_1, TxData_1, &TxMailbox_1);
}
```

```
if(HAL_CAN_GetTxMailboxesFreeLevel(&hcan2) > 0)
{
    HAL_CAN_AddTxMessage(&hcan2, &TxHeader_2, TxData_2, &TxMailbox_2);
}
```