CAN bus and stm32

STM32

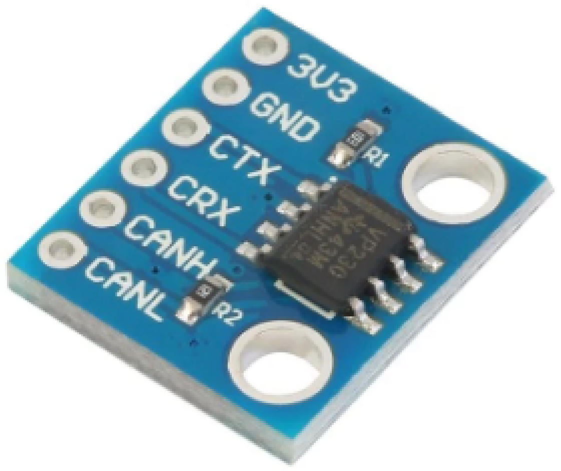
Hello.

For most readers, the CAN bus is associated with a car. However, if you are not a car enthusiast, do rush to close the article, since you can use CAN anywhere, for example, in the construction of a “sma home”, or somewhere else where reliable data transfer is needed.

The advantages of the CAN bus are that only two wires *(twisted pair)* are required , data can be transmitted over long distances *(up to several kilometers)* , resistance to strong interference, and the ability to automatically recover from failures.

All this reliability is achieved through a complex data processing mechanism, using timings, segment and other ingenious garbage, which we will consider in the course of the article.

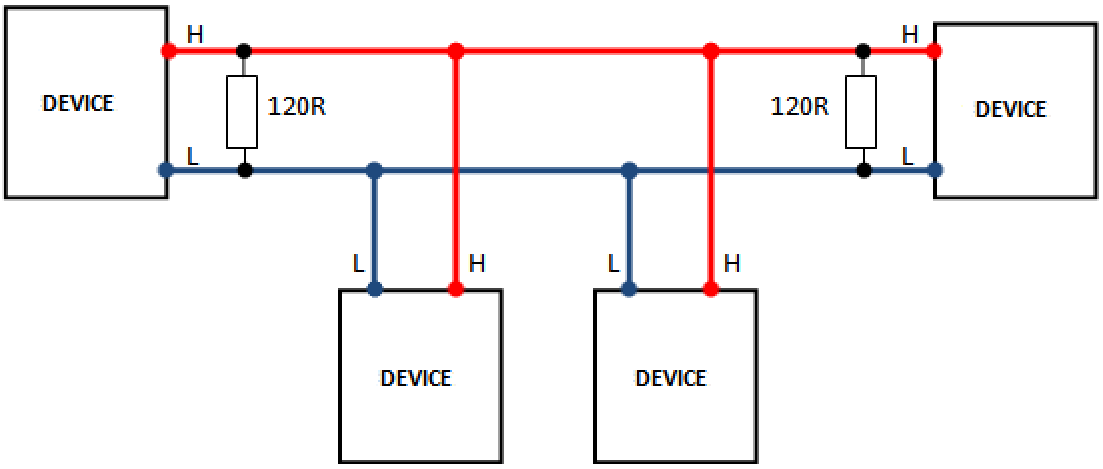
The only inconvenient moment is the need for a special CAN transceiver *(a small eight-legged mikruha)* . In order not to solder anything, you can buy these things on Ali ...



convenience when working with stm32. You can also use other popular transceivers - MCP2551 and TJA1050 (aka A1050). The TJA1050 has a small drawback - the datasheet says that it does not sup speeds below 60Kbps, although it worked for me even at lower speeds, but not always stably. Howe this is not scary, since you are unlikely to use such a low speed in your projects.

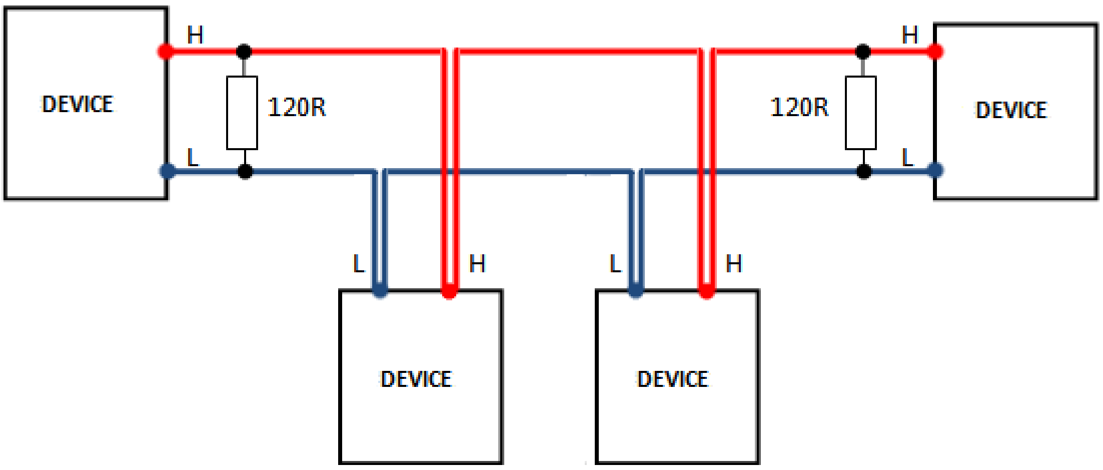
Pins C **TX** and C **RX** are connected to stm32, and CANH and CANL to twisted pair. "Earths" do not r to be connected, this is a differential signal . *A transceiver is connected to each stm'ki.*

The topology of the network is as follows ...



There can be two or more devices on the bus *(any number is possible, but in practice it is limited by load capacity of the transmitters and ranges from 100 to 200)* . Terminating resistors *(120 Ohm)* mus installed on the two extreme devices .*On the scarves, about which I wrote above, they are already there.*If there are intermediate devices on the bus, then resistors are not installed on them*(on the scarves above, they need to be soldered)*.

It is not recommended to make long branches for intermediate devices, for good there should be no branches at all, it should be like this ...



The network is similar to RS485, but in the CAN bus you do not need to programmatically select the direction of transmission, there are no master and slaves, and there are no device identifiers like I2C

Until you have purchased transceivers, you can experiment without them, for this there is a special m when one stm can send packets to itself on one interface*(and no, for this you do not need to close th legs between each other, as is sometimes done with UART, this will not work)*.

The maximum transfer rate is 1Mbps. *Now devices are being made with a higher speed (CAN FD), this does not interest us.*

The length of the line depends on the transmission speed:

1000 Kbps - 40 meters.
500 Kbps - 100 meters.
250 Kbps - 200 meters.
125 Kbps - 500 meters.
10 Kbps - 6 kilometers.

These are the most common speeds in the industry, but there are others - 800 Kbps, 400 Kbps, 83.3 Kbps, 50 Kbps, 20 Kbps. And in general, you can set any speed in your projects, though there is not much point in this.

Now let's move on to a more detailed consideration of the CAN bus.

In what follows, I will refer to the devices connected to the bus as nodes.

Data on the CAN bus is transmitted in small packets called frames *(frame)* . There are four types of frames:

Data Frame - the main frame in which various useful data is transmitted.

Remote Frame - this frame does not contain useful data, and serves to "ask" any node to send a frame. That is, some node sends this frame to the network, and another node *(which understands th is being addressed)* sends a frame with useful data. This method is suitable for polling some sensors and slightly reduces the load on the network. *How the node understands that it is being addressed is written below.* And here it is written that it is not recommended to use Remote Frame.

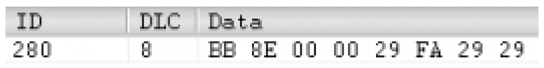
Error Frame - error frame. When one of the nodes detects a frame format error, it sends an Error Fr to the network*(has the highest priority)* . Other nodes receiving Error Frame understand that an error occurred in the network and the last message should be considered incorrect.

Overload Frame - this frame is sent by a node that is heavily overloaded and has not had time to dig incoming messages. Having sent an Overload Frame to the network, it kind of asks other nodes to w bit and resend messages. This also happens in hardware. *Now this frame is practically not used, sin CAN controllers are powerful enough to have time to process everything.*

participation.

Let's start with a detailed analysis of the Data Frame and slowly touch everything.

If we connect to the bus with a CAN sniffer working with the Can Hacker program and shoot a frame the bus, we will see the following in the program ...



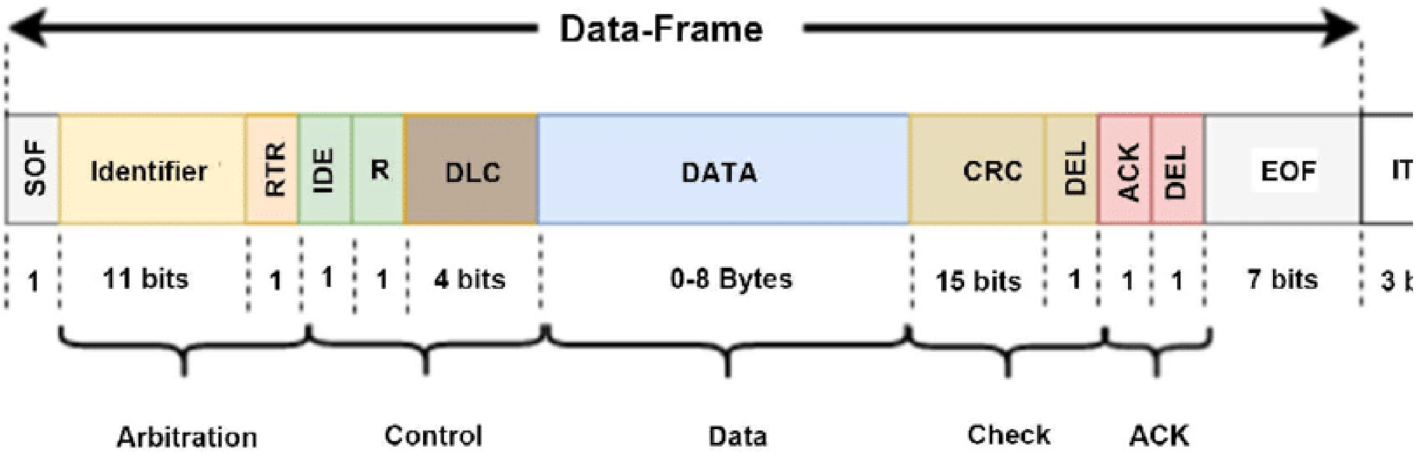
ID	DLC	Data
280	8	BB 8E 00 00 29 FA 29 29

This is how a frame looks in the CAN bus through the eyes of an average person. There is a frame identifier *(ID)* , the number of useful bytes *(DLC)* , and the useful bytes themselves *(Data)* , eight pieces. This is enough to analyze the CAN bus.

All data in CAN is written in HEX format. That is, in our case, the identifier is 0x0280.

The disadvantages of the CAN bus include a limited number of transmitted useful bytes in one frame there can be no more than eight. Less is possible.

And now let's look at the frame through the eyes of a programmer ...



Here we see:

SOF *(Start of Frame)* - the start bit informing about the beginning of the frame. *We will return to it late since it plays an important role in the operation of the entire network.*

Identifier - frame identifier. I emphasize that this is not the identifier of any network node, it is the identifier of the frame itself. The CAN bus nodes do not have any identifiers or any other addressing. When a node sends a frame, all nodes in the network receive it, and then each node loo at the identifier and decides whether it needs to process this frame or just discard. That is, when designing a network, you specify in each node which identifiers it needs to accept and process, and which ones to ignore. In this case, you do not need to check the identifiers "manually" in your program. For this, stm32 has hardware frame filtering, which, using custom filters, decides for itself w to do with a frame that has a particular identifier. That is, for example, you configured the filter so tha multiple groups. In general, a fairly flexible system. Thanks to this hardware filtering, you don't have t worry about whether your nodes will digest all the traffic, even if it is very dense.

Identifiers are of two types. Initially, when the CAN bus was designed, it was decided to make the identifier 11-bit, that is, the number of identifiers was from 0x00 to 0x07FF *(0 - 2047)* , that is, only 20 After some time, they came to the conclusion that this was too small and came up with 29 bit identifie from 0x00 to 1FFFFFFF *(0 - 536870911)* . However, for compatibility with the previous system, we decided not to increase the identifier field for the old frame, but made two types of frames. The first o remained with an 11-bit identifier and was called standard . The second received a 29-bit identifier ar was called extended . Thus, the CAN bus has two types of Data Frame, standard and extended.

extended frame

FieldIdentifiercircled in the picture with a curly bracket signed asArbitration. This indicates that the identifier also performs arbitration on the bus - it sets the frame priority. The smaller the numeric valu the frame ID, the higher its priority over other frames *(see below)* . In addition, the extended frame ha higher priority than the standard one.

RTR is one bit. If this bit is zero, thendata frame, if it is equal to one, then it is transmittedremote fran

IDE is one bit. If this bit is equal to zero, then a standard frame is transmitted, if equal to one, then ar extended frame is transmitted.

R is a bit reserved for the future.

DLC - four bits that determine the number of useful bytes *(DATA)* in the frame.

DATA - useful data.

CRC - frame checksum. Calculated by hardware based on the transmitted bits *(from SOF to the DAT field inclusive)* and the generator polynomial G *(x)* as defined in ISO 11898-1 .

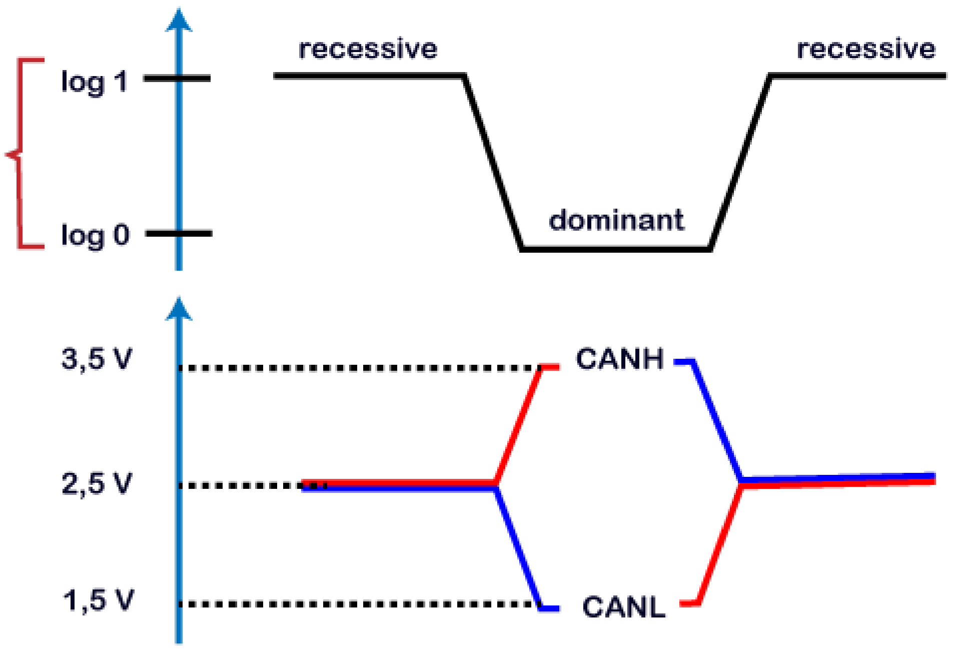
DEL is the delimiter bit.

ACK- This is the confirmation bit for the correct sending of the frame. Before sending a frame, the transmitter writes a unit to this bit ... But in order to explain what happens next, you will have to briefl describe the operation of the CAN protocol*(just below)*.

EOF *(End of Frame)* - end of frame. Seven high bits.

ITM - three high-level bits to separate the transmitted frame from the next.

CAN specification high*(logical unit)*called recessive, and low*(logical zero)*dominant. *The dominant bit takes precedence over the recessive bit. In short, zero is more important than one (below it will be cl why this is said).*



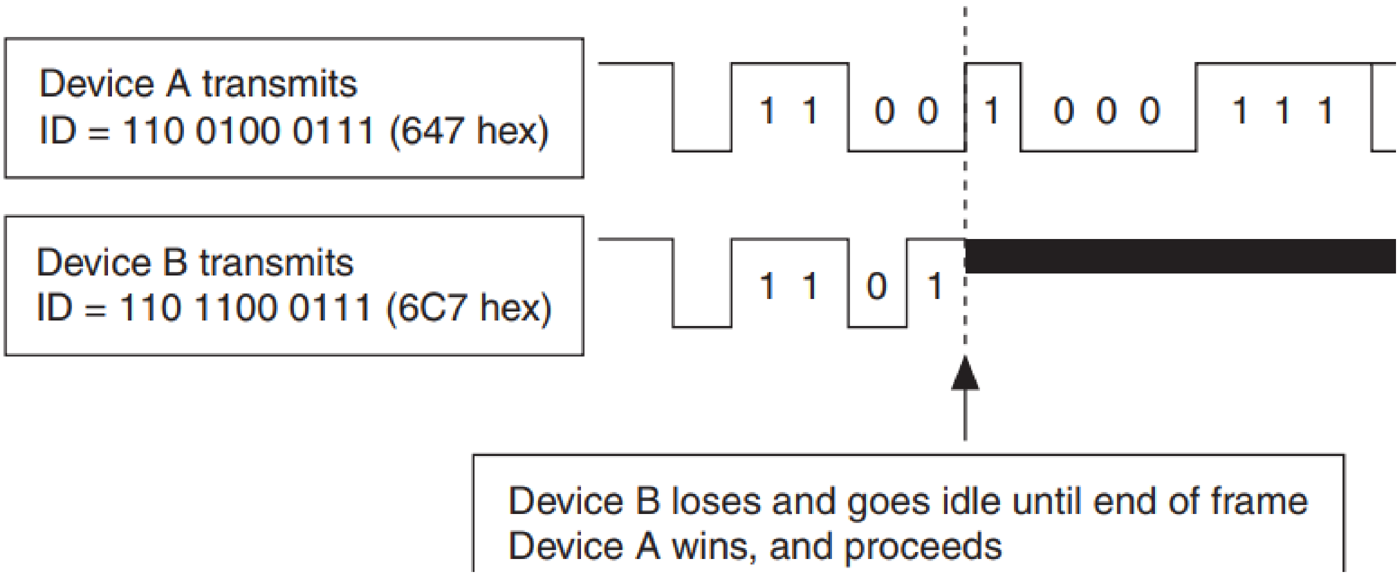
When no one transmits anything, the same voltage of 2.5 volts is set on both CAN_H and CAN_L lin
which means that the bus is free, and if the voltages are different, then the bus is busy.

additionally

When a node wants to send a frame, it "probes" the bus, and if it is free, it starts transmission. If the
is busy, then the node will not transmit anything, but will constantly “probe” the bus in anticipation of when it is free. As soon as it is free, the node will immediately begin transmission. This mechanism allows the nodes not to interfere with each other.

However, not all so simple. Since there is a very intensive data exchange on the CAN bus, it often happens that two or more nodes see that the bus is free and start transmitting at the same time. This where bitwise bus arbitration based on frame IDs comes into play.

Arbitrage works very simply. So, we have two *(or more)* nodes started simultaneously sending frame
the bus ...


Device A transmits
ID = 110 0100 0111 (647 hex)
1 1 0 0 1 0 0 0 1 1 1
Device B transmits
ID = 110 1100 0111 (6C7 hex)
1 1 0 1
Device B loses and goes idle until end of frame
Device A wins, and proceeds

The first node (Device A) transmits a frame with ID 0x0647, and the second (Device B) with ID 0x06(

Since the nodes do not know anything about each other, when transmitting each bit, they constantly monitor *("probe")* the line. If a node currently transmitting a one detects that someone is transmitting zero at the same time, then it will immediately stop transmitting *(since zero has a higher priority than one)* and will wait for the bus to be released. This is illustrated in the picture above.

At the very beginning, the line was free being in a high (recessive) state, then both nodes transmittec start bit (low state, it is also dominant), then both began to transmit identifiers. First, two ones, then o zero, and finally the first node transmitted another zero, and the second one transmitted a one and lc arbitration, since the dominant level (zero) takes precedence over the recessive level (one) . Now th second node will wait for the bus to become free and try to resend.

Thus, identifiers determine which frame will fly first, and which will wait. However, the question arises what happens if two nodes simultaneously send frames with the same identifier to the network? And answer is very simple - the CAN specification suggests that there should not be nodes in the network that send the same identifiers. Nevertheless, you can make it so that you have several nodes that se the same identifiers, but you need to organize the work so that they do not break into the network at same time.

Based on what you read about arbitration, it becomes clear that the functioning of the entire bus, its stability and proper operation, is tied to the fact that all nodes must be clearly synchronized with each other. Node synchronization occurs every time someone starts a transmission, i.e. when a node appe on the line.**SOF** - start bit. As soon as it appears, all nodes immediately begin the countdown. There also resynchronization during frame transmission, but more on that later.

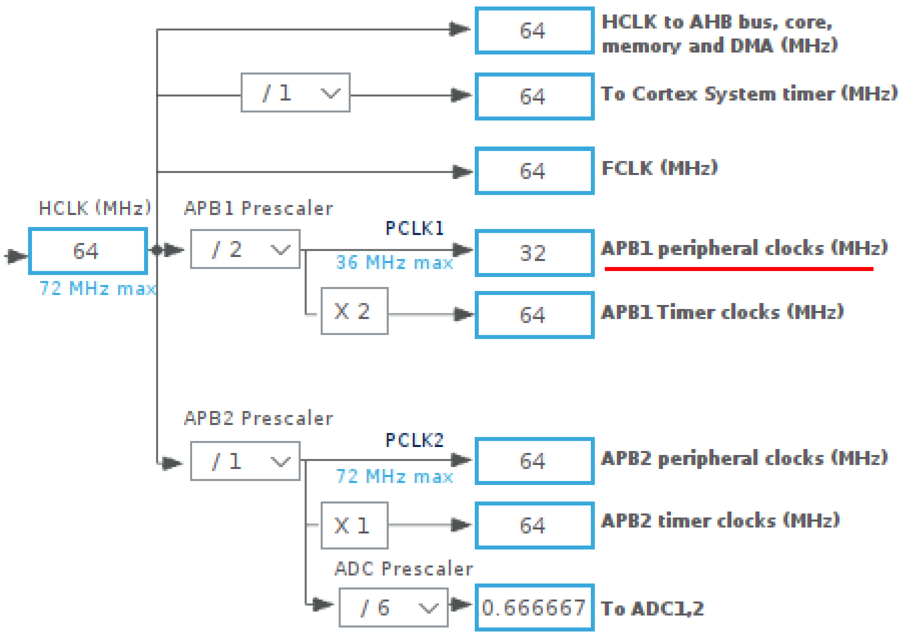
And finally, about the **ACK** bit for checking frame delivery. As mentioned above, when a node transm a frame, it writes a unit to this bit and constantly "probes" the line. When any of the nodes receives th frame, it will check it for validity *(CRC matched)*and if everything is fine, then it will change this bit fro one to zero. The transmitting node will immediately "see" that the bit has changed, and will understar that the frame was successfully delivered. At the same time, no one knows which node flipped the bi maybe the one who "needs" this frame, or maybe the one who does not pass it through the filter and discards it. Thus, the only thing the transmitting node can be sure of is that at least one of the nodes the network correctly received its frame. *In theory, if the line is working, then all nodes should have received this frame.*

If none of the nodes "answered", then, depending on the settings, the frame will be retransmitted or not *(see below "Automatic Retransmission")* .

Next, we will deal directly with the configuration and continue the study.

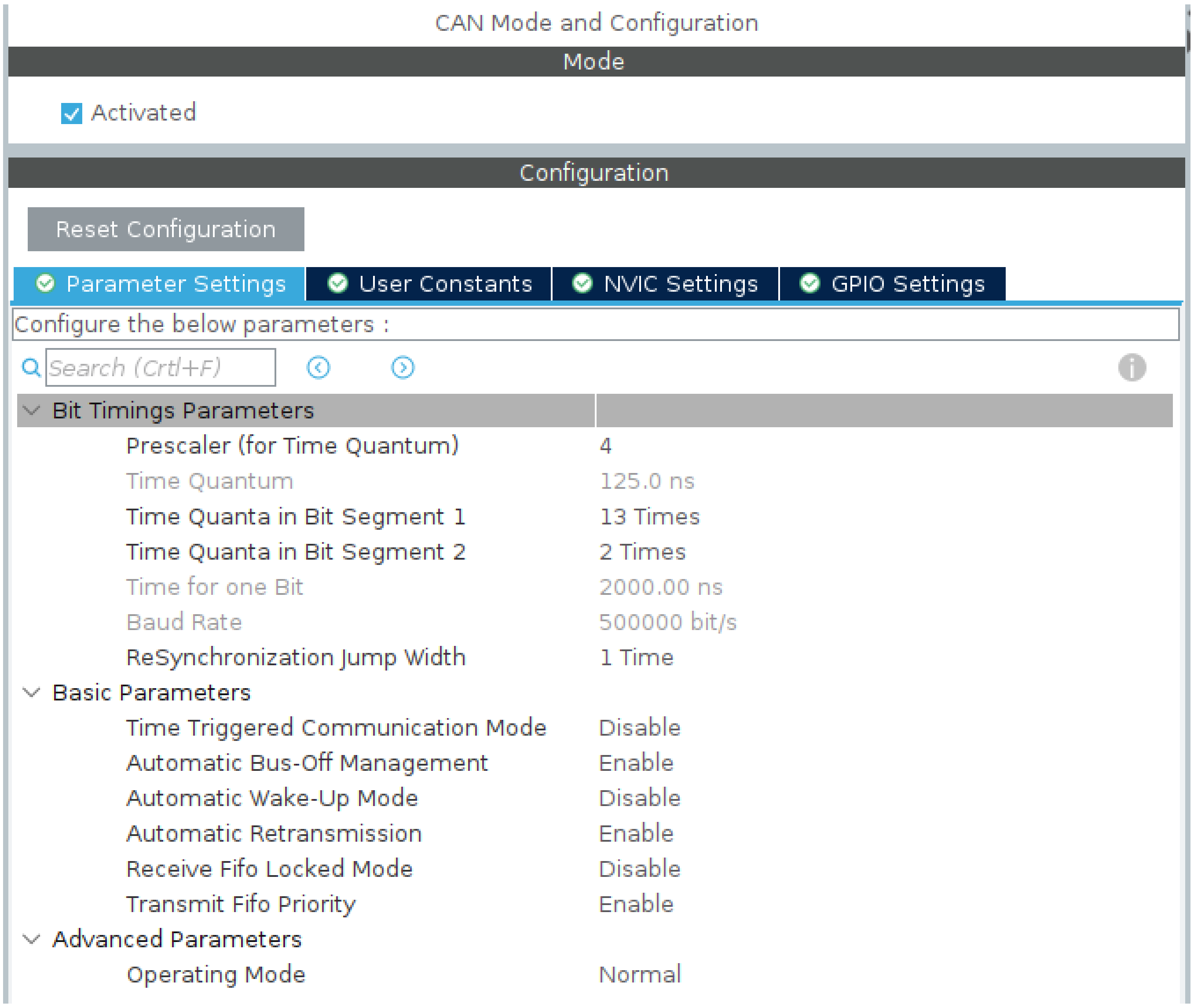
Traditionally, I will make a description for BluePill, however, it is also true for other stones. Setting up CANs on one MK will also be described.

So, first of all, you need to look in the manual on which bus you have the CAN interface hanging on. F103, F105, F205, F303, F407 and F446 have this busAPB1...



It is desirable that the frequency of this bus be either 16 MHz or 32 MHz. This is necessary for optima
tuning.

Now we activate CAN and see the various settings *(I already have it configured for 500Kb speed)* ...



CAN Mode and Configuration

Mode

✓ Activated

Configuration

Reset Configuration

Parameter Settings | User Constants | NVIC Settings | GPIO Settings

Configure the below parameters :

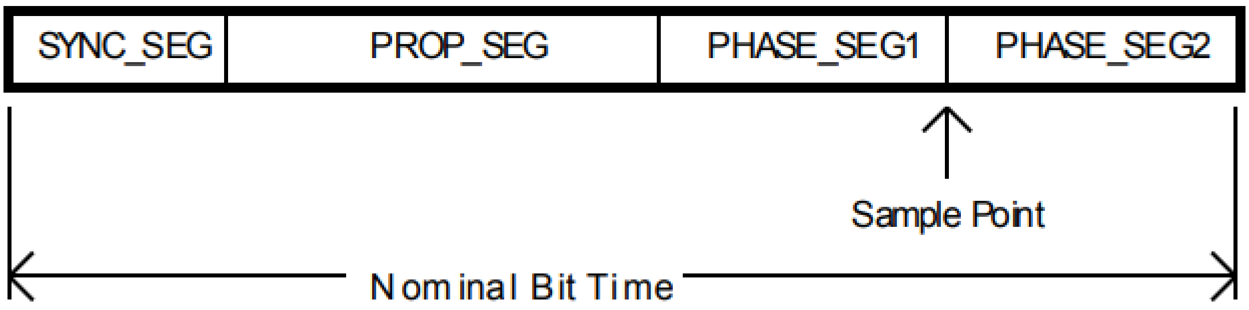
Search (Crtl+F)

Bit Timings Parameters	
Prescaler (for Time Quantum)	4
Time Quantum	125.0 ns
Time Quanta in Bit Segment 1	13 Times
Time Quanta in Bit Segment 2	2 Times
Time for one Bit	2000.00 ns
Baud Rate	500000 bit/s
ReSynchronization Jump Width	1 Time
Basic Parameters	
Time Triggered Communication Mode	Disable
Automatic Bus-Off Management	Enable
Automatic Wake-Up Mode	Disable
Automatic Retransmission	Enable
Receive Fifo Locked Mode	Disable
Transmit Fifo Priority	Enable
Advanced Parameters	
Operating Mode	Normal

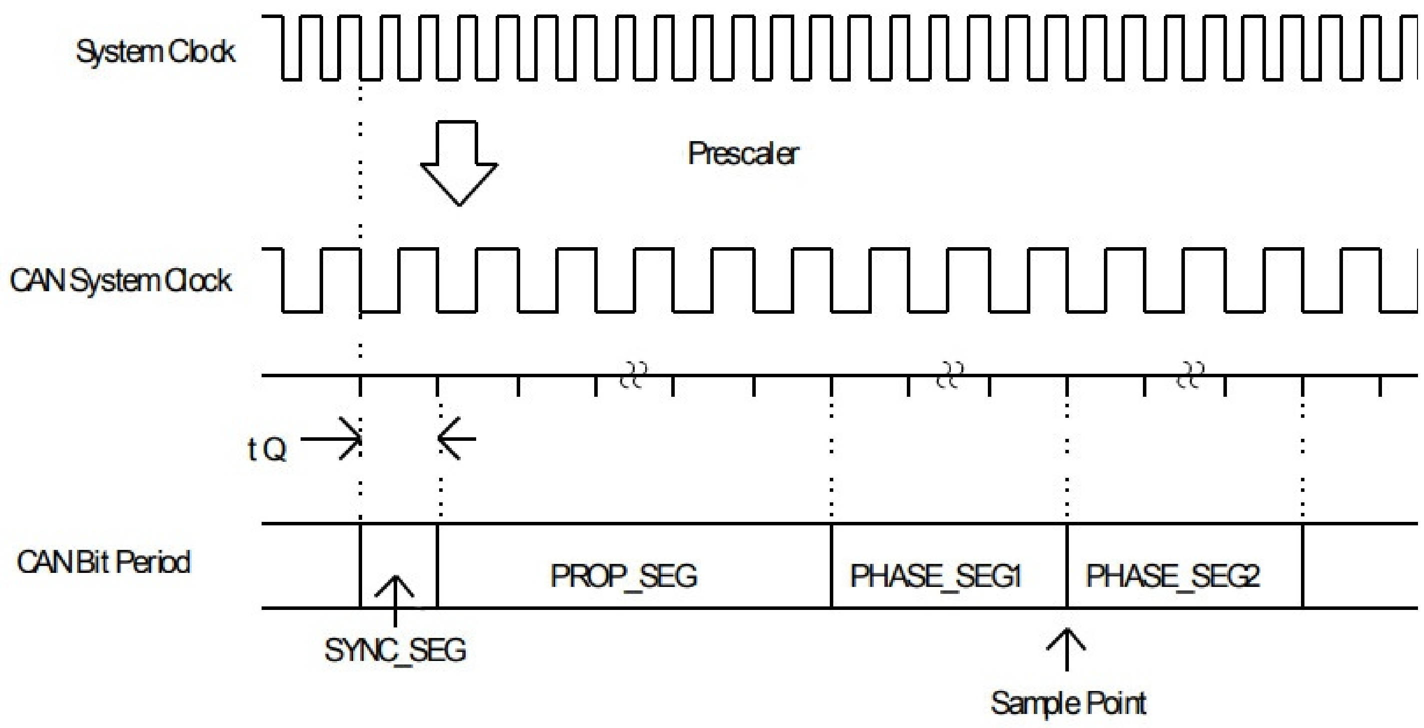
The first section (Bit Timings Parameters) is the most difficult, it is responsible for setting the timings transmission speed.

Above, I wrote that the CAN interface carefully monitors the line. So, not just logical levels are monitored, but each bit *(not a byte, but a bit) is* "decomposed" into segments that should last a certai number of time slices.*In this example, one time quantum is equal to 125.0 nanoseconds (Time Quantum).*

This is how one bit decomposed into SYNC_SEG, PROP_SEG, PHASE_SEG1 and PHASE_SEG2 segments looks like ...



Now let's figure out what we will configure and how it works ...



System Clockthis is the APB1 bus frequency. Prescaler *(Prescaler)* we divide it by some number anc the time of one quantum *(tQ)* . That is, one quantum will be equal to one "tick"CAN System Clock.

According to the CAN specification, a SYNC_SEG segment is always exactly one quantum long. The remaining three segments can last from 1 to 8 quanta. Their number must be adjusted. *The time of c quantum and the number of these quantums in one bit determines the bus speed (see below).*

You can read about the function of the segments in the manual at the link below. I won't describe it h because, firstly, it's dreary, and secondly, there is no practical sense in these explanations because i works in hardware. In a nutshell, they are needed for correction and constant resynchronization of th bus during frame transmission, since due to the remoteness of the nodes from each other and the imperfection of the quartz resonators installed on them, there is a time discrepancy, and the bus mus work clearly synchronously.

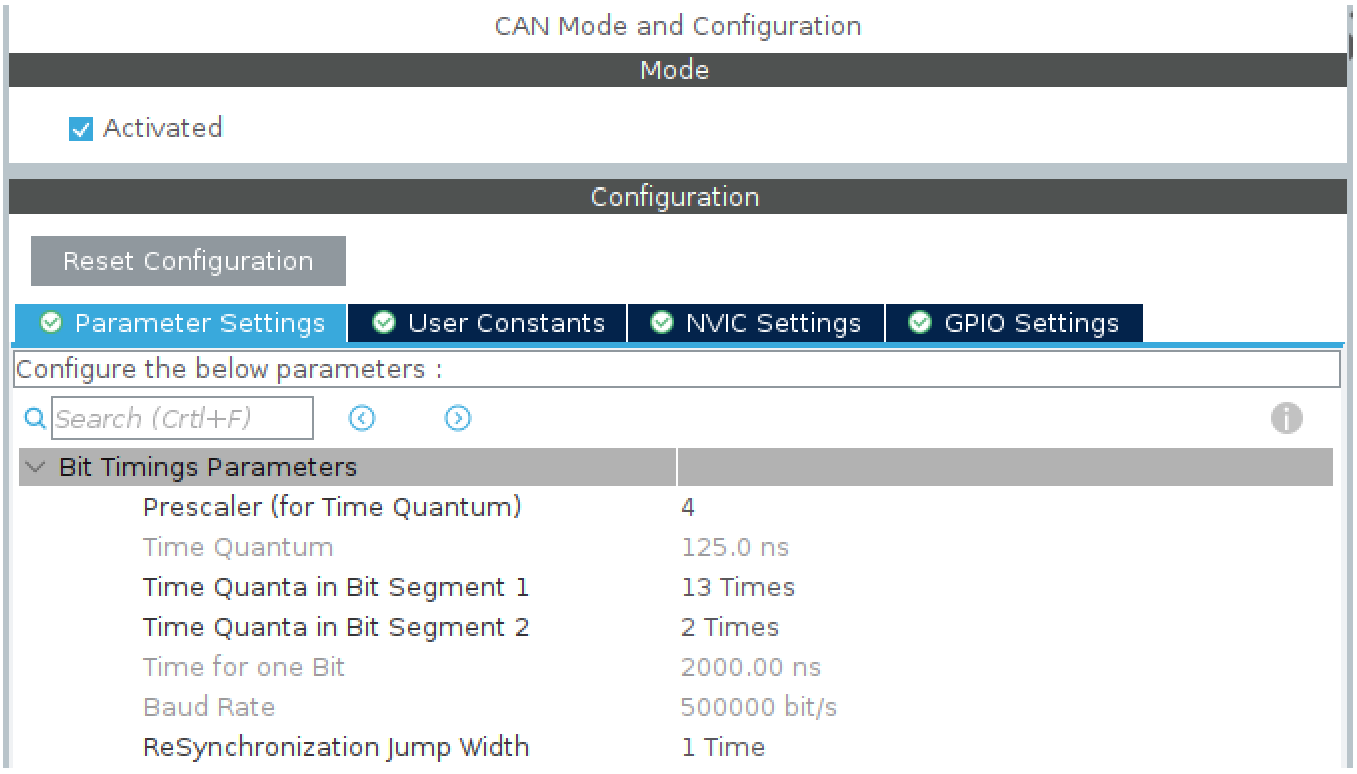
sample point- this is the point at which the "capture" of the bit occurs.

The number of quanta in the PROP_SEG, PHASE_SEG1 and PHASE_SEG2 segments is adjusted that the Sample Point is around 87.5% of the bit duration *(see below)* .

All these segments work in hardware.

This is a very superficial description, made so that you understand what the settings in Cube are responsible for. You can read all the details and nuances in this manual .

So, let's return to the Cube and now it's more meaningful to see what I have configured there ...


CAN Mode and Configuration

Mode

✓ Activated

Configuration

Reset Configuration

Parameter Settings | User Constants | NVIC Settings | GPIO Settings

Configure the below parameters :

Search (Crtl+F)

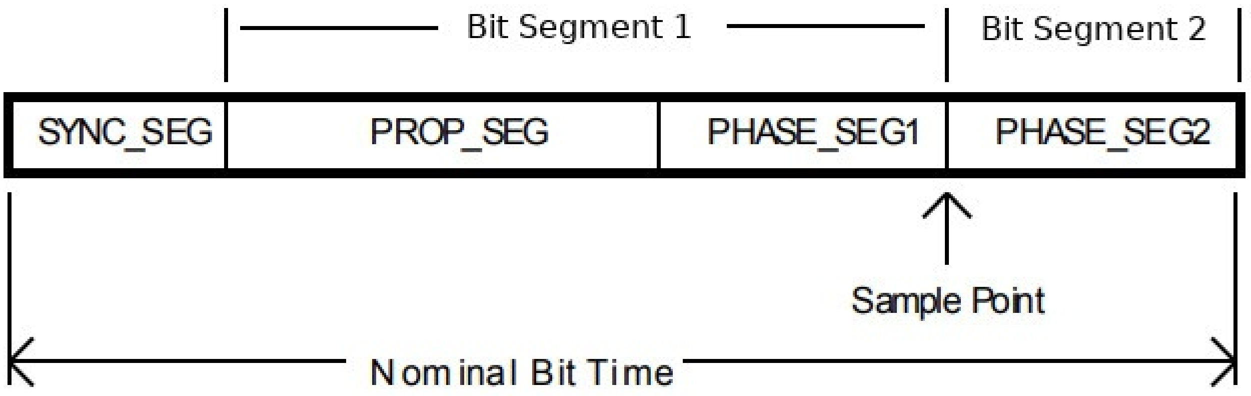
Bit Timings Parameters

Prescaler (for Time Quantum)	4
Time Quantum	125.0 ns
Time Quanta in Bit Segment 1	13 Times
Time Quanta in Bit Segment 2	2 Times
Time for one Bit	2000.00 ns
Baud Rate	500000 bit/s
ReSynchronization Jump Width	1 Time

SpecifyPrescaler (for Time Quantum)4 and get the time of one quantumTime Quantum125 ns. *We ha APB1 frequency 32MHz, divide by 4, we get 8MHz, 1 / 8000000 = 0.000000125.*

Time Quanta in Bit Segment 1is the number of quanta that make up the two segments PROP_SEG a PHASE_SEG1 *(they are combined)* .

Time Quanta in Bit Segment 2is the number of quanta that make up the last segment *(PHASE_SEG*

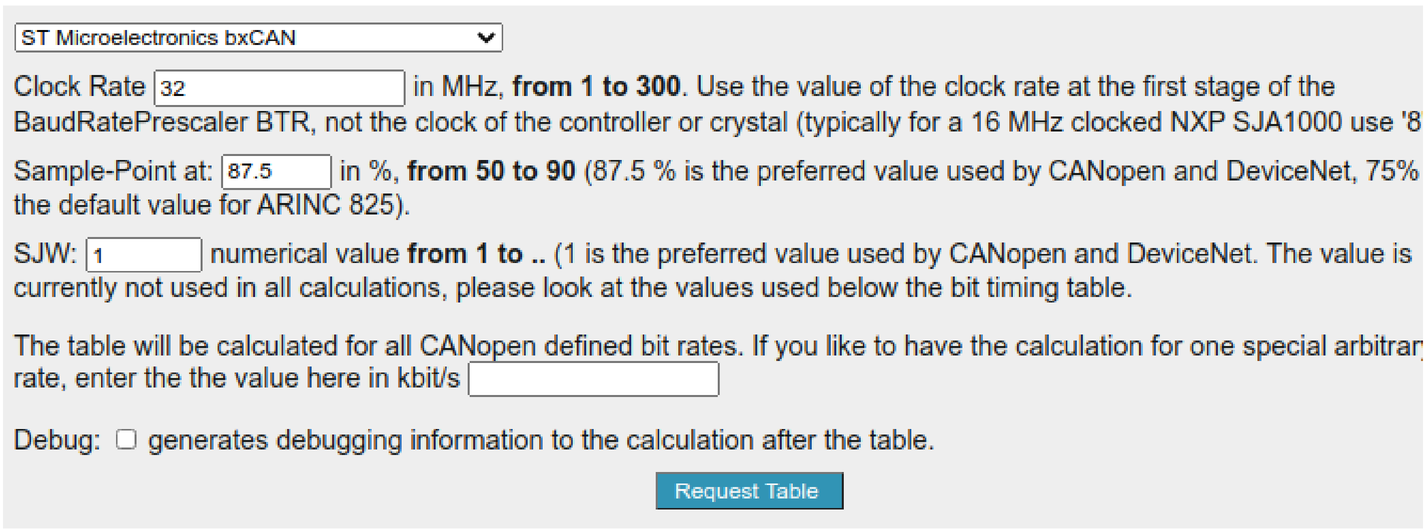


As a result, it turns out that one bit we have consists of 16 quanta - SYNC_SEG *(one quantum)* + Bit Segment 1 + Bit Segment 2, with a total duration of 2000 ns *(Time for one Bit)* . Accordingly, the data transfer rate *(Baud Rate)* turned out to be 500 Kbps.

ReSynchronization Jump Widthis a value from 1 to 4 quants, by which the duration of the PHASE_SI and PHASE_SEG2 segments can be increased or decreased by hardware for more accurate correction. The purpose of the correction is to place the "capture" point of the bit at a more favorable moment and resynchronize the nodes on the bus. In short, as far as I understand, if the line is very lo or there is a lot of noise around it, which may cause unstable operation, then this value can be increa

Well, at the end, let's figure out where the numbers 13 *(Bit Segment 1)* and 2 *(Bit Segment 2) came*
.

Open the online calculator ...



ST Microelectronics bxCAN

Clock Rate 32 in MHz, **from 1 to 300**. Use the value of the clock rate at the first stage of the BaudRatePrescaler BTR, not the clock of the controller or crystal (typically for a 16 MHz clocked NXP SJA1000 use '8

Sample-Point at: 87.5 in %, **from 50 to 90** (87.5 % is the preferred value used by CANopen and DeviceNet, 75% the default value for ARINC 825).

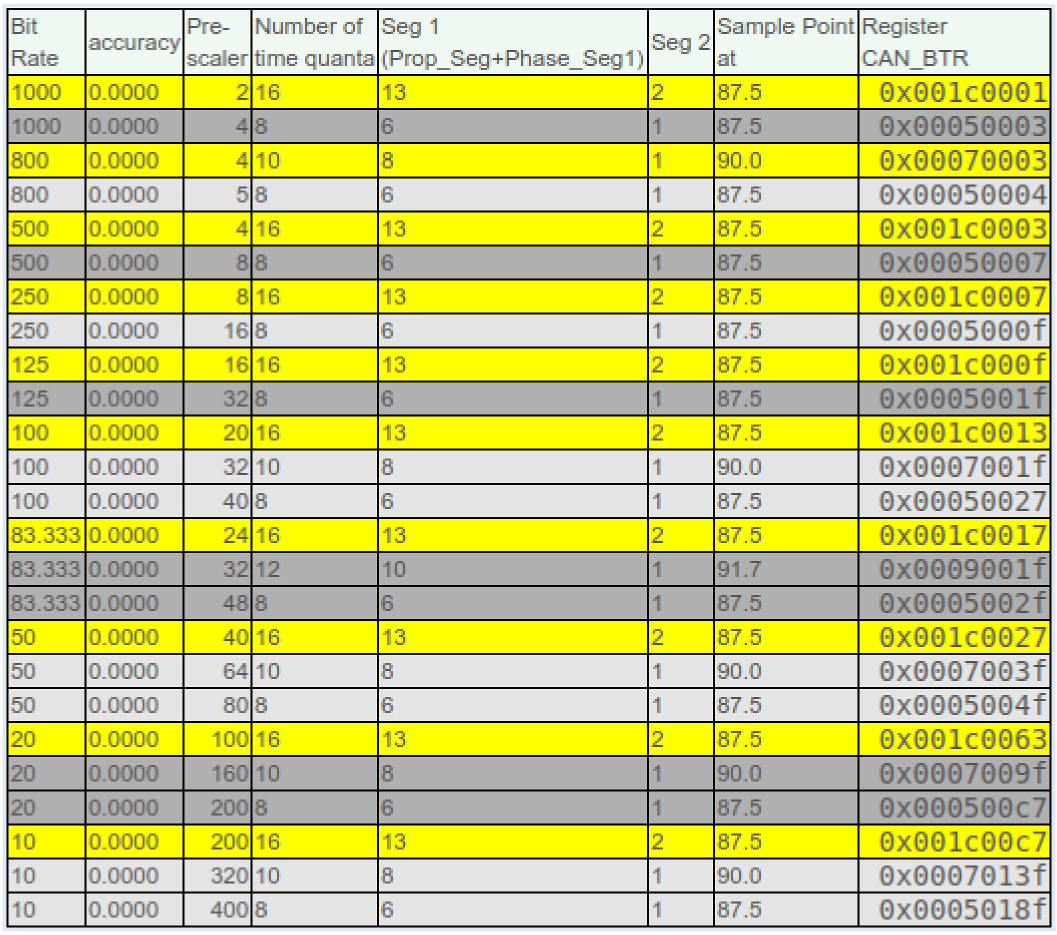
SJW: 1 numerical value **from 1 to ..** (1 is the preferred value used by CANopen and DeviceNet. The value is currently not used in all calculations, please look at the values used below the bit timing table.

The table will be calculated for all CANopen defined bit rates. If you like to have the calculation for one special arbitrar rate, enter the the value here in kbit/s

Debug: ☐ generates debugging information to the calculation after the table.

In the drop-down list, selectST Microelectronics bxCANand inclock ratespecify the frequency of the APB1 bus, in our case it is 32.

As you can see here it is written that 87.5% is the preferred value for capturing a bit *(I mentioned this above)* . We don't need anything else. Now press the buttonRequest Tableand get this picture...



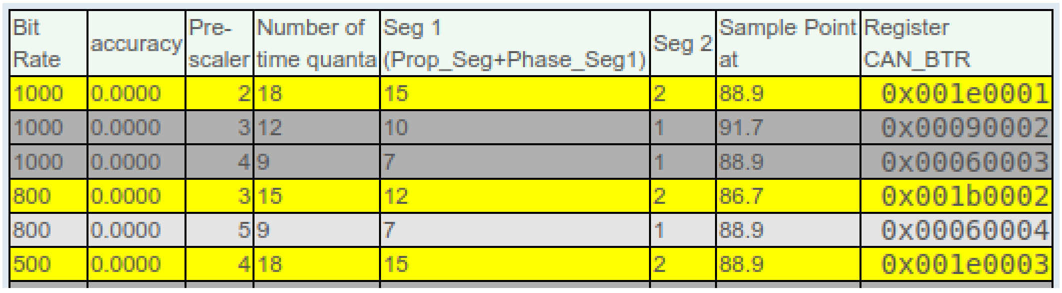
Bit Rate	accuracy	Pre-scaler	Number of time quanta	Seg 1 (Prop_Seg+Phase_Seg1)	Seg 2	Sample Point at	Register CAN_BTR
1000	0.0000	2	16	13	2	87.5	0x001c0001
1000	0.0000	4	8	6	1	87.5	0x00050003
800	0.0000	4	10	8	1	90.0	0x00070003
800	0.0000	5	8	6	1	87.5	0x00050004
500	0.0000	4	16	13	2	87.5	0x001c0003
500	0.0000	8	8	6	1	87.5	0x00050007
250	0.0000	8	16	13	2	87.5	0x001c0007
250	0.0000	16	8	6	1	87.5	0x0005000f
125	0.0000	16	16	13	2	87.5	0x001c000f
125	0.0000	32	8	6	1	87.5	0x0005001f
100	0.0000	20	16	13	2	87.5	0x001c0013
100	0.0000	32	10	8	1	90.0	0x0007001f
100	0.0000	40	8	6	1	87.5	0x00050027
83.333	0.0000	24	16	13	2	87.5	0x001c0017
83.333	0.0000	32	12	10	1	91.7	0x0009001f
83.333	0.0000	48	8	6	1	87.5	0x0005002f
50	0.0000	40	16	13	2	87.5	0x001c0027
50	0.0000	64	10	8	1	90.0	0x0007003f
50	0.0000	80	8	6	1	87.5	0x0005004f
20	0.0000	100	16	13	2	87.5	0x001c0063
20	0.0000	160	10	8	1	90.0	0x0007009f
20	0.0000	200	8	6	1	87.5	0x000500c7
10	0.0000	200	16	13	2	87.5	0x001c00c7
10	0.0000	320	10	8	1	90.0	0x0007013f
10	0.0000	400	8	6	1	87.5	0x0005018f

The first column shows the speeds you can adjust. We look at the values \u200b\u200bfor 500

Point turned out to be 87.5%.

If, for example, you want to set the speed to 250 Kbps, then it is enough to change only the prescale and if it is 800 Kbps, then only the number of quants for segments. I think everything is clear, and as promised, everything is simple. *You can*

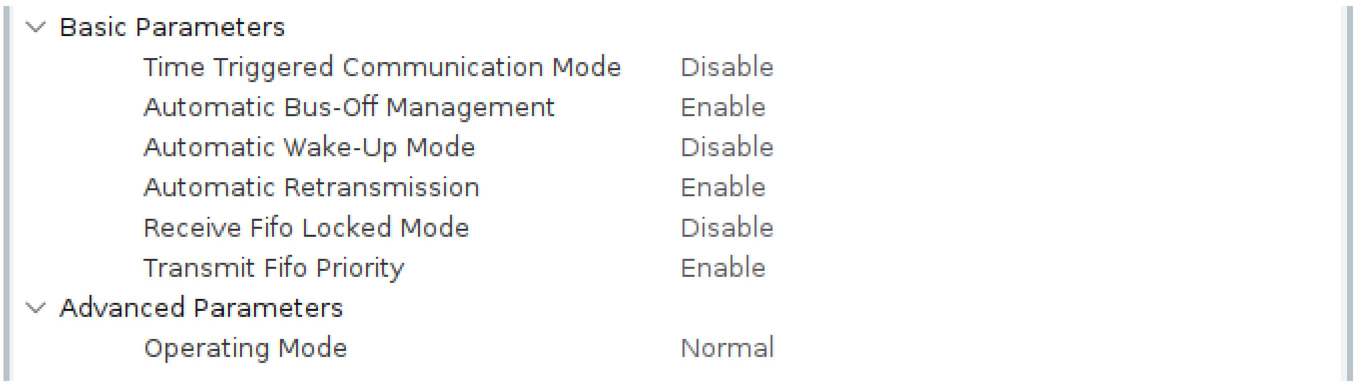
Why I wrote that it is desirable that the frequency of APB1 was 32 or 16 MHz. If you specify a freque of, for example, 36 MHz, then we will get the following result ...



Bit Rate	accuracy	Pre-scaler	Number of time quanta	Seg 1 (Prop_Seg+Phase_Seg1)	Seg 2	Sample Point at	Register CAN_BTR
1000	0.0000	2	18	15	2	88.9	0x001e0001
1000	0.0000	3	12	10	1	91.7	0x00090002
1000	0.0000	4	9	7	1	88.9	0x00060003
800	0.0000	3	15	12	2	86.7	0x001b0002
800	0.0000	5	9	7	1	88.9	0x00060004
500	0.0000	4	18	15	2	88.9	0x001e0003

It can be seen that the Sample Point "floats" a little, but this is not critical, you can work.

Next, we need to deal with the following block of settings −Basic Parameters.



Basic Parameters

Time Triggered Communication Mode	Disable
Automatic Bus-Off Management	Enable
Automatic Wake-Up Mode	Disable
Automatic Retransmission	Enable
Receive Fifo Locked Mode	Disable
Transmit Fifo Priority	Enable

Advanced Parameters

Operating Mode	Normal

Time Triggered Communication Mode- in this document , describing the operation of this item, it is written that this is a difficult-to-understand *(just as it is written)* mechanism for synchronizing nodes. I essence is approximately as follows: in normal mode, we synchronize by the start bit, and if you enal this item, then the node turns into a Time Master and at a certain interval *(there is an internal counter)* starts sending messages *(frames)* to the network , according to which other nodes are synchronized . Plus, it seems like a time frame is set for the transmission of messages by nodes. I ca say for sure because I didn't go deep into this issue myself. It is also not clear to me whether this is supported by any CAN networks and devices. In general, I do not include it.

Here is a couple more documents on this topic, which one will "go in" better)))

www.can-cia.org/fileadmin/resources/documents/proceedings/2006_fredriksson.pdf

www.cs.put.poznan.pl/wswitala/download/.pdf/CiA2000Paper_1.pdf

Yes, I will refer to frames as messages in what follows, and vice versa.

Automatic Bus-Off Management is an important and useful item. The CAN module has two error counters -Transmit Error Counter *(transmission error counter)* andReceive Error Counter *(reception e counter)* . If a receive error occurs, the Receive Error Counter is incremented by one; if a transmissio error occurs, the Transmit Error Counter is incremented immediately by 8*(apparently this is due to th fact that sending is considered a more important action)*. Upon successful reception or transmission, corresponding counter is decremented by one. If, due to a network failure, any of the counters reach the value 255, the CAN module will switch to the Bus-Off state - it does not transmit or send anythinc the bus.

If Automatic Bus-Off Management is enabled, then the CAN module will automatically recover. Reco consists in the fact that the module waits for “silence” *(recessive state)* on the bus for a period of time equal to the transmission time of 11 bits 128 times in a row, and if everything is OK, then it starts working.

*At a rate of 500Kbit/s, one bit is transmitted in 2µs. This means that in order for the bus to recover, th line must be in a recessive state for 2.8 ms (128 * 11 * 2µs = 2816µs).*

If Automatic Bus-Off Management is disabled, then you need to “manually” catch the moment the no transitions to the Bus-Off state and re-initialize it. In general, it is desirable to include this item.

Automatic Wake-Up Mode - everything is clear here, if enabled, then activity on the bus will wake u sleeping node without additional software *(a new word was invented)* 😎

Automatic Retransmission - if this item is enabled, then the node will repeat attempts to send a message if it does not receive an acknowledgment*(remember, this is the inverted ACK bit)*.

If the item is disabled, then the node will simply pull the message, and if it does not receive an acknowledgment, it will not try to resend this message.

Regardless of whether it is enabledautomatic retransmissionor disabled, if the transmitter does not receive an acknowledgment, a HAL_CAN_ERROR_ACK error is thrown *(did not receive an acknowledgment)* .

If the mode is usedTime Triggered Communication *(see above)* , then paragraphautomatic retransmissionshould be disabled.

Here it is worth saying a few more words: imagine a simple scheme of two nodes - one transmits sev different frames, the other receives them. If we break any one bus wire *(between transceivers)* , then transmitter will continue to transmit, and the receiver will receive. That is, the signal will go on <u>one</u> wire. Such is the super stability of the CAN bus. However, the caveat is that the transmitter w constantly throw the HAL_CAN_ERROR_ACK error. This I mean that if you damage one wire, and th system continues to work, then do not be surprised. At the same time, ifautomatic retransmissionis turned on, then the transmitter will hammer with the same frame, trying to repeat the sending.

I carried out this experiment on a line about half a meter long. I don't know how things will be on long distances.

Here is an interesting, but incredible, situation with Bus-Off and Automatic Retransmission .

Receive Fifo Locked Mode - the receiver has two independent buffers *(RX_FIFO_0 and RX_FIFO_* you can use one buffer or both. *Which messages will go to the zero or the first buffer depends on the filter settings (see below).* Each of the buffers is divided into three cells called mailboxes *(see the figu below)*. Each mailbox can store one message.

If this mode is disabled, then if all boxes are full and no messages are being read, the last message be overwritten by the new one.

If this mode is enabled, then if all boxes are full and no messages are being read, new incoming messages will be discarded. That is, old messages will remain in the boxes.

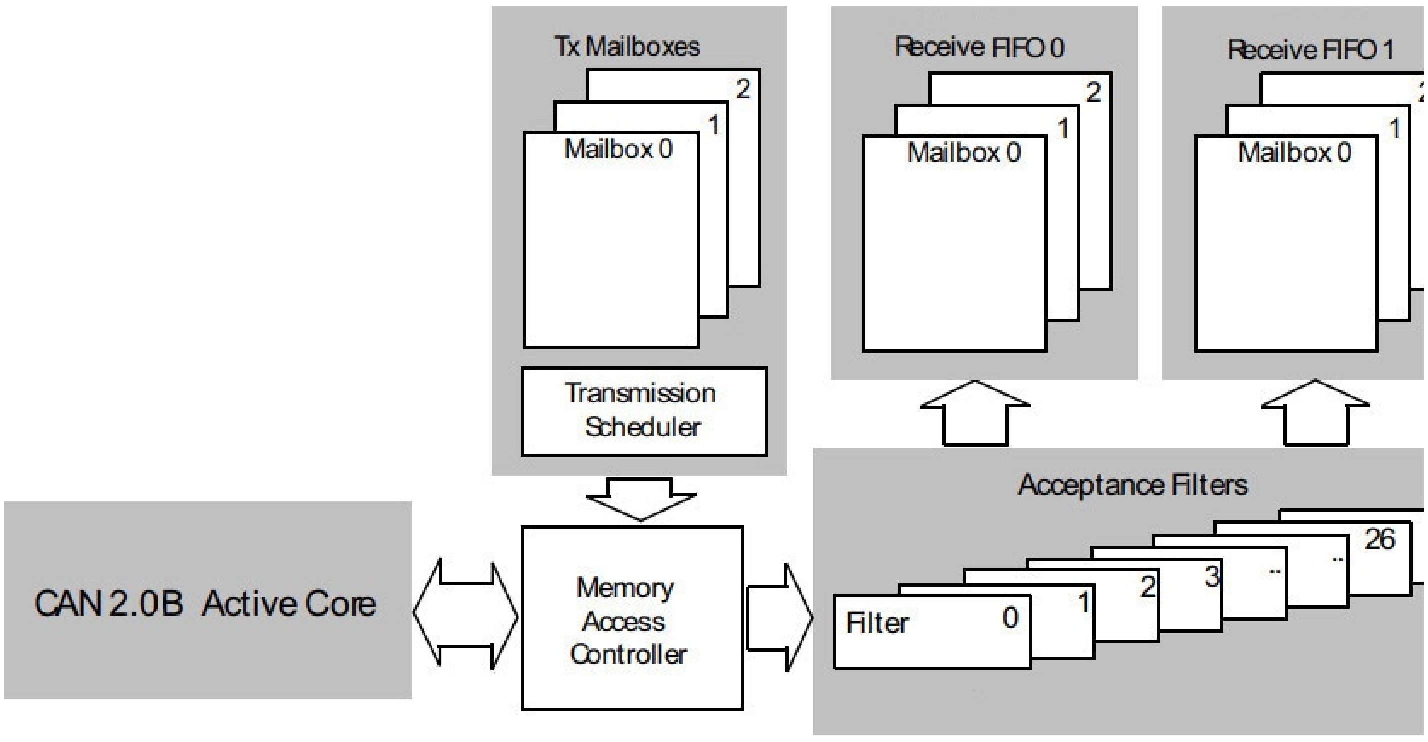
Enable or disable this mode, you decide, based on what is more important.

Transmit Fifo Priority - the transmitter also has a buffer *(one)* divided into three mailboxes. The sen messages are placed in these mailboxes, and from there they fly to the network. If you send messag too often, they will accumulate in these boxes.

If the mode is enabled, then messages leave the boxes in chronological order, that is, according to th FIFO principle - first in, first out.

If the mode is disabled, then messages with a higher priority will fly first *(as you remember, the priori set by the identifier)* . Thus, with a very intensive sending of messages, it may turn out that a messag with a low priority will never be sent, and it will lie in its box all the time.

Mailbox layout...

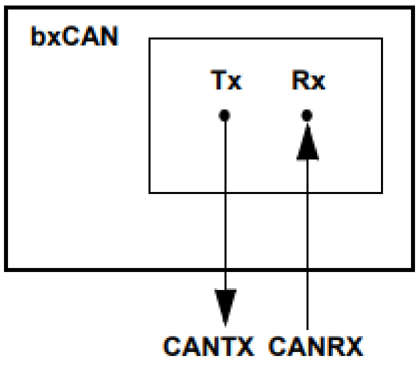


Above we see three mailboxes for sending, which are controlled by the Transmission Scheduler bloc depending on the settings of these same filters, allowing or preventing them from getting into incomir mailboxes.

All work with mailboxes *(except for reading incoming messages and sending)* occurs at the hardware level.

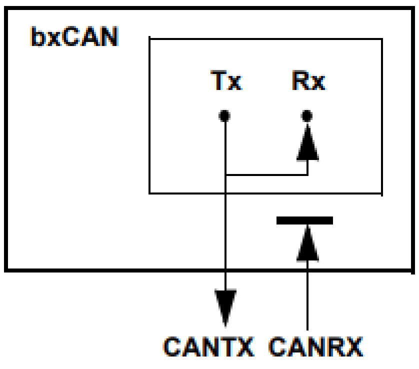
And finally, the last setting item -operating mode- operating mode or connection method.

Normal …



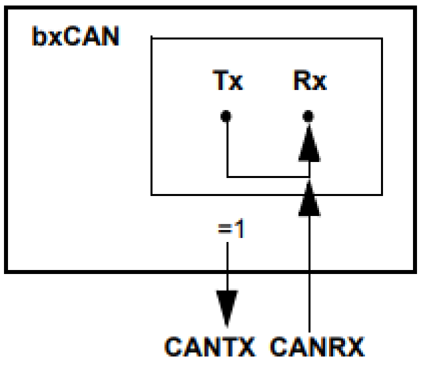
A transceiver is connected to the board and it works as a full-fledged node in a network of two or mor devices. That is a normal job.

Loopback …



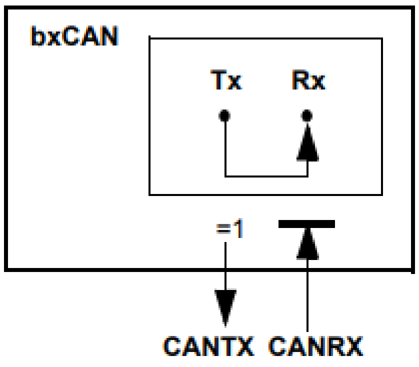
The transceiver board will transmit data to the bus and listen to itself at the same time. No data will b received from the bus.

Silent …



The board with the transceiver receives data from the bus, but does not transmit anything itself. This option is suitable when you need to read the CAN bus in the car, and at the same time do not accidentally send anything there.

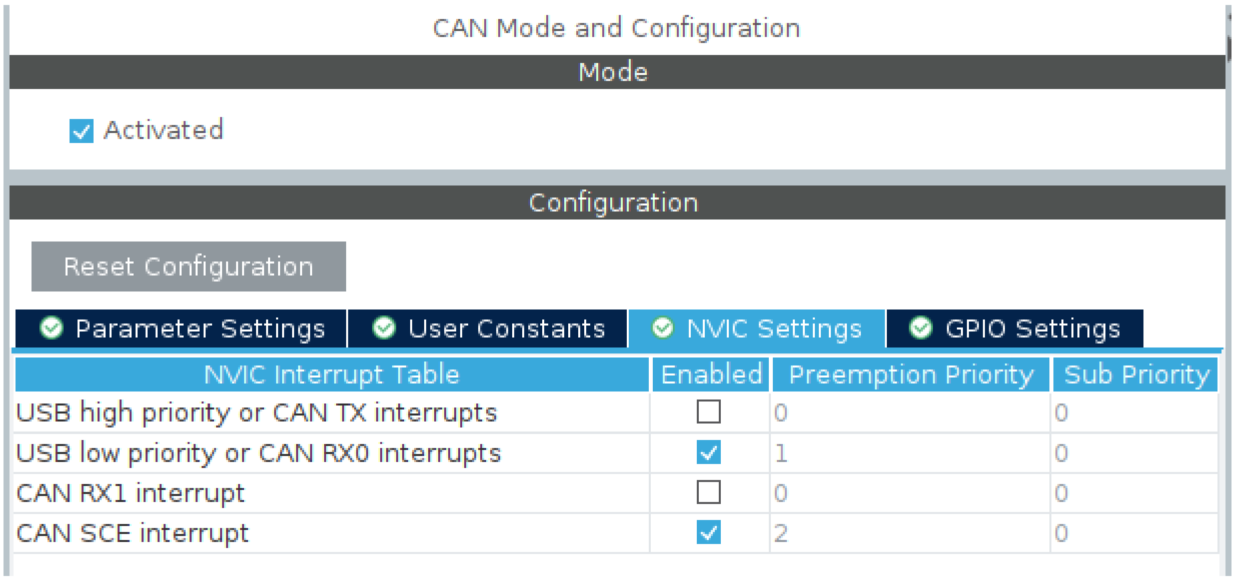
Loopback combined with Silent …



Board without a transceiver. All data is spinning inside the MK. This mode is suitable for testing the program in the absence of a transceiver and another node.

Now it's time to program.

Since, for sure, not all readers have transceivers at hand, but they want to try, we will first write a program for the modeLoopback combined with Silent. Specify it in the settings, do the rest as in the pictures and enable interrupts ...


CAN Mode and Configuration

Mode

✓ Activated

Configuration

Reset Configuration

Parameter Settings | User Constants | NVIC Settings | GPIO Settings

NVIC Interrupt Table	Enabled	Preemption Priority	Sub Priority
USB high priority or CAN TX interrupts	☐	0	0
USB low priority or CAN RX0 interrupts	✓	1	0
CAN RX1 interrupt	☐	0	0
CAN SCE interrupt	✓	2	0

Interrupts for the RX_FIFO_0 *(CAN RX0)* buffer will be called when a frame arrives and, after passin
through the filters, will be placed in one of the three mailboxes. In which box it does not matter for us
is important to read it right away, right in the interrupt. We will not use the RX_FIFO_1 buffer, one is
enough, we will return to this issue later.

Please note that for F103 *(and some other gems)* this interrupt overlaps with the USB interrupt. If yo
plan to use USB, then enable the interrupt for the RX_FIFO_1 buffer, you will not use the RX_FIFO_
buffer. It won't interfere with the USB.

The CAN SCE interrupt is called when an error occurs.

The CAN TX interrupt is called when the frame is sent - we are not very interested in it, so we do not
it on.

We declare globally two structures, two arrays and a variable ...

```
/* USER CODE BEGIN PV */
CAN_TxHeaderTypeDef TxHeader;
CAN_RxHeaderTypeDef RxHeader;
uint8_t TxData[8] = {0,};
uint8_t RxData[8] = {0,};
uint32_t TxMailbox = 0;
```

The TxHeader structure is responsible for sending frames, we will fill it in below.
The RxHeader structure for the received frame, we will read from it when receiving.
We will enter the useful data that we want to transfer into the TxData array.
The RxData array will contain useful data from the received frame.
You don't need to do anything with the TxMailbox variable.

We add a callback for receiving data *(for the RX_FIFO_0 buffer)* ...

```
/* USER CODE BEGIN 0 */
void HAL_CAN_RxFifo0MsgPendingCallback(CAN_HandleTypeDef *hcan)
{
   if(HAL_CAN_GetRxMessage(hcan, CAN_RX_FIFO0, &RxHeader, RxData) == HAL_OK)
   {
      HAL_GPIO_TogglePin(GPIOC, GPIO_PIN_13);
   }
}
```

When receiving any frame, we immediately pick it up from the mailbox using the functionHAL_CAN_GetRxMessage(...)and flash a light.

If you have an interrupt configured for the RX_FIFO_1 buffer, then the callback is…

```
void HAL_CAN_RxFifo1MsgPendingCallback(CAN_HandleTypeDef *hcan)
{
   if(HAL_CAN_GetRxMessage(hcan, CAN_RX_FIFO1, &RxHeader, RxData) == HAL_OK)
   {
      HAL_GPIO_TogglePin(GPIOC, GPIO_PIN_13);
   }
}
```

*void HAL_CAN_RxFifo **1** ... and CAN_RX_FIFO1.*

And a callback for errors...

```
void HAL_CAN_ErrorCallback(CAN_HandleTypeDef *hcan)
{
   uint32_t er = HAL_CAN_GetError(hcan);
   sprintf(trans_str,"ER CAN %lu %08lX", er, er);
   HAL_UART_Transmit(&huart1, (uint8_t*)trans_str, strlen(trans_str), 100);
}
```

Before an infinite loop, we fill in the structure responsible for sending frames ...

```
/* USER CODE BEGIN 2 */
TxHeader.StdId = 0x0378;
TxHeader.ExtId = 0;
TxHeader.RTR = CAN_RTR_DATA; //CAN_RTR_REMOTE
TxHeader.IDE = CAN_ID_STD;   // CAN_ID_EXT
TxHeader.DLC = 8;
TxHeader.TransmitGlobalTime = 0;
```

StdIdis the ID of the standard frame.

ExtIdis the extended frame identifier. We will send the standard one, so we write 0 here.

RTR = CAN_RTR_DATA- this indicates that we are sending a data frame *(Data Frame)* . *If you specifyCAN_RTR_REMOTE, then it will be Remote Frame.*

IDE = CAN_ID_STD- this indicates that we are sending a standard frame. *If you specifyCAN_ID_EX then it will be an expanded frame. ATStdIdmust be set to 0, andExtIdwrite extended identifier.*

DLC=8- the number of useful bytes transmitted in the frame *(from 1 to 8)* .

TransmitGlobalTime- refers to Time Triggered Communication Mode, we don't use it, so we write 0. \

fill the array to send useful data with some kind of rubbish ...

```
for(uint8_t i = 0; i < 8; i++)
{
   TxData[i] = (i + 10);
}
```

Start CAN...

```
HAL_CAN_Start(&hcan);
```

If there are several CANs, then we launch the first one ...

```
HAL_CAN_Start(&hcan1);
```

And we activate events that will cause interrupts ...

```
HAL_CAN_ActivateNotification(&hcan, CAN_IT_RX_FIFO0_MSG_PENDING | CAN_IT_ERROR | CA
```

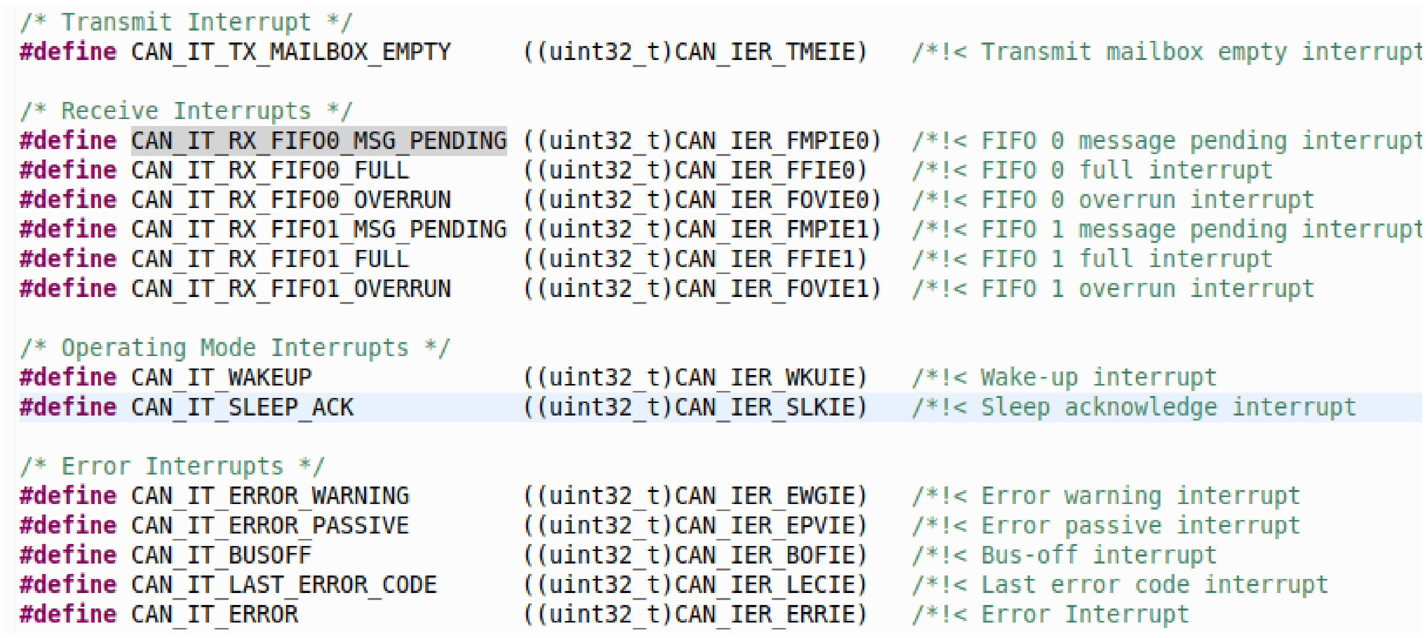
CAN_IT_RX_FIFO0_MSG_PENDING- will cause an interrupt when a message is received in the CAN_RX_FIFO0 buffer. Kolbek for this we registered. *If the CAN_RX_FIFO1 buffer is used, then the macro is CAN_IT_RX_FIFO1_MSG_PENDING.*

If both buffers are used, then so...

```
HAL_CAN_ActivateNotification(&hcan, CAN_IT_RX_FIFO0_MSG_PENDING | CAN_IT_RX_FIFO1_M
```

The rest is interruptions due to various errors. We also prescribed a callback for them. By and large, callback.

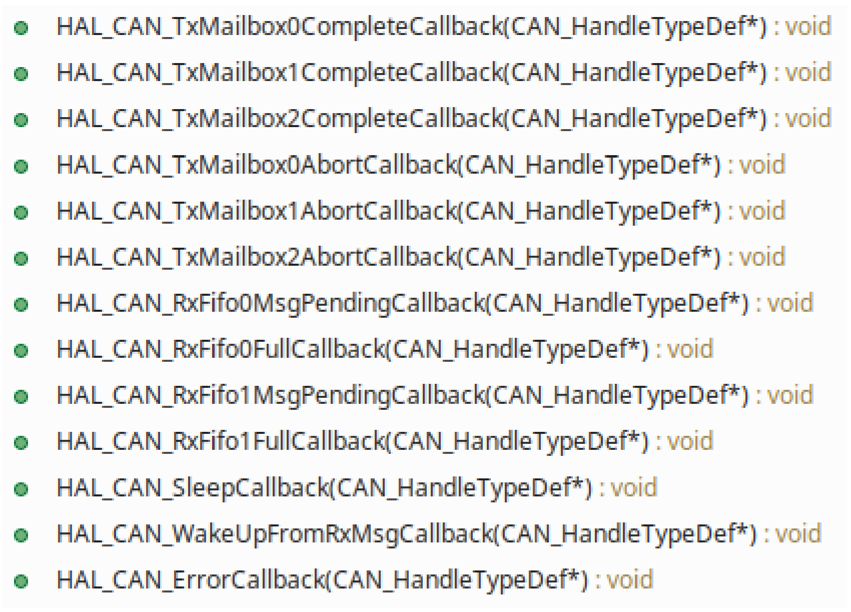
Here are all the possible events that can be added to the function...



```
/* Transmit Interrupt */
#define CAN_IT_TX_MAILBOX_EMPTY     ((uint32_t)CAN_IER_TMEIE)   /*!< Transmit mailbox empty interrupt

/* Receive Interrupts */
#define CAN_IT_RX_FIFO0_MSG_PENDING ((uint32_t)CAN_IER_FMPIE0)  /*!< FIFO 0 message pending interrupt
#define CAN_IT_RX_FIFO0_FULL        ((uint32_t)CAN_IER_FFIE0)   /*!< FIFO 0 full interrupt
#define CAN_IT_RX_FIFO0_OVERRUN     ((uint32_t)CAN_IER_FOVIE0)  /*!< FIFO 0 overrun interrupt
#define CAN_IT_RX_FIFO1_MSG_PENDING ((uint32_t)CAN_IER_FMPIE1)  /*!< FIFO 1 message pending interrupt
#define CAN_IT_RX_FIFO1_FULL        ((uint32_t)CAN_IER_FFIE1)   /*!< FIFO 1 full interrupt
#define CAN_IT_RX_FIFO1_OVERRUN     ((uint32_t)CAN_IER_FOVIE1)  /*!< FIFO 1 overrun interrupt

/* Operating Mode Interrupts */
#define CAN_IT_WAKEUP               ((uint32_t)CAN_IER_WKUIE)   /*!< Wake-up interrupt
#define CAN_IT_SLEEP_ACK            ((uint32_t)CAN_IER_SLKIE)   /*!< Sleep acknowledge interrupt

/* Error Interrupts */
#define CAN_IT_ERROR_WARNING        ((uint32_t)CAN_IER_EWGIE)   /*!< Error warning interrupt
#define CAN_IT_ERROR_PASSIVE        ((uint32_t)CAN_IER_EPVIE)   /*!< Error passive interrupt
#define CAN_IT_BUSOFF               ((uint32_t)CAN_IER_BOFIE)   /*!< Bus-off interrupt
#define CAN_IT_LAST_ERROR_CODE      ((uint32_t)CAN_IER_LECIE)   /*!< Last error code interrupt
#define CAN_IT_ERROR                ((uint32_t)CAN_IER_ERRIE)   /*!< Error Interrupt
```

stm32f1xx_hal_can.h

And various callbacks...



- HAL_CAN_TxMailbox0CompleteCallback(CAN_HandleTypeDef*) : void
- HAL_CAN_TxMailbox1CompleteCallback(CAN_HandleTypeDef*) : void
- HAL_CAN_TxMailbox2CompleteCallback(CAN_HandleTypeDef*) : void
- HAL_CAN_TxMailbox0AbortCallback(CAN_HandleTypeDef*) : void
- HAL_CAN_TxMailbox1AbortCallback(CAN_HandleTypeDef*) : void
- HAL_CAN_TxMailbox2AbortCallback(CAN_HandleTypeDef*) : void
- HAL_CAN_RxFifo0MsgPendingCallback(CAN_HandleTypeDef*) : void
- HAL_CAN_RxFifo0FullCallback(CAN_HandleTypeDef*) : void
- HAL_CAN_RxFifo1MsgPendingCallback(CAN_HandleTypeDef*) : void
- HAL_CAN_RxFifo1FullCallback(CAN_HandleTypeDef*) : void
- HAL_CAN_SleepCallback(CAN_HandleTypeDef*) : void
- HAL_CAN_WakeUpFromRxMsgCallback(CAN_HandleTypeDef*) : void
- HAL_CAN_ErrorCallback(CAN_HandleTypeDef*) : void

stm32f1xx_hal_can.c

I think that everything is clear from the names. And again, you can safely do without these interruptic

Now let's add one filter that will pass all messages. *At least one filter must be configured otherwise it not work.* We go down and find the functionstatic void MX_CAN_Init(void).

Adding a structure to it for configuring filters...

```
/* USER CODE BEGIN CAN_Init 0 */
CAN_FilterTypeDef  sFilterConfig;
```

And filter settings...

```
/* USER CODE BEGIN CAN_Init 2 */
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;
//sFilterConfig.SlaveStartFilterBank = 14;

if(HAL_CAN_ConfigFilter(&hcan, &sFilterConfig) != HAL_OK)
{
Error_Handler();
}
/* USER CODE END CAN_Init 2 */
```

Filters will be discussed in detail in the next part, but here is just a brief explanation.

Each CAN module has 14 filters. F103 has one CAN module, respectively, it has 14 filters from 0 to 1
Stones that have two CAN modules have 28 filters available, from 0 to 13 for CAN_1, and from 14 to
for CAN_2. Each filter is called a "bank" and has a sequence number. In this case, we use filter bank
number 0 *(the first element of the structure)* .

Missing multiple elements*(they contain the identifiers of the frames to be received)*and we see that w
have written the CAN_RX_FIFO0 macro in the FilterFIFOAssignment element. This means that the f
will work with the RX_FIFO_0 buffer. Since we have not configured any filters for the RX_FIFO_1 buf
it will not take part in the work, all messages will only come to the RX_FIFO_0 buffer. *If you configure
an interrupt for the RX_FIFO_1 buffer, then you need to write CAN_RX_FIFO1 to the
FilterFIFOAssignment.*

To use both buffers, you need to configure at least two filters for different identifiers, and then it will b
possible to skip some messages into one buffer and others into the other. Then it would make sense
use both buffers.

At the end, the filter configuration function is called.

As a result, the CAN initialization function will look like this ...

```
static void MX_CAN_Init(void)
{
  /* USER CODE BEGIN CAN_Init 0 */
  CAN_FilterTypeDef  sFilterConfig;
  /* USER CODE END CAN_Init 0 */

  /* USER CODE BEGIN CAN_Init 1 */

  /* USER CODE END CAN_Init 1 */
  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 = 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(&hcan, &sFilterConfig) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE END CAN_Init 2 */

}
```

And finally, the last thing to do is to add sending messages to the endless loop ...

```
/* USER CODE BEGIN WHILE */
while (1)
{
        while(HAL_CAN_GetTxMailboxesFreeLevel(&hcan) == 0);

        if(HAL_CAN_AddTxMessage(&hcan, &TxHeader, TxData, &TxMailbox) != HAL_OK)
        {
                HAL_UART_Transmit(&huart1, (uint8_t*)"ER SEND\n", 8, 100);
        }

        HAL_Delay(500);
}
```

FunctionHAL_CAN_GetTxMailboxesFreeLevel(...)returns the number of free boxes to send *(as you remember, we have three of them)* , that is, it should return 1, 2 or 3. If all the boxes are busy *(messa did not have time to fly)* then 0 will be returned. Accordingly, we slow down the program until it is free although one box.

FunctionHAL_CAN_AddTxMessage(...)sends a message to the mailbox *(from there it flies automatically)* . As arguments, we pass the structure in which we have the message settings *(identifi etc.)* , and an array with useful data. The last argument is of no interest to us.

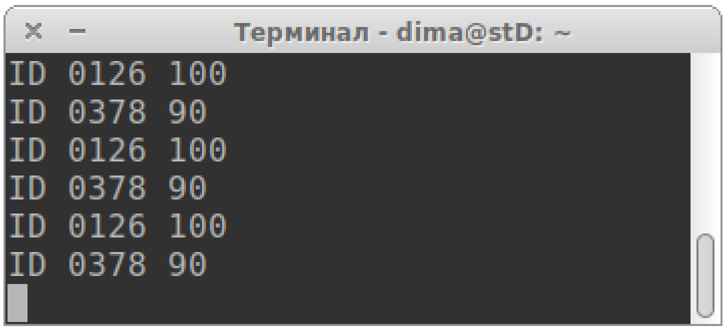
Actually, that's all, you can flash it. The board will send messages to itself, and the light will blink in th receive callback.

To make it a little more interesting, we will send messages with different identifiers and change the ze element in the array with useful data ...

```
/* USER CODE BEGIN WHILE */
while (1)
{
        TxHeader.StdId = 0x0378;
        TxData[0] = 90;

        while(HAL_CAN_GetTxMailboxesFreeLevel(&hcan) == 0);

        if(HAL_CAN_AddTxMessage(&hcan, &TxHeader, TxData, &TxMailbox) != HAL_OK)
        {
                HAL_UART_Transmit(&huart1, (uint8_t*)"ER SEND\n", 8, 100);
        }

        HAL_Delay(500);


        TxHeader.StdId = 0x0126;
        TxData[0] = 100;

        while(HAL_CAN_GetTxMailboxesFreeLevel(&hcan) == 0);

        if(HAL_CAN_AddTxMessage(&hcan, &TxHeader, TxData, &TxMailbox) != HAL_OK)
        {
                HAL_UART_Transmit(&huart1, (uint8_t*)"ER SEND\n", 8, 100);
        }

        HAL_Delay(500);

/* USER CODE END WHILE */

/* USER CODE BEGIN 3 */
}
```

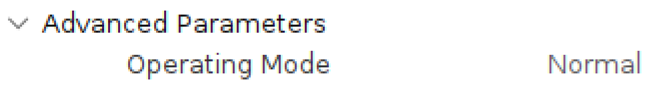
In the callback, we will check the identifiers and display the information in the USART ...

```
void HAL_CAN_RxFifo0MsgPendingCallback(CAN_HandleTypeDef *hcan)
{
    if(HAL_CAN_GetRxMessage(hcan, CAN_RX_FIFO0, &RxHeader, RxData) == HAL_OK)
    {
        HAL_GPIO_TogglePin(GPIOC, GPIO_PIN_13);

        if(RxHeader.StdId == 0x0378)
        {
                snprintf(trans_str, 128, "ID %04lX %d\n", RxHeader.StdId, RxData[0]);
                HAL_UART_Transmit(&huart1, (uint8_t*)trans_str, strlen(trans_str), 100);
        }
        else if(RxHeader.StdId == 0x0126)
        {
                snprintf(trans_str, 128, "ID %04lX %d\n", RxHeader.StdId, RxData[0]);
                HAL_UART_Transmit(&huart1, (uint8_t*)trans_str, strlen(trans_str), 100);
        }
    }
}
```



```
Терминал - dima@stD: ~
ID 0126 100
ID 0378 90
ID 0126 100
ID 0378 90
ID 0126 100
ID 0378 90
```

If you have two boards with transceivers, then you can set the Normal mode...



```
Advanced Parameters
    Operating Mode          Normal
```

And upload this project to both boards - they will begin to exchange data with each other. Just write different identifiers for sending for different boards, in an endless loop.